Natural Computing Series

Shigeyoshi Tsutsui
Pierre Collet *Editors*

Massively Parallel Evolutionary Computation on GPGPUs

Springer

Natural Computing Series

Shigeyoshi Tsutsui • Pierre Collet
Editors

Massively Parallel Evolutionary Computation on GPGPUs

Springer

Editors

Shigeyoshi Tsutsui
Dept. of Management and
Information Science
Hannan University
Matsubara, Osaka
Japan

Pierre Collet
ICube Laboratory UMR CNRS 7357
Université de Strasbourg
Illkirch
France

Series Editors
G. Rozenberg (Managing Editor)

Th. Bäck, J.N. Kok, H.P. Spaink
Leiden Center for Natural Computing
Leiden University
Leiden, The Netherlands

A.E. Eiben
Vrije Universiteit Amsterdam
The Netherlands

ISSN 1619-7127 Natural Computing Series
ISBN 978-3-642-37958-1 ISBN 978-3-642-37959-8 (eBook)
DOI 10.1007/978-3-642-37959-8
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013952962

ACM Computing Classification (1998): F.2, G.1, C.1, I.2, I.4, F.1

To our wives Kumiko and Valérie

Foreword

During my search for an evolutionary algorithm able to self-adapt its internal parameters like mutation strengths and correlations, I had to leave established paths plastered already with one-variable-at-a-time and steady-state approaches for optimization purposes. The latter operates with just one individual created anew by mutation and crossover from the survivors of past selection steps. Creating and evaluating several descendants before the next selection operation, an idea immediately damned as wasteful, especially when not merely the best was taken as the single parent of the next generation, in my mind cried out for parallel processing resources like digital computers. That was in the early 1970s and I was taught by experts of renowned computer firms that such devices would not be built before the end of that century. They were wrong. Already in the mid-1980s I was able to make use of a Cray machine at a big research center.

However, that was a so-called SIMD (single-instruction multiple-data) machine. Neither an SPMD (single-program multiple-data) nor an MIMD (multiple-instructions multiple-data) computer was available back then. The Cray turned out to be useful for testing self-adaptive evolutionary algorithms, but only on the usual benchmark functions with rather short and nearly equal evaluation times for different sets of variables. In practice the evaluation of different individuals requires the running of a complex simulation model and takes long and largely unequal times. Waiting for the longest run until the selection routine can start means idling of all other devices used for the simulation step and thus an unnecessary loss of computing capacities. Therefore I appreciated very much the arrival of the new transputers which could be programmed to handle asynchronous evolutionary processes with message passing software to facilitate the communication between the (in my best case up to 64 in the 1990s) transputers.

Now you have CPUs with multiple cores as well as parallel-core GPUs. And you can combine them into powerful hybrid systems. Even GPGPU cards (general-purpose computing on graphics processing units) and large amounts of shared memory are available and affordable so that you can build your own heterogeneous machines that are best suited to solve your special tasks optimally.

It is difficult to gain an overview of today's complex world of computing devices all operating principally in parallel but not necessarily synchronized. Two leading experts in this modern field will guide you from the theoretical background to the best practice for real-world applications in this edited volume comprising all important information you need for getting hands-on at your own tasks.

Dortmund, Germany Hans-Paul Schwefel
January 2013

Preface

Evolutionary algorithms (EAs) are generic heuristics that learn from natural collective behavior and can be applied to solve very difficult optimization problems. Applications include engineering problems, scheduling problems, routing problems, assignment problems, finance and investment problems, analyzing problems of complex social behavior, to state a few.

One of the most important features of EAs is that they are population-based searching algorithms, i.e., they work using one or more sets of candidate solutions (individuals). As a result, EAs deal well with the EvE (Exploration vs. Exploitation) paradox and are ideally suited for massively parallel runs, at the operator level, individual level, and population level and can therefore exploit efficiently general-purpose graphics processing units (GPGPUs) as well as massively parallel supercomputers. Because EAs are well suited to be applied to complex problems, they may require a long time to obtain good solutions or may require a large amount of computational resources to obtain high-quality solutions in a given time. Thus, since the earliest studies, researchers have attempted to use parallel EAs in order to accelerate the execution of EAs.

Many of the traditional parallel EAs run on multi-core machines, massively parallel cluster machines, or grid computing environments. However, recent advances in scientific computing have made it possible to use GPGPUs for parallel EAs. GPGPUs are low-cost, massively parallel, many-core processors. They can execute thousands of threads in parallel in single-program multiple-data (SPMD) manner. With the low-cost GPGPUs found in ordinary PCs, it is becoming possible to use parallel EAs to solve optimization problems of all sizes.

Decreasing execution time by orders of magnitude opens new possibilities for many researchers and EA users who were until now limited by the limited computing power of CPUs. At the time of writing, current GPGPU cards computing power are in the order of 10 TFlops, turning any PC into a high-performance parallel processing computer. Essentially, this provides anyone with the processing power previously available only to those with access to expensive supercomputers. Not only is there benefit to solve large problems, but also many small problems can be run many times in order to optimize parameter tuning.

The impact on optimization and inverse problem solving should be huge and this book should be of interest to anyone involved with EAs, including researchers, scientists, professionals, teachers, and students in the field.

Although the parallelism of EAs is well suited to SPMD-based GPGPUs, there are many issues to be resolved in the design and implementation of EAs on GPUs. For example, EAs run with a certain degree of randomness which can bring divergence that could degrade performance on SIMD cores. Some chapters of this book propose solutions to these kinds of issues in solving large-scale optimization problems.

This book provides a collection of 19 chapters ranging from general tutorials to state-of-the-art parallel EAs on GPUs to real-world applications that will hopefully help those who would like to parallelize their work on GPGPU cards.

The three chapters of Part I (Tutorials) introduce the problem, and present the hardware specificities of GPGPU cards as well as an easy way to start using GPGPUs for evolutionary computation, using the EASEA platform that was developed in Strasbourg.

Part II presents more advanced implementations of various kinds of EAs on GPGPUs, in order to show what can be done and serve as inspiration for new developments.

Part III is more concerned with real-world applications, showing that evolutionary GPGPU computing has come of age and is not confined to solving toy problems.

We would like to end this preface with our deepest thanks to all authors for their effort, help, and their pioneering contributions to the topics they have written about. Thanks to their commitment, we believe that this volume can have a great impact on our respective fields. Also, we would like to thank Prof. A. E. Eiben and Prof. Hans-Paul Schwefel for encouraging the publication of this volume, and Ronan Nugent of Springer for his continuous help throughout the publishing stages of this book.

Osaka, Japan — Shigeyoshi Tsutsui
Strasbourg, France — Pierre Collet
January 2013

Contents

Part I
Tutorials

Why GPGPUs for Evolutionary Computation?

Pierre Collet

Abstract In 2006, for the first time since they were invented, processors stopped running faster and faster, due to heat dissipation limits. In order to provide more powerful chips, manufacturers then started developing multi-core processors, a path that had already been taken by graphics cards manufacturers earlier on. In 2012, NVIDIA came out with GK110 processors boasting 2,880 single precision cores and 960 double precision cores, for a computing power of 6 TFlops in single precision and 1.7 TFlops in double precision. Supercomputers are currently made of millions of general purpose graphics processing unit cores which poses another problem: what kind of algorithms can exploit such a massive parallelism? This chapter explains why and how artificial evolution can exploit future massively parallel exaflop machines in a very efficient way to bring solutions to generic complex inverse problems.

1 Introduction

The context in computer science is changing: ever since the first computers were manufactured in the 1950s, CPU clock speed kept increasing in a pretty exponential way until about 2005, when it reached a maximum with a 3.8 GHz Pentium 4 570 (cf. Fig. 1).

This plateau comes from the physical fact that CPU energy consumption more or less varies as the cube of clock rate. By 2005, a maximum was reached in the amount of heat that could be evacuated through the small surface of a CPU chip. Doubling frequency speed from 3.8 to 7.6 GHz would mean that it would be necessary to extract eight times more heat from the same surface which is impossible using

P. Collet (✉)
ICUBE, Strasbourg University, Strasbourg, France
e-mail: pierre.collet@unistra.fr

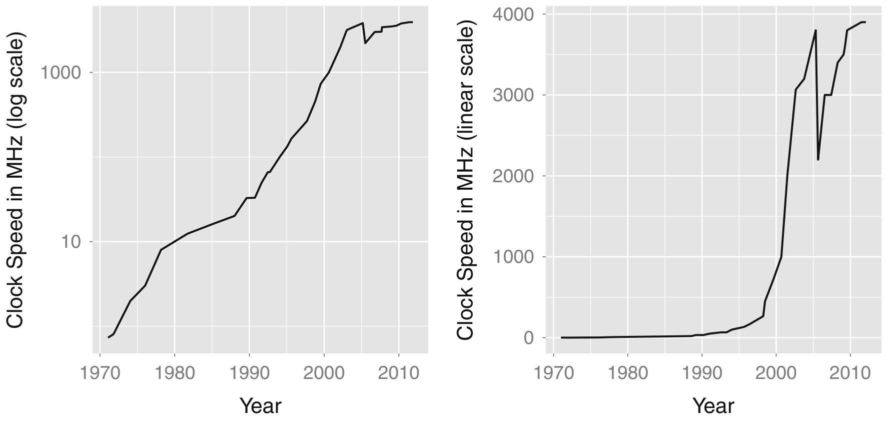


Fig. 1 Evolution of clock speed for Intel processors from 1971 to 2012, gleaned on http://en.wikipedia.org/wiki/List_of_Intel_microprocessors. The fact that the curve looks quite linear up to 2005 on the *left* figure (that uses a log scale for the *Y* axis) shows that the progression was quite exponential. Then one notices a drop and the curve goes nearly horizontal. On the linear scale curve (*right*) one sees that since 2005, progression has stopped and is capped by 4 GHz

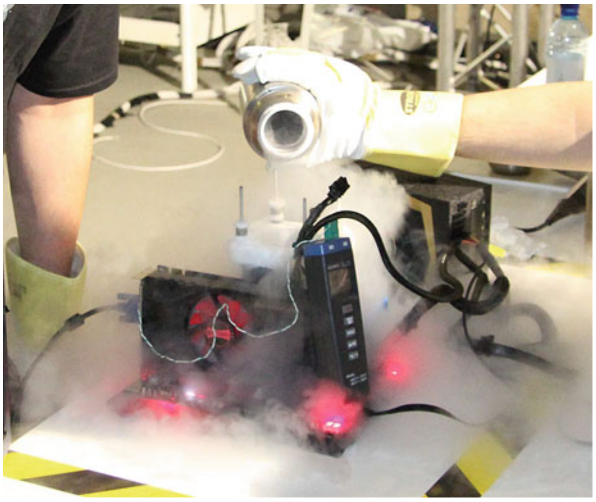


Fig. 2 It is possible for a standard chip to run faster than 4 GHz, but with a special cooling system

conventional heat dissipation technology. Going beyond 4 GHz is possible, as shown in Fig. 2, but not very practical (liquid nitrogen at −195 °C is constantly poured on the chip to cool it down).

Would this mean that the maximum computing power for a chip was reached by 2005? This would be without taking Moore's law [2] into account, which states that transistor density on silicon doubles every other year. It appears that this observed law may still apply until at least 2025.

In principle, having twice as many transistors per square millimetre does not improve computer performance, unless one uses the additional transistors to implement two CPUs on the same chip (dual core technology).

Indeed, doing this would allow us to double the number of performed operations per second by "only" multiplying consumption by two, rather than by eight if doubling the number of operations per second was obtained by doubling clock frequency.

If maximum heat dissipation was reached by 3.8 GHz chips in 2005, then dual core chips would need to run slower, but not at half the speed of a single core chip because consumption varies as the cube of the frequency. Therefore, in order to reduce consumption by 2, one only needs to reduce clock frequency by the cubic root of 2, i.e. 1.26. Then, it also happens that smaller transistors consume less, meaning that doubling the number of cores thanks to the ability to use smaller transistors would allow us to reduce clock speed by less than 1.26.

This discussion only takes clock frequency and transistor density into consideration, where so many other factors play important roles into computing power, so it must be taken with a pinch of salt, but it is nevertheless useful to understand the new and important trend that started in 2005: increase in computing power will come through parallelism rather than through the acceleration of single cores.

The implication of this new direction in the development of CPUs is huge in computer science and software development: indeed, as most computers were using single core CPUs until 2005, the vast majority of existing software was designed to run sequentially on single core CPUs at a time when it seemed that the clock frequency exponential increase (see again Fig. 1) would never stop!

Unfortunately, clock frequency reached a maximum in 2005, and, since then, CPU computing power develops thanks to Moore's law, i.e. with multi-core chips whose frequency is capped by an invisible 4 GHz barrier (current CPUs can now reach this top frequency because they use variable speed cores, meaning that when one core is used at 3.90 GHz, the others slow down in order to keep heat production within a tolerable range).

It seems that unless a new technology breakthrough happens that allows us to break this barrier, Seymour Cray may have been wrong with his famous quip: "If you were ploughing a field, which would you rather use: two strong oxen or 1024 chickens?"

This first chapter will start by describing how general purpose graphics processing unit (GPGPU) chips developed, before giving hints on how and why they will very probably revolutionize evolutionary computing and scientific computing in general, thanks to evolutionary and complex systems.

2 Development of GPU Chips

In parallel with the development of CPUs, the very powerful gaming industry had devised its own processors called graphics processing units or GPUs based on different principles than those that governed the development of CPUs. Indeed, where most computer algorithms are sequential, it so happens that the vast majority of computer graphics algorithms are inherently parallel. There are basically two kinds of paradigms:

Pixel-based algorithms such as those used on photographs, when one wants, for instance, to zoom in on a specific part of a photo.

Vertex-based algorithms take into account a 3D representation of a scene that must be rendered thanks to rendering or shading, for immersive 3D graphics such as pioneered by Doom.

Both paradigms involve running the very same algorithm on millions of independent pixels in the first case or millions of independent vertices/polygons in the second case. This means that even though such algorithms could run on sequential processors, they could also exploit multi-core processors in an efficient way, meaning that they could greatly benefit from the added computing power that is made available by running more cores at a slower speed.

GPU hardware designers saw this very early and proposed parallel processors for graphics rendering many years ago. For a while, processing power was not large enough so the processors included hardwired micro-code to efficiently run pixel-based or vertex-based algorithms independently. However, this meant that when the card was used on a raster image, the transistors of the GPU that implemented vertex-based algorithms were unused and conversely when 3D representations were being rendered.

In the beginning of the 2000s, the computing power of these GPUs was such that it was finally possible to envisage generic cores that would either run pixel-based or vertex-based algorithms stored as software programs, rather than hardwired as was previously the case. In 2006, the G80 chip of NVIDIA's 8800GTX card boasted 128 general purpose cores that would provide a fantastic computing power of 518 GFlops (to be compared with the 12–15 GFlops of the Intel Core 2 Duo chip of the same era). GPGPU cards were born, and specific APIs and SDKs were released by manufacturers to allow us to program them accurately.

While AMD chose to rely on higher-level environments such as OpenCL (Open Computing Language) to deal with higher-level parallelism, NVIDIA chose to develop a lower level environment and released CUDA (Compute Unified Device Architecture) in 2004 [3]. NVIDIA cards are widely used in scientific computing (and throughout this book) probably due to these strategic choices, so we will concentrate on those cards.

3 Some Considerations on Hardware Design

Constraints of graphics rendering were used to design GPUs and a certain number of specificities were used to simplify the architecture of these chips that ended up in allowing us to implement more generic computing cores in order to maximize computing power.

First of all, one must understand that a graphics card is a fully fledged massively parallel computer that is totally independent from the host. This means that no memory is shared between the graphics card and the host system. In order to use the card, one needs to prepare some object code compiled for the GPGPU chip (using an `nvcc` compiler, for instance, if one uses an NVIDIA card) as well as some data, then dump everything on the graphics card and remotely start execution from the host. When the massively parallel calculation is finished, then one must transfer back all the processed data to the host CPU.

Typically, identical pixel-based or vertex-based algorithms need to be performed on independent pixels or polygons, so rather than implementing independent cores in GPU as is done in multi-core CPUs, it is possible to exploit a Single-Instruction Multiple Data (SIMD) architecture where several cores execute the same instruction at the same time, but on different data (pixels or vertices). This is a great simplification of the architecture because, for example, only one Fetch and Dispatch unit can be used to find and load the next instruction for several cores.

SPMD Architecture

Ideally, if all the cores were executing the very same algorithm on many different pixels or vertices, all the cores could run in SIMD mode. However, probably in order to add some flexibility in the software design, NVIDIA chose to create clusters of 32 SIMD cores called multi-processors that have their own program counter and Fetch and Dispatch unit. This allows GPGPUs to implement Single-Program Multiple-Data (SPMD) parallelism.

This means that even though the complete multi-core chip can only execute a single program at a time, different multi-processors can run on different parts of the same program. This provides some flexibility while making it possible to avoid implementing complex context swapping between different address spaces: only minimal (and very fast) scheduling capabilities are needed to switch between different pixels if this is done within the same program. This particular architecture is explained in the chapter “Understanding NVIDIA GPGPU Hardware”.

Spatio-temporal Parallelism (Pipelining Replaces Cache Memory)

Typically, pixel-based or vertex-based algorithms need to deal with many more pixels or polygons than there are cores in a GPU. This specificity allows GPU manufacturers to do without cache memory (only a very small amount of very fast shared memory is available within multi-processors) and use the spared transistors to implement yet more computing cores.

Where instruction or data cache memory is normally used to store enough instructions or data to perform very fast on-chip memory accesses only, applying this to a graphics chip would mean that all the pixels or the vertices would need to be stored in cache memory, because typically pixels or vertices are accessed only in a limited number of times: once the new value of a pixel is calculated, the algorithm will work on another different pixel. Storing a pixel into cache memory would not provide great acceleration because rendering algorithms that access a single pixel hundreds of times are very rare.

Because cache memory is not adapted to graphics rendering, manufacturers decided to do without cache memory and benefit from SPMD parallelism to implement very efficient hardware scheduling. If a particular pixel is not immediately available, this automatically schedules calculation on the next pixel, and so on, until the data concerning the first pixel is finally available, possibly several hundreds of cycles later.

This very fast and efficient scheduling is only possible because the processor need not switch between totally different contexts. Whenever scheduling between pixels or vertices is needed to accommodate for the slow cache-less memory, a new set of registers is stacked over the previous registers to allow the calculation to be performed on a new pixel or vertex. This implements a very efficient automatic pipelining mechanism that uses thousands of registers to allow a single core to implement temporal parallelism on different pixels or vertices.

Parallelizing a rendering program on many different cores, each implementing pipelining on several pixels or vertices results in implementing very efficient "spatio-temporal" parallelism. One of the implications of such parallelism is that, to run efficiently, a GPGPU card needs to work on many more pixels or vertices than it has available cores. If a 1,024-core GPU is used on only 1,024 different pixels or vertices, one will not benefit from pipelining, and because virtually no data cache is available, all memory accesses will be very slow.

Where standard CPUs do not like to be overloaded because of the risk to use more space than is available in cache memory, resulting in memory thrashing between cache memory and main memory, GPUs like to be overloaded (within the number of registers available on the GPU processor to stack several tasks) because it allows their very efficient hardware scheduling mechanism to compensate for the lack of a large cache memory.

Very High Memory Transfer Bandwidth Between GPU Card and Host Computer

Because graphics cards need to dump very heavy images to the visual display unit of the computer many times per second, they implement very fast CPU–GPU bus communication with bandwidth of several hundred gigabytes per second. This means that once transmission between the computer and the card is initiated, data transmission is extremely fast.

This is very nice in the case of algorithms that need to communicate with the CPU at every generation, because it allows us to dump the whole population to or from the GPU in virtually no time should a particular type of algorithm need this feature (a memetic algorithm [4], for instance, if it was only partially parallelized). Assuming the evolutionary loop remains on the CPU and only the evaluation function (that implements a local optimizing method) is parallelized over the GPU card, then the complete genome of the evaluated (and locally optimized) children must be sent back to the host CPU for the next generation (one cannot only send the evaluation values because the genome gets modified in the evaluation function that is running on the GPU card).

Very high memory transfer bandwidth makes this possible without much overhead as only two memory transfers are needed per generation. As a result, memory transfer between the CPU and GPU is hardly measurable, even with large populations of large individuals (cf. Fig. 7 of the chapter "Generic Local Search (Memetic) Algorithm on a Single GPGPU Chip").

Coalescent Memory Accesses

Because some graphics rendering algorithms may need to repeat the same instructions on neighbouring pixels, GPGPU card manufacturers have implemented "coalescent" memory accesses that are much faster than random memory accesses. Very fast coalescent memory accesses take place when SIMD cores of the same multi-processor simultaneously access contiguous data in memory. Rather than looking for pieces of memory scattered everywhere in the global memory of the card, a chunk of memory is directly accessed and made available to all the cores in one step, which saves a lot of time.

All these specificities mean that parallelizing an existing program containing sequential parts to get it to run efficiently on GPGPU cards is not an easy task. On the contrary, if one implements an embarrassingly parallel algorithm from scratch on these cards, it is possible to use the special features described above to get the most out of the graphics card. This is why they are so well suited to implementing evolutionary computation, which is an embarrassingly parallel generic solver.

4 Why GPGPU Chips are Ideal for Artificial Evolution and Other Complex Systems

All this means that programs that run efficiently on CPUs may not run well on GPUs and conversely. Indeed, the latest GK110 chips (featured on K20 and K20X Tesla cards) contain 14 multiprocessors that each contain 192 single precision (SP) and

64 double precision (DP) cores for a total of 3,584 cores, each of which must be loaded with several tasks (at least one order of magnitude higher in our experience with artificial evolution on GPU chips) for efficient pipelining to take place to make up for the virtual absence of cache memory.

In order to run efficiently on a K20X card, a single program would need to be divided into an excess of 35,000 independent tasks, knowing that these tasks should be grouped on 192 SP + 64 DP = 256 core SIMD multiprocessors that should execute the same instruction at the same time. Although this perfectly suits the needs of computer graphics algorithms (which run identical tasks on millions of different pixels or polygons), such hardware is mostly unadapted for standard algorithms that most probably cannot be decomposed into so many independent yet identical tasks, because most standard algorithms were originally designed to run on single core machines.

In order to use these GPUs efficiently, it is therefore necessary to either design new algorithms adapted to their special architecture or adapt existing algorithms whose flowchart could be close to that of rendering algorithms, for which these cards have originally been designed.

A complex system can be defined as any system comprised of a great number of autonomous entities, where local interactions among entities create multiple levels of collective structure and organization. Typically, complex systems are good candidates for an efficient execution on a GPGPU graphics card: all the cores can emulate autonomous entities and interaction can occur through the global memory shared between the cores. Among all types of complex systems, artificial evolution is known to be a generic solver of inverse problems.

4.1 *Partial Parallelization and Amdahl's Law*

A simple way to exploit a parallel machine with an evolutionary algorithm consists in parallelizing the evaluation step. First of all, this ensures that the parallelized algorithm is strictly identical to the sequential algorithm. Secondly this method is very simple to implement and is also very efficient, provided that evaluation time is long compared to all the other steps of the evolutionary loop, which is fortunately quite often the case.

But in 1967, Gene Amdahl came out with a law (Amdahl's law [1]) that states that if P is the proportion of an algorithm that can be parallelized and $(1 - P)$ is the sequential proportion of the same algorithm, then the maximum achievable speedup with N cores is

$$S(N) = \frac{1}{(1 - P) + P/N}.$$

If N tends to infinity, the maximum speedup is then $1/(1 - P)$.

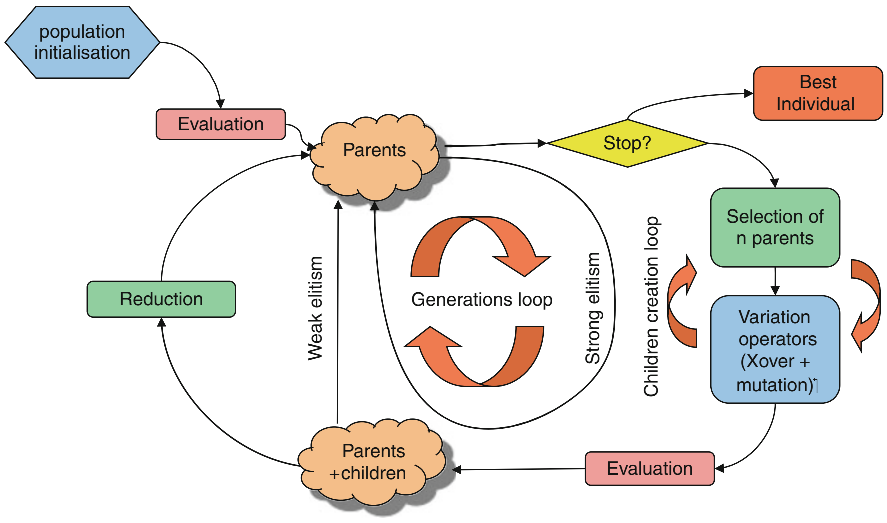


Fig. 3 Generic evolutionary loop

In other words, if a program has 10 % sequential code and 90 % perfectly parallelizable code, then, if this program is run on a parallel machine with an infinite number of cores, only the 10 % sequential part will remain, meaning that the maximum speedup that can be achieved by using a parallel machine is only 10×.

In order to obtain a 100× speedup with a parallel machine with an infinitely large number of cores, one needs to run an algorithm whose parallel part represents 99.9 % of the algorithm. This means that only highly parallelizable programs can achieve such a speedup.

4.2 *Full Parallelization for Maximum Speedup*

Fortunately, evolutionary algorithms are *embarrassingly parallel*, meaning that it should be possible to parallelize them completely. Indeed, supposing one considers the generic evolutionary loop of Fig. 3, running a generational GA on a 1,024-core (32 SIMD multiprocessors of 32 cores each) GPGPU card with a population size of 10,240 individuals (10 individuals per core to exploit pipelining) located in global memory:

1. In order to initialize the population, one can ask each of the 1,024 cores to randomly and independently create ten individuals. Because the initialization procedure will be identical for all individuals, it is likely that all tasks will be compatible with a near-perfect SIMD execution on the 32 cores of the 32 multiprocessors of the card. Then, individually, each core has to create

and initialize ten individuals, allowing the hardware scheduler of the card to implement temporal parallelization through a pipelining process.

2. The next step is to evaluate all individuals of the population. Once more, each of the 1,024 cores can evaluate ten individuals in an independent and mostly SIMD manner (things are slightly different for genetic programming, but evaluation of different GP individuals can also be done very efficiently using SPMD and SIMD parallel architectures as described in the chapter "Genetic Programming on GPGPU Cards Using EASEA").
3. Then, each core can check whether any of the 10 individuals it evaluated meets a particular stopping criterion or if the planned number of generations (or maximum allowed runtime) was reached. If this is the case, one or more cores can stop the evolutionary loop, and, of course, the best of all the individuals in the population will be the result of the evolutionary algorithm. This last step is not perfectly parallel, but it occurs once in the whole artificial evolution program, and checking which is the best individual is not a particularly time-consuming task.
4. If no stopping criterion was met, one can ask each core to select *n* parents among the best and perform variation operators on the parents to create one child and repeat the operation ten times (if we run a generational GA, it is necessary to create as many children as there are parents in the population). Children creation can be done independently by each core without requiring any kind of communication between the cores. The children creation procedure being identical for all cores, this step will also be mostly SIMD and therefore very efficiently parallelized on a GPU chip.
5. Once all cores created their ten children, it is necessary to evaluate them. This can be done in parallel as in step 2.
6. In the case of a generational GA, reduction is implemented in a very simple manner: all parents are removed and the next generation starts using the children as population. In practice, this can be done by allocating two populations and exchanging pointers at each "reduction" phase.

So one can see that evolutionary algorithms are perfectly well fitted to execute very efficiently on SPMD/SIMD hardware, with experienced accelerations beyond 100× obtained routinely, as described in the different chapters of this book.

5 Parallelizing over Several Machines Thanks to an Island Model

This book is mainly about parallelizing evolutionary algorithms over one or several GPGPU cards. It is however important to mention the fact that evolutionary algorithms parallelize very well over many loosely coupled computers through an island model.

The idea is that different computers evolve independent populations over the same problem. If the search space is large enough (typically the case that would justify using several machines) and the initialization of individuals was done using random values, it is very probable that different machines will end up in different regions of the global search space, because evolutionary algorithms are stochastic.

The typical difficulty with problem solvers is to find the best compromise between exploration and exploitation. Too much exploitation and the algorithm will very likely prematurely converge towards a local optimum, and, conversely, if the algorithm is tuned so that exploration is favoured, the algorithm may skip above very good valleys and may not find good solutions.

One way around this is to implement convergent evolutionary algorithms on different machines or different cores of a single machine and get the machines to cooperate to mutually help themselves out of local optima.

Suppose you wanted to cross a soft sand desert with a four-wheel-drive car. Even if you are a good driver, there are good chances that you will get stranded at the bottom of a deep soft sand dune before you reach the end of the desert, and even more so if, while you want to cross the desert, you also want to explore the deepest pits you can find in between the dunes.

Using several vehicles will not solve the problem: if one vehicle has a very good chance to get stranded, so will n vehicles, unless you allow the different 4WD vehicles to interact, by giving them a tow cable and a winch. With such equipment a stranded vehicle can ask another one to throw it a cable and tow it out of the local optimum with the winch, allowing it to explore further and, eventually, cross the desert.

The higher level ability to cross the desert when different vehicles are allowed to interact is called emergence. A set of n vehicles in interaction will not go n times faster across the desert: it will go infinitely faster than a single vehicle because the single vehicle will get stranded in a local optimum and never reach the other side of the desert, while the set of vehicles in interaction will. This is what we call a supra-linear speedup. More extensive description of this phenomenon can be found in the chapters “Automatic Parallelization of EC on GPGPUs and Clusters of GPGPU Machines with EASEA and EASEA-CLOUD” and “Using Large-Scale Parallel Systems for Complex Crystallographic Problems in Materials Science”.

6 Conclusion

Evolutionary computation is perfectly well suited for efficient execution on massively parallel GPGPU cards. This is great news because this allows us to use these very powerful cards to solve what evolutionary algorithms are good at, i.e. generic inverse problems.

Then the current trend for supercomputers is to put together thousands of nodes equipped with one or more GPU cards: the fastest supercomputer when this book was written was Titan, developed by Cray at Oak Ridge National Laboratory for use in a variety of science projects. It is made of 18,688 16-core AMD Opteron 6274 CPUs with as many 3,584-core NVIDIA TESLA K20X GPGPU cards, for a total of 267,008 CPU cores and 67 million GPU cores and an advertised computing power of around 20 petaflops.

Dealing with such massively parallel machines is not a problem for artificial evolution that is scalable over a large number of massively parallel computers using an island model, yielding supra-linear speedup thanks to emergent properties (as will be shown in several chapters of this book).

A conclusion is that evolutionary computation may be one of the very rare generic solvers of inverse problems that could efficiently exploit tomorrow's massively parallel exa- and zetaflop machines.

References

1. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18–20, 1967, Spring Joint Computer Conference, pp. 483–485. ACM, New York (1967)
2. Moore, G.: Cramming more components onto integrated circuits. Electron. Mag. **38**(8), 114–117 (1965)
3. NVIDIA. CUDA v5.0 documentation. http://docs.nvidia.com/cuda/index.html
4. Smith, J.E., Hart, W.E., Krasnogor, N. (eds.): Recent Advances in Memetic Algorithms. Springer, Berlin (2005)

Understanding NVIDIA GPGPU Hardware

Ogier Maitre

Abstract This chapter presents NVIDIA general purpose graphical processing unit (GPGPU) architecture, by detailing both hardware and software concepts. The evolution of GPGPUs from the beginning to the most modern GPGPUs is presented in order to illustrate the trends that motivate the changes that occurred during this evolution. This allows us to anticipate future changes as well as to identify the stable features on which programmers can rely. This chapter starts with a brief history of these chips, then details architectural elements such as the GPGPU core structuration, the memory hierarchy and the hardware scheduling. Software concepts are also presented such as thread organization and correct usage of scheduling.

1 Introduction

General purpose graphical processing units (GPGPUs) were introduced to users a few years ago. They are new, complex and evolving chips that deliver high theoretical computation power at relatively low cost. As the hardware evolves rapidly, programmers using such architectures face various challenges in following this evolution and anticipating future trends. Indeed, due to the approach taken by this hardware, algorithm developments have to be low level and relatively adapted to the underlying hardware.

The architectural model of these chips is different from that of standard processors, yet common details were already implemented into past architectures. But this particular combination is new, and if standard recipes can be used, they have to be

O. Maitre (✉)
Chôros Laboratory
EPFL, BP 2130, Station 16, Lausanne, Switzerland
e-mail: ogier.maitre@epfl.ch

adapted to GPGPUs. Furthermore, modern programmers are used to more standard architectures, such as *x86* processors, with powerful compilers, that present a rather simple view of the underlying hardware.

High-computing-power architectures gained in abstraction level, over the years, and the new usage trend is to rely on an abstract view of the machine and let the software fill the gap to the concrete processor.

As GPGPUs have a complex hardware model, programmers need to understand the underlying hardware in order to use it to achieve interesting speedups. This can be difficult, as this architecture is not yet fixed and modifications are constantly occurring.

1.1 History and Origins

Graphical processing units (GPUs) emerged at the end of the 1990s, with the growth of 3D computing needs. They are fast and parallel chips that implement parts of the classical 3D software 3D rendering pipeline (OpenGL and DirectX). They can apply common rendering operations such as projection and lighting calculations. With the evolving needs of 3D rendering processes, shaders appeared, which allow the programming of certain steps of the rendering pipeline. Indeed, two steps become programmable, i.e. vertex and pixel calculations that allow us to customize the process before and after the rasterization stage. This allows developers to use these devices to apply custom algorithms, for example, for shadow calculation, by implementing custom vertex or pixel shaders.

These new capabilities inspired a number of works [4–6] which diverted the 3D rendering pipeline by inserting calculations that were not graphical, on non-graphical data.

These studies faced two major difficulties. The programming model of the languages used for pixel or vertex shader programming required a difficult translation to 3D rendering paradigms, and as the chips contained an architecture for each kind of shader, with a specific programming model (vertex or pixel related), these works generally used only one, leaving the other half of the processor idle.

At the end of 2006, the manufacturer NVIDIA introduced a new architecture, the G80, which led to major changes in this method. The pixel and vertex shader computing units are unified, and the G80 only has one type of core, which is more generic than before. This modification enhances programmability and eases load balancing between the two types of computing units.

Concurrently with this new architecture, CUDA appeared, a framework designed to allow the GPU to be programmed directly. CUDA is composed of a language (C-like, extended with keywords intended to manage GPU and CPU codes in the same source), a compiler for this language and libraries to handle GPU management.

Still, the chip contains limitations, compared to standard CPUs, such as:

- Lack of call stack management, requiring complete inlining at compile time and preventing recursive functions
- Lack of hardware support for double-precision calculation
- No communication with other devices, which prevents calls to functions like *printf*.

Double precision calculation will be introduced in GT200 chips and call stacks on the Fermi architecture. However, the freedom in programming offered by CUDA allows NVIDIA to claim their architecture as being GPGPU for "General Purpose Graphical Processing Unit". A large amount of scientific work takes advantage of this new architecture and its use spread quickly. The CUDA concept was therefore born, and some of its principles took the shape that we know nowadays.

1.2 Market Considerations

GPU computing is a hot area in scientific computing, as these devices allow us to obtain very interesting speedups for low costs, which compared to more conventional parallel hardwares may seem trivial.

After the first experiments that unofficially diverted 3D rendering cards, the emergence of specialized equipment has enabled many scientists to use these cards for their own parallel implementations.

The GPU market is specialized and relatively new, especially the GPGPU one. Indeed, these devices target two types of buyers, with the same products. Yet the needs of these two categories of users are not necessarily similar. Unfortunately for scientists, the sale volumes are not the same and 3D rendering necessarily has high priority compared to scientific computing. Yet this latter is an emerging market for these products and may become more important for manufacturers.

It has to be noted that the changes made to the GPUs, for scientific computing, cause few changes to 3D rendering. Ultimately, there is a change if it allows performance gain in both markets at the same time. The change that led to the emergence of GPGPUs (unification of pixel/vertex shaders) is a good illustration of this phenomenon. This may seem like a constraint, but it is important that these architectures continue to address these two markets together, from an economic point of view. Still, this point is not the only one to be taken into account, as high innovation also characterizes the development of these devices. Furthermore, the scientific computing market is much less competitive than the 3D rendering market. Tesla cards (graphics cards without a graphics output, such as the C-D-S870, C-M-S2050, K20-X) are an illustration of the drift that could lead to the separation of these two markets, leading to less affordable devices, targeted for a niche market.

Table 1 Flynn's taxonomy with example architectures

		Instruction management	
Data management		Synchronous for several cores	Asynchronous for several cores
	Synchronous for several cores	SISD	MISD
		Von Neumann (no parallelism)	Redundancy for critical systems
	Asynchronous for several cores	SIMD	MIMD
		GPGPU, vectorial processor	Multi-processors, multi-cores

2 Generalities

2.1 Flynn's Taxonomy

Modern GPGPU processors, in particular those which concern us here (the NVIDIA GPGPUs), are SIMD/MIMD. By SIMD, we need to understand, according to Flynn's taxonomy detailed in Table 1, several calculation units performing the same task on different data (Single Instruction Multiple Data). As for the abbreviation MIMD, this is a set of computing elements that can perform different tasks on different data. Vectorial calculation units, for example, can be said to be SIMD. Indeed, in these units, the same operation is applied to all elements of a vector at the same time. A very widespread example of MIMD processors is the classic multi-core processor, where each core runs its own thread, without the constraints induced by neighbour cores at the instruction level. It should be noted that strict implementations of this taxonomy are rare. Indeed, modern processors, such as Intel multi-cores, have multiple cores (MIMD) but also implement a vectorial instruction set (SSE3, MMX, ...) and several vector registers (SIMD). Later on, we shall see exactly how these SIMD/MIMD principles are implemented into GPGPU processors.

2.2 Warp Notion

We shall see below that GPGPUs have a large number of cores and run even more tasks. These tasks are represented by threads, which contain the instructions, the program counter, the memory context and the registers on which they work. The SIMD character of a GPGPU is expressed at the logical level through the concept of a warp, which is a set of threads that are always executed together. The warps are managed by the GPGPU, and the user has no control over the creation of these warps nor their organization. We will later see this concept in more detail.

Table 2 Compute capability *w.r.t* to chip and card versions

	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
Chips	G80	G86, G9x	GT215	GT200	GF100	GF106	GK104	GK110
Cards	8800 GTX	8800 GT	GT 240	GTX 285	GTX 480	GTS 450	GTX 680	Tesla K20

2.3 Compute Capability

NVIDIA has released each of its GPGPUs under a certain compute capability. This starts at 1.0 for G80 and the latest chips have a 3.5 compute capability. This main idea is to assign to a certain feature a compute capability and to implement every feature of an $n - 1$ capability in capability n. It allows us to know that such features are available on devices with compute capability x.x and beyond. While, certain features disappear during GPGPU evolution, the vast majority of compute features follow this rule (Table 2).

Finally, compute capability does not strictly follow the raw computing power of the cards. For example, a GF100 chip with 2.0 compute capability is mounted on high-end GTX 480 cards, but a GF106 chip has 2.1 compute capability on GTS 450 cards. This is related to the fact that high-end cards were the first to be launched on the market and some minor modifications were made by the time the low-end cards were commercialized, leading to minor differences in terms of compute capability.

3 Hardware

The GPGPU has a complex hardware implementation. It is interesting to study the overall chip and the different elements that comprise it. GPGPUs have undergone large modifications during their evolution; it is interesting to study some of the major steps in their evolution to draw a clearer picture of the current state. The G80 was the first chip to be GPGPU, and as such, it is a solid base on which it is possible to understand GPGPU internal functioning. With time and programming needs, the architecture has evolved and we will introduce major features along with the chip where they first appear.

The GPGPU is quite a different chip compared to a conventional central processor, as used in work stations. It is generally mounted on an internal board, connected to the main memory by a PCI-E connector. Therefore, it generally has its own memory space (except in the case of small GPUs, as in some laptops), which is almost as large as the central memory of the work station. The power of GPGPU resides more in its large number of processing cores, rather than on a few powerful computing units. Therefore, it primarily copes with largely parallel problems.

Even if they are claimed as general, GPGPUs' main purpose is 3D rendering, for which most cards sold on the market are used. The architecture can be seen as mainly oriented towards 3D rendering and marginally towards scientific computing.

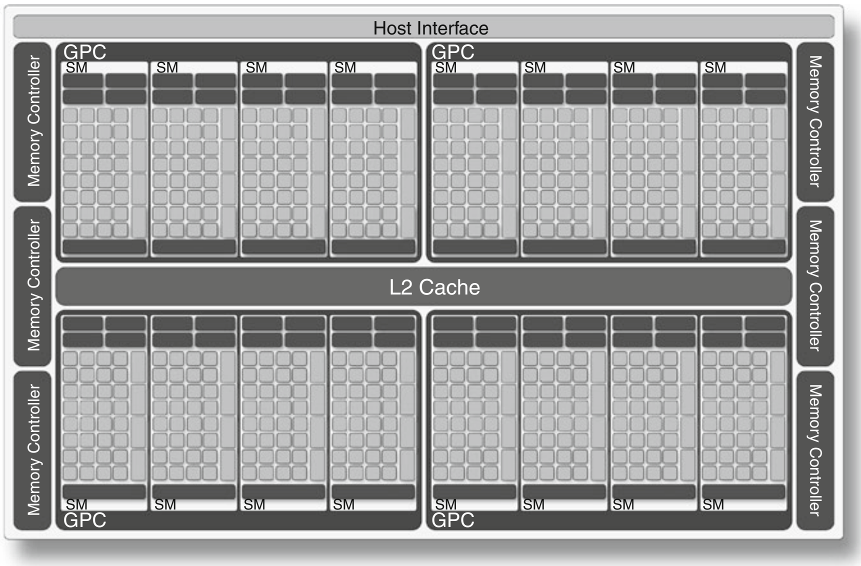


Fig. 1 A schematic view of an NVIDIA GF110 chip

The large number of cores present in these chips requires a strong and constraining structuring, which induces significant changes in the programming paradigm, especially for tasks distribution (Fig. 1).

3.1 Core Organization

The G80 contains 128 scalar cores in a chip, for a quantity of transistors close to a current processor. Increasing the number of cores, compared to an Intel P4, is performed due to the lack of certain complex features that are usually implemented on standard processors, including out-of-order execution, cache memory, and call stack. In addition, the cores are organized into groups of eight, each with two texture units, a texture cache and shared memory. Core structuring changes regularly, therefore the number of groups and scalar cores depends on the general architecture of the processor but also on the model. Entry-level models generally contain a total of fewer groups and fewer cores.

3.1.1 Streaming Processors

Streaming Processors are the basic computing elements of a GPGPU. They correspond to the core and are capable of scalar calculations. They have similarities with

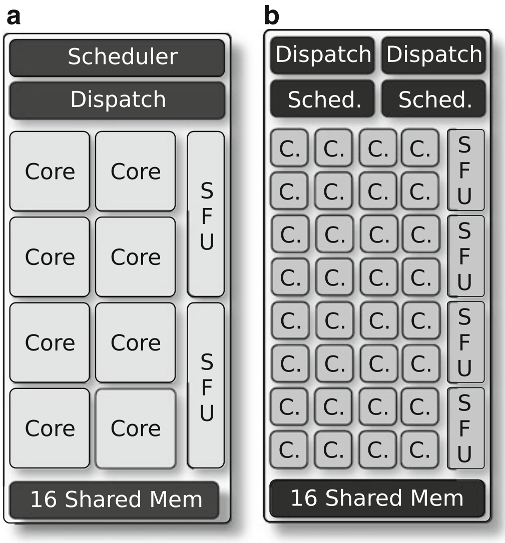


Fig. 2 MP schematic views, including cores, on-chip memories and instruction units. (**a**) The MP of a G80/GT200 chip, (**b**) the MP of a Fermi chip

standard modern CPU cores, as they are able to handle several threads at the same time, also called simultaneous multi-threading (SMT) or Hyper-Threading (on Intel CPU). This allows the core to change its thread whenever it is stalled on a memory operation. But it is also different from a standard CPU core. For example, it does not contain computation registers or instruction units.

The cores have their own frequency called the shader clock, which can be different from the GPGPU clock. For example, on G80, the shader clock was four times the GPGPU frequency, it goes down to two times on the Fermi architecture, and this notion disappears on the Kepler architecture, where the shader is equal to the GPGPU clock.

Finally, the SP can hardly be considered as a real core, mainly because of its lack of instruction units. In order to be compatible with our usual understanding of a computing core, it has to be considered as a part of the multi-processor.

3.1.2 Multi-Processor

The multi-processor (MP) is a group of SPs, with one or several instruction units. The MP structure varies widely depending on the model. Figure 2a presents the first MP structure in G80/GT200 cards where only eight SPs were packed into an MP. The first great changes appeared with the Fermi architecture, as shown in Fig. 2b, where the number of SPs increased to 32. A new level of complication is reached with the Kepler architecture, as the number of cores rises significantly, compared to older architectures (256, 192 single precision and 64 double precision).

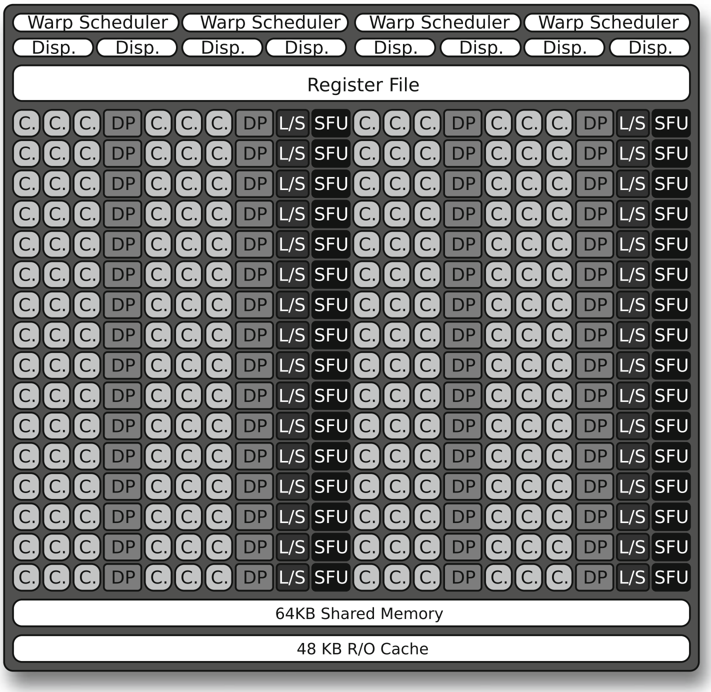


Fig. 3 A schematic view of an NVIDIA GK1xx chip (Kepler)

An MP has a set of registers that are distributed to the cores and to the sets of threads that run on it. On a G80, each MP can handle 768 threads at the same time (maximum 12,288 threads on the whole system). This number increases to 1,536 on the Fermi architecture and culminates in 2,048 on the architecture for Kepler (30,720 loaded threads; Fig. 3).

An important feature of MPs is the integration of several SFUs (Special Function Units). These units are inherited from 3D rendering and are helpful in computing fast approximations of transcendental functions (EXP2, LOG2, SIN, COS). The number of SFUs varies across the different generations of GPGPUs.

Finally, the MPs contain an on-chip memory that will be detailed in Sect. 3.2.2, as well as warp schedulers detailed in Sect. 3.3.

3.1.3 Warp

An MP always executes its threads in an SIMD way. Due to hardware constraints and in order to allow modification of the core structuration among the models, the number of threads executed together stays the same on all models of NVIDIA

GPGPUs. The *warp* is a bundle of threads that are always executed together.

Several strategies exist in executing the warps on MPs. Indeed, the G80 has eight cores and one instruction unit, which leads to a warp being executed, quarter by quarter, until the next instruction comes up. This is directly connected to the speed of the instruction unit, which is four times slower than the core speed.

The GF100 (GTX480, etc.) has 32 cores per MP and two instruction units, so each MP runs two half warps at each time. Then, two warps are performed every two clock cycles and the SPs run at two times the instruction unit clock rate.

On Kepler, 192 single precision cores are packed into an MP as well as eight instruction units. In this case, two instructions can be executed on four different warps. Here, the instructions are executed in one go, on a whole warp, due to the high number of cores in an MP.

3.1.4 Divergence

As seen in Sect. 3.1.2, several cores of the processor are structured around the same instruction unit. Executing different instructions at the same time on these cores is impossible. In order to manage code that still requires this kind of behaviour, such as in Code Snippet 1, the cores have to diverge, i.e. some cores run their common instructions (in this example the “then” part (Fig. 4b)), while the others remain idle, then the first do nothing, while the second group run their instructions (the “else” code (Fig. 4c)).

```
if( threadId/2==0 ){ // then part }
else { // else part }
```

Code Snippet 1: Code producing divergences between four cores

Obviously, these divergences lead to performance loss because some cores are idle for a moment. At high doses, these divergences slow down the execution of the code and should be avoided.

A good example of code that could produce divergences is code using the thread number in a control structure. It is possible to have a number of divergences on such an operation, up to the number of threads running in SIMD (that is, the warp size, see Sect. 3.1.3).

The definition of divergence is based on the warp notion. Indeed, only the threads within a warp are likely to create conflicts. Two threads that do not belong to the same warp therefore cannot create divergences. This means that the minimum degree of parallelism of an NVIDIA GPGPU is 32.

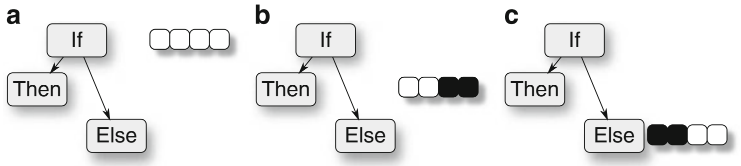


Fig. 4 The evolution of the available amount of shared memory during GPGPU evolution. (**a**) All thread tests the conditions, (**b**) first threads execute the "then part", (**c**) other threads execute the "else part"

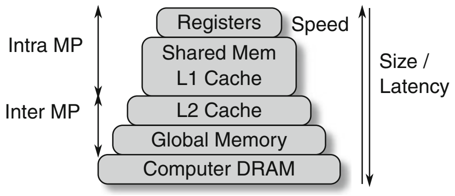


Fig. 5 Schematic view of the memory hierarchy of a Kepler GPGPU

3.2 Memory Hierarchy

As with the core organization, the GPGPU memory hierarchy is very specific to this kind of architecture. This is mainly due to its 3D rendering inheritance. The memory hierarchy is deeper than in standard processors as shown in Fig. 5, and it contains several disjoint memory spaces.

3.2.1 Cache Memory

The cache memory implementation has varied greatly during GPGPU evolution. The G80 has no R/W cache. It only has two R/O caches, one for constant data and the other for textures.

From the Fermi architecture onwards, an L1 cache is implemented of size 16 or 48 K/MP. It has to be noted that due to the large number of threads potentially running on each MP (up to 2,048), the available cache memory per thread is greatly reduced compared to a conventional processor (64 KB for two threads of a core on an Intel Core i7).

From a programming point of view, G80 can use cache memory to store constant data. Programmers can also transform data into textures, in order to use their caches to speedup memory accesses (by using spatial and temporal locality).

The Fermi architecture allows the programmer to use the L1 and L2 caches, in the same way as with a standard processor (i.e. transparently), keeping in mind that the size per thread will be small. The cache system is built on top of the device memory and uses the same memory address space. Furthermore, using the largest setting for L1 cache sizes (48 KB) will reduce the size of the shared memory, as we will see in the next section. L2 cache is common to all the cores of the processors, and its maximum size is 768 KB.

In the Kepler architecture, global variables are not cacheable in L1. At best they are stored in the L2 cache. Here, L1 is reserved for local variables and register backups. L2 cache on Kepler grows to a maximum of 1.5 MB.

Programming with constant and texture caches can be beneficial on G80, but it can prove to be counterproductive on newer models ([3]). Considering the age of this kind of GPU, and the complexity in implementing this technique, this should disappear quickly from GPGPU programming techniques.

3.2.2 Shared Memory

Each MP has a memory called "shared", by which we must understand that it is shared between threads running on the same block. G8x and GT2xx architectures have 16 KB of this shared memory on each MP. Shared memory has to be directly accessed in the code, as it has a different memory space from standard GPU memory. This memory is very fast (within an order of magnitude comparable to registers) and has complex access modes.

In fact, it is organized in different banks, which are served by a bus allowing very fast access, with particular configurations. The general idea is that the access is the fastest if each bank serves one thread. Otherwise, the accesses are serialized, and the access time is multiplied by the number of threads that access the bank. This behaviour is called a *bank conflict*.

Firstly, on G8x and GT2xx, one-to-one accesses are allowed without conflict. This architecture uses 16 banks and the threads are served by half warps (16 threads at a time). The address space is spread over all banks by 32 bits. Finally, the G8x/GT2xx shared memory bus allows values to be read that are broadcasted from a bank to all the cores, i.e. if all threads of a half warp read a variable that is in the same word of 32 bits.

From Fermi onwards, several threads of a (half) warp can access the same bank, without conflict if they access the same 32-bit word. In this case, the word is sent to the requesting threads, which extract the needed part. Conflict persists if two threads try to access two bits that do not fall within the same 32-bit word, but still belong to the same bank.

In Kepler, the number of banks rises to 32, allowing the warps to be served entirely (and no longer by half warps). The same operation as above is extended to words of 64 bits.

The shared memory is an important feature of NVIDIA GPGPUs. It is very fast to access, because it is close to the cores, but it is also difficult to use, as its size is small compared to standard DRAM devices. Furthermore, other complications have appeared during GPGPU evolution. The increase in number of cores and manageable threads has side effects on the use of this memory.

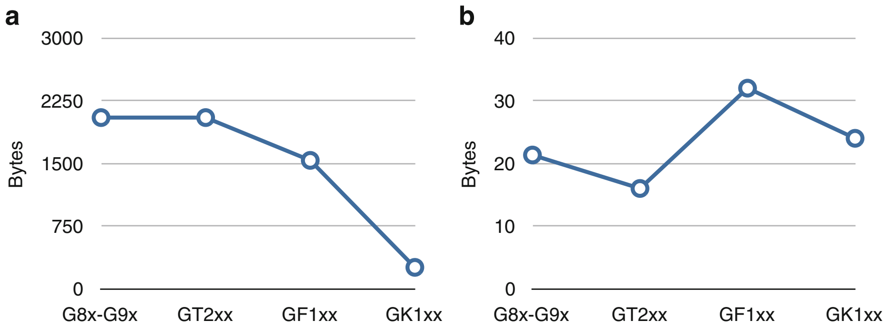


Fig. 6 The evolution of the available amount of shared memory during GPGPU evolution. (**a**) SHM per core, (**b**) SHM per thread

Implementations that use shared memory to store thread-specific data are difficult to maintain. Indeed, the amount of shared memory available for each thread will vary depending on the model, and an adaptation to the processor is necessary to port the algorithm to different processor generations.

Indeed, Fig. 6a, b shows the evolution of the shared memory ratio per core and thread on different important architectures. It has to be noted that the available shared memory per core tends to decrease with time, mainly due to the sharp increase in the number of cores. Available memory per thread also undergoes a contrasting trend, by going down when the number of schedulable threads increases, and finally going up on the Fermi architecture when shared memory size is increased to 48 KB. Finally, the last generation card causes a further decrease of the value, which returns to a value close to what it was originally.

However, shared memory is the privileged place for communication between threads of the same process. It allows for quick exchange of information between the cores, if communications are small sized. The amount of shared memory used by a block can be determined statically at compile time or dynamically at runtime. As we will see in Sect. 3.3, the amount used by a block must be suitable, in order for that block to be placed on an MP.

3.2.3 Registers

As discussed in the section dedicated to software 4, a large number of threads are loaded on a core at a given time. This leads to the use of a large number of registers. To maximize the capabilities of the register allocation, registers are not dedicated to a core in particular but are potentially common to all the cores of an MP and are assigned to a particular thread when the thread execution starts. Registers are installed in the “register file” to be used by all the threads loaded onto an MP.

Table 3 Evolution of register characteristics during time

	G8x-G9x	GT200	GF1xx	GK104	GK110
Registers per thread	128	128	63	63	255
Registers per MP (K)	8	16	32	64	64

It has to be noted that the number of registers in the case of GPGPU is more important than in a conventional processor. Indeed, the lack (or low size) of cache can cause a memory access to be expensive for each register backup. Therefore, a large number of registers are important for the proper functioning of a program on GPGPU, in order to avoid access to the main memory or an L1 cache saturation.

The number of registers that can be used by a thread, and the number of registers implemented into MPs, has varied over generations of cards, as presented in Table 3. In Sect. 4, we shall see how the number of registers can affect the behaviour of a GPGPU program.

3.2.4 Device Memory

The memory device is the widest memory of the card. It is directly accessible by the GPGPU and has a separate memory space, which is common with L1 and L2 caches. The local and global variables are stored in this memory. It is a fast memory, with high latency. However, the memory bus is relatively large and can be used by multiple threads in a single operation if access patterns are correct. This memory is accessed by a transaction system, allowing the loading of several variables per transaction.

Processors with compute capabilities 1.0 and 1.1 (G8x, G9x) allow memory access by multiple cores on the same half warp at the same time, if these accesses fall into a common segment, which should be 64 bytes for 4-byte words, 128 words of 8 bytes and 128 bytes for $2x$ 16-byte words (a half warp here is served by two transactions; thus, a warp is served by four). In addition, the words must be accessed in sequence (the nth thread using the nth word of x bytes). For processors with compute capability 1.2 and 1.3 (GT2xx), the sequential access constraint is released, and the words can be accessed in any order by the threads of a half warp, as long as every word fits into the corresponding segment. If these constraints are not met, 32-byte transactions are performed separately for each thread of the half warp.

For higher compute capability processors, the use of the cache simplifies the transactions, as they are made directly through the cache or used by the threads, allowing more complicated schemes than with previous versions. It is also possible to reduce the number of transactions if the data can be loaded by the first half of the warp and retained until the execution of the second half warp. With previous generations, the two half warps caused data loading, even if one transaction could have brought all of them in one load. Indeed, no storage allows the data to wait until the execution of the second half warp.

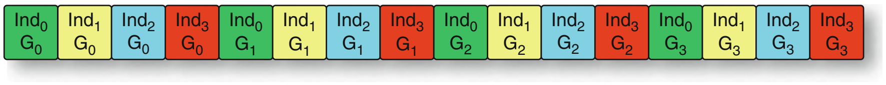


Fig. 7 A gene-wise population organization

This type of well-shaped, sequential access is called coalescent memory access, but the term is mainly used for the first processor models, and it tends to disappear from CUDA literature due to recent cache improvement.

The main memory is usually widely used, due to its impressive size. It may be important to make proper use of it, and in particular of its transaction system, which reduces up to 32× the number of memory accesses. The reorganization of the data can help improve this behaviour. In artificial evolution, the organization of a population of individuals by genes rather than by individuals as in Fig. 7 allows the threads of a warp to use the result of one transaction several times.

While the implementation of cache memory on Fermi architectures allows caching of local variables, the small amount of cache memory should motivate programmers to focus on optimizing the SMT behaviour (Sect. 3.3) and the memory transaction behaviours.

3.2.5 Host Memory

One last memory deserves to be mentioned here. It is of course the standard memory of the host machine. This space is not accessible directly by the GPU, but a large part of the data are loaded from this memory. They should indeed be sent by the program running on the CPU to the GPU memory, in order to be used by the program running on it.

This space is often the largest memory space of a machine and it allows access to other devices of the host machine (including other GPGPU memories). It is therefore essential to access it in order to load the initial data, export results, etc. The exchanges between the CPU and the GPU memory are executed through DMA transfers that are initiated by the CPU program, which indicates the address of the source and destination buffer, these addresses being located in different spaces (GPU vs CPU).

These DMA transfers have significant latency, but a high speed. It is important to reduce the number of transfers as much as possible, even if larger transfers should be made. The flattening of the tree structures can allow a reduction in transfers and facilitates the calculation of addresses on the GPGPU side.

3.3 Simultaneous Multi-threading

The GPGPU has a relatively high memory latency and relatively little cache memory to compensate (especially compared to the number of loaded threads). The optimization of the memory behaviour is based on a multi-level scheduling

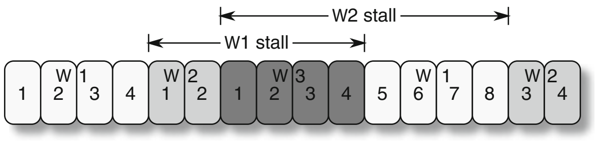


Fig. 8 An example of warp scheduling

mechanism. Indeed, we have seen that SPs are able to execute several threads "at a time", thanks to SMT.

This means that a GPGPU will handle a great deal of threads at a time, and the schedulers will map threads to SPs whenever they are ready to be executed. As threads are always executed with the other threads of their warp, the schedulers handle warps as their scheduling unit. Similar to SMT, a warp is replaced by another whenever it is terminated or it attempts a memory operation that provokes a latency.

The warps are placed on an MP and stay on it until termination, i.e. there is no warp migration between MPs, as the execution environment (registers, shared memory, PC, etc.) is not shared across the whole chip. Figure 8 presents an example of scheduling. The first warp executes four instructions before accessing memory and being stalled; it is replaced by warp 2 for two instructions, until it is also replaced by warp 3. The number of operations that the SPs perform can vary, depending on the architecture.

4 Software

As we have remarked in the hardware part, GPGPU architectures vary greatly with each generation. Moreover, even within a generation, the configuration of the chip (in particular, the number of cores) may change. This can cause complications when porting programs to different target architectures. To overcome this problem, NVIDIA has developed a software framework called CUDA that provides some abstraction of the underlying hardware, which facilitates portability even across different architectures and understanding of the underlying processor.

CUDA in particular defines the logical organization of threads in a program, provides an intermediate language compiler compatible with all the cards that support it and defines a set of computing capacities that the cards can support. A debugger is also provided, as well as a set of libraries, which are useful for scientific computing.

4.1 Logical Thread Organization

The warp defines an interface notion between the hardware and the software. Yet this is not the only existing thread structure. CUDA terminology defines two levels

Table 4 Block and grid constraints

	Compute capability			
	1–1.1	1.2–1.3	2.x	3–3.5
Max block grid dimension	2		3	
Max size of x dimension	2^{16}			$2^{31}-1$
Max size of y, z dimension	2^{16}			
Max thread block dimension	3			
Max size of x, y dimension	512		1,024	
Max size of z dimension	64			
Warp size	32			
Max number of threads per block	512		1,024	
Max number of threads per MP	768	1,024	1,536	2,048
Max number of blocks per MP	8		16	

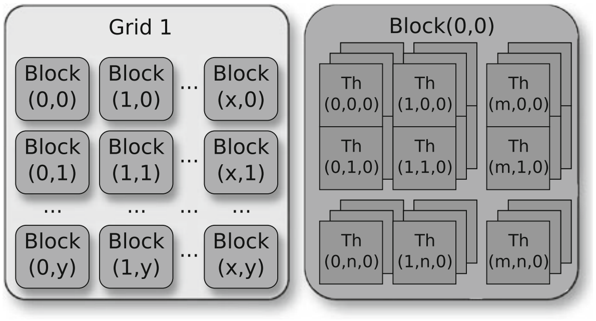


Fig. 9 Thread organization per block and grid

of structures above the threads. Indeed, threads are grouped in a block and the blocks themselves are grouped in a grid, which forms a kernel, i.e. a GPGPU program. A grid is assigned to a GPGPU, and each block is assigned to an MP. In addition, the MP can be assigned to several blocks, if the total number does not exceed the thread hardware scheduling limits. Similarly, multiple grids can be assigned to a GPGPU.

These groups (grid and block) take a multidimensional form (detailed in Table 4), which is used to map threads to data tables used in the calculations. Figure 9 shows the logical structuring of threads per block, then blocks per grid. The links with 3D rendering are obvious here.

Possible configurations vary greatly depending on the compute capability of the GPU. For example, only blocks of arrays are possible for compute capability below 2.x, while cubic blocks become possible above. Yet limitations exist in the matter of size, as a limit on the number of threads per block, even if dimensional limits should allow greater thread sets.

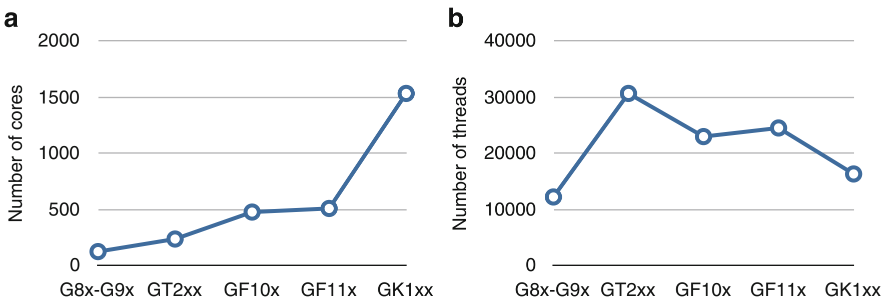


Fig. 10 The evolution of the number of cores and manageable threads per GPGPU over generations. (**a**) Cores per processor, (**b**) manageable threads per processor

4.2 Scheduling Usage

Scheduling greatly optimizes the access time to the memory. If scheduling is done at two levels that are linked to thread organization, in blocks and grids, it has also to be considered in a global manner. Indeed, an algorithm to be parallelized can be divided into a given number of tasks, and it is often difficult to make an arbitrary distribution. For example, in the case of the parallelization of an evolutionary algorithm with the master–slave model (where only the evaluation of the population is parallelized), the number of tasks is fixed (i.e. the number of individuals at maximum). The overall population needs to be considered and distributed into a set of blocks while taking into account the number of threads in each block, which should be set to the most suitable configuration. Here, this problem remains simple, because the evaluation of an individual is independent of other individuals in the population. An individual may be placed in one block or another, without affecting the algorithm.

It is also important to consider the SIMD behaviour of a group of threads in order to create the blocks. The first approximation is that a set of tasks in a block must execute the same instruction at any time. It is possible to refine this statement, because different threads located in different warps can execute different instructions, without loss of performance. Finally, it is also acceptable for threads in a warp to execute different instructions, but divergences will appear and performance will drop.

To ease the scheduling process, it is important for the number of tasks to be as large as possible. The scheduling process is described by the manufacturer as very fast, and no significant overhead should be considered here. We have seen in Fig. 10b that the number of threads loaded onto a processor is high. If the application allows, it should be divided into a large number of independent tasks, and these tasks should avoid synchronization, which is expensive here, as on most parallel architectures.

If the tasks are completely independent and can be grouped freely, it is easy to create blocks and threads that will maximize the use of schedulers. The blocks must have, preferably, a number of threads which is a multiple of 32 (warp size). The grid

Table 5 Number of threads per block or MP

	Compute capability			
	1.0–1.1	1.2–1.3	2.x	3.0–3.5
Max number of threads per block	512		1,024	
Max number of threads per MP	768	1,024	1,536	2,048

must contain a number of blocks that is a multiple of the number of MPs per card. This information can be found in the technical documentation of the card but also dynamically thanks to the function detailed in the Device Management section of the CUDA Runtime API [2] or in the Driver API [1].

It is still necessary to allocate more blocks than there are MPs on the card, as a block cannot contain enough threads to saturate the scheduler. Table 5 summarizes the maximum number of allocatable threads in a block or an MP.

4.3 Scheduling Evolution

A block uses some resources of the MP on which it is placed. Before loading a block, it is necessary for these resources to be available on the target MP. These resources are the number registers and the shared memory needed by each thread of a block.

Since all threads in a block are executed, potentially at the same time, and the SMT mechanism allows the thread data to stay in the registers, a thread uses its resources from when it is loaded, until it is cleaned (its block is terminated). If the number of registers used by threads multiplied by the number of threads in the block exceeds the number of available registers on the MP, the execution of this block is not possible. When placing a second block at the same time as another, the amount of registers left available by the first block should be sufficient to execute the second, with the same mechanism as above.

The shared memory has to be distributed by a very similar process. The only difference comes from the fact that shared memory is assigned to a block and not to a thread. Otherwise, the shared memory consumed by a block is considered in the same way as the number of registers by the allocation mechanism.

The number of registers used by a thread and the amount of shared memory used by a block can generally be seen at compile time. It can also be dynamically calculated by CUDA library functions such as cudaFuncGetAttributes. This method takes into account both software consumption and hardware resources and gives the maxThreadPerBlock value directly. Information about these functions can be found into the section “Execution Control” of the Runtime or Driver CUDA API [1, 2].

It is interesting to design the thread groups of a GPGPU program in order to fulfil the scheduler, in particular to reduce the number of registers and shared memory used by a block, so that two can be placed at the same time on an MP.

Reducing the amount of shared memory in order to increase the number of loadable threads can have a negative effect on performance, especially if the change is accompanied by a massive increase in the device memory usage. Conversely, increasing the use of shared memory can reduce performance if the reduction of the number of schedulable threads reduces the memory performance of the program. It is thus a balance that depends mainly on the current GPGPU program.

4.4 Stream

Due to the high-level management character of blocks, it is possible to place multiple grids on a chip. Indeed, the blocks from different grids having their own resources, as well as blocks of the same grid, can therefore coexist on an MP or on a single chip. From Fermi onwards, it is possible to run 16 grids simultaneously and a transfer host/device in each direction at all times.

This method allows programmers to harness the power of a chip, using several small programs, rather than with a single more power-consuming one. In addition, an algorithm can be cut into several sub-algorithms, allowing better load balancing and concurrent execution of steps.

However, tasks must be independent, which is typically not the case, if an algorithm is cut into several sub-algorithms. This introduces a notion of dependency between tasks, which can be difficult to satisfy.

CUDA introduces the notion of stream, which is used to indicate the existence of a dependency between grids. A stream thus allows a flow organization of calculations. Different flows may run on the same chip at the same time, but not different stages of the same flow.

The standard execution of a program uses standard streams (number 0), but programmers can use other streams to add dependant tasks and to ensure one will be launched after another. The memory transfers (in and out of the GPGPU memory) will take place automatically before and after their respective kernel.

5 Conclusion

GPGPUs have a complex architecture which is very different from that of standard processors. The hardware model is different, and the provided software does not yet entirely cover the gap between these two types of architectures. Indeed, the core architecture is unusual (by its vastness and constraints) but so too is the memory hierarchy, which contains more levels than traditional CPUs. Furthermore, GPGPU is a processor that does not run any operating system and is attached to the host as a stand-alone device.

GPGPU programming has undergone significant changes in recent years. Starting from a diversion of rendering hardwares to become an official solution developed by manufacturers, it has become much easier to use. GPGPU programmers have a modern interface to the hardware, at a much lower level than previously (avoiding an OpenGL or DirectX layer), which is therefore more efficient and generic. However, the programmer is therefore more exposed to changes in the processor architecture. In addition, it is also increasingly linked to a manufacturer, particularly using the CUDA framework.

Shared memory is a good illustration of this principle with the drastic changes that have been made during GPGPU evolution. But these cards still have a cost-to-performance ratio (whether to learn or to buy) that remains interesting. In addition, it is a very common feature which will allow the use of a parallel port on a large number of machines. The development of large GPGPU clusters also motivates a part of the scientific community to port and develop algorithms targeted at these architectures.

References

1. CUDA 5.0 driver API documentation. http://docs.nvidia.com/cuda/cuda-driver-api/index.html
2. CUDA 5.0 runtime API documentation. http://docs.nvidia.com/cuda/cuda-runtime-api/index.html
3. CUDA v5.0 Kepler tuning guide. http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html
4. Fok, K.-L., Wong, T.-T., Wong, M.-L.: Evolutionary computing on consumer graphics hardware. IEEE Intell. Syst. **22**(2), 69–78 (2007)
5. Kedem, G., Ishihara, Y.: Brute force attack on Unix passwords with SIMD computer. In: Proceedings of the 8th Conference on USENIX Security Symposium, vol. 8, p. 8. USENIX Association, Berkeley (1999)
6. Yu, Q., Chen, C., Pan, Z.: Parallel genetic algorithms on programmable graphics hardware. In: Proceedings of Advances in Natural Computation ICNC 2005, Part III, Changsha, 27–29 August 2005. Lecture Notes in Computer Science, vol. 3612, pp. 1051–1059. Springer, Berlin (2005)

Automatic Parallelization of EC on GPGPUs and Clusters of GPGPU Machines with EASEA and EASEA-CLOUD

Pierre Collet, Frédéric Krüger, and Ogier Maitre

Abstract GPGPU cards are very difficult to program efficiently. This chapter explains how the EASEA and EASEA-CLOUD platforms can implement different evolution engines efficiently in a massively parallel way that can also serve as a starting point for more complex projects.

1 Introduction

Many papers show how GPGPU cards can be used to implement very fast massively parallel evolutionary algorithms. However, because these cards are quite complex to program, it is often very difficult to reproduce published results, because it is nearly impossible for authors to describe exactly how they implemented their algorithm. There are simply too many parameters and subtleties to consider.

Because hardware is involved in the performance of algorithms implemented on GPGPUs, details such as how individuals are implemented in memory may result in very different performance. For instance, coalescent memory calls are much faster than random memory accesses when several cores of a multiprocessor make memory calls. Supposing one tries to solve a problem coded with four floating-point values a, b, c, d. Storing the population as an array of individuals will result (in C) in the following memory layout:

$$\underbrace{a_0, b_0, c_0, d_0,}_{i_0}\underbrace{a_1, b_1, c_1, d_1,}_{i_1}\underbrace{a_2, b_2, c_2, d_2,}_{i_2}\underbrace{a_3, b_3, c_3, d_3,}_{i_3} \quad \ldots$$

P. Collet (✉) · F. Krüger · O. Maitre
ICUBE, University of Strasbourg, Illkirch, France
e-mail: Pierre.Collet@unistra.fr; Frederic.Kruger@etu.unistra.fr; Ogier.Maitre@unistra.fr

Now, supposing four SIMD cores running the same code, try to access the first parameter a of each individual they need to evaluate. The data that will be simultaneously read by the four cores will be the values of the variables a_0, a_1, a_2, a_3 which are not contiguous in memory.

However, if one had interleaved the different parameters of the population (as would have been the case if the array of individuals had been stored *à la* FORTRAN, for instance), the memory layout would have been the following:

$$a_0, a_1, a_2, a_3, \dots b_0, b_1, b_2, b_3, \dots c_0, c_1, c_2, c_3, \dots d_0, d_1, d_2, d_3, \dots$$

With this memory layout, the same four cores running the exact same code as before will end up with a much smaller execution time, because a much faster coalescent memory call will be performed.

Because of the difficulty to implement efficient GPGPU code and the difficulty to reproduce obtained results, in Strasbourg we have designed a massively parallel evolutionary platform that can automatically parallelize evolutionary algorithms in a standard predefined way. Whenever our team designs and publishes a new GPGPU algorithm, we take the effort to include the algorithm into the platform, therefore allowing anyone to benefit from this generic implementation to either:

- Implement a massively parallel evolutionary algorithm to solve a problem, using one of the available paradigms, or
- Implement a basic massively parallel evolutionary algorithm that is implementing one of the available paradigms, and use the produced source code as a starting point for a specific development.

This chapter rapidly describes the EASEA language and platform [18] for the reader who would like to benefit from the computing power of GPGPU cards without going through the hassle of directly programming the graphics card.

2 Scope of the EASEA Parallelization Platform

Evolutionary algorithms are quite diverse: by 1965, about ten independent beginnings in Australia, United States and Europe have been traced in David Fogel's excellent *fossil record* [12] on evolutionary computing. However, the main evolutionary trends that survived are:

- Evolutionary Programming, by Lawrence Fogel and later David Fogel on the US west coast [11, 13]
- Evolutionary Strategies, by Rechenberg and Schwefel, best described in [4, 21, 22]
- Genetic Algorithms, by Holland, later popularized by Goldberg on the US East Coast (Michigan) [14, 15]
- Genetic Programming, by Cramer [9] and later developed by John Koza [17, 20]

All these algorithms share a common evolutionary loop that was united into the EASEA language, back in 2000 [8], and its Graphic User Interface GUIDE [7] presented in 2003 that is currently being redesigned as part of the French EASEA-CLOUD project.

The first versions of EASEA (until v0.7) produced C++ (or Java) source code for sequential CPUs. Since 2009, EASEA v1.0 can produce massively parallel code for NVIDIA GPGPU cards. Since 2011, v1.1 can implement an island model over several computers connected over the Internet.

Further, the EASEA platform extends into the EASEA-CLOUD platform to run over computing eco-systems, made of heterogeneous resources that can combine the power of CPU or GPU individual computers or Beowulf clusters, or grids, or clouds of computers using an island model to solve the same problem. The web sites for EASEA and EASEA-CLOUD are respectively http://easea.unistra.fr and http://easea-cloud.unistra.fr. Both platforms can be found under Sourceforge (http://www.sourceforge.net).

3 Standard CPU Algorithm

An EASEA (EAsy Specification of Evolutionary Algorithms) compiler was designed to build a complete evolutionary algorithm around four problem-specific functions and a representation of a potential solution all integrated into a `.ez` source file. Throughout this section, an example will be used that implements an evolutionary algorithm whose aim is to find the minimum of a 100-dimensional Weierstrass benchmark function.

Structure of an Individual

Defining the structure of an individual is done in a section devoted to the declaration of user classes that all `.ez` source files must contain. Declarations use a C-like syntax, allowing one to compose a genome out of integers, floats, doubles, and booleans but also pointers:

```
\User classes :
  GenomeClass {
    float x[100];
  }
\end
```

`GenomeClass` is the reserved name for the class that describes individuals. More than one class can be defined, but at least one must be named `GenomeClass`.

Initialization Function

This function tells how to initialize a new individual. It is a simple loop that assigns a random value to all 100 variables of the genome, within X_MIN and X_MAX, which are macro-definitions declared by the user in a user declaration section.

```
\GenomeClass::initialiser :
  for(int i=0; i<100; i++ ) Genome.x[i] = random(X_MIN,X_MAX);
\end
```

Genome is the reserved name for the individual to be initialized. random is a function of the EASEA library that calls an appropriate random number generator depending on the version of EASEA (typically a Mersenne Twister).

Evaluation Function

A Weierstrass function is used for this example. It is mathematically defined as:

$$W_{b,h}(x) = \sum_{k=1}^{\infty} b^{-kh} \sin(b^{kx}) \quad \text{with } b > 1 \text{ and } 0 < h < 1$$

In this example, we try to minimize the absolute value of this function, hence the evaluation code below for each of the 100 dimensions of the genome. Note that the function is an infinite sum of sines. For practical reasons, only 125 sums are performed.

```
\GenomeClass::evaluator :
  float Res=0, Sum, b=2, h=0.5;

  for (int i=0; i<100; i++) {
    Sum=0;
    for (int k=0; k<125; k++)
      Sum+=pow(b,-(float)k*h)*sin(pow(b,(float)k)*Genome.x[i]);
    Res += (Sum < 0 ? -Sum : Sum);
  }
  return (Res);
\end
```

The evaluation function must return a floating point value that represents the fitness of the individual (note that if one wants to use a roulette-wheel selector as defined in [15], the returned fitness value must be $>= 0$ and the goal of the evolutionary algorithm must be to maximize the fitness function, which is not the case here).

Crossover Function

In this example, a very basic barycentric crossover is presented :

```
\GenomeClass::crossover :
  for (int i=0; i<100; i++) {
    float alpha = random(0.,1.); // barycentric crossover
    child.x[i] = alpha*parent1.x[i] + (1.-alpha)*parent2.x[i];
  }
\end
```

By default, the crossover function already knows about three individuals: parent1, parent2, and child, which are members of the GenomeClass that was defined by the user. By default, child is instantiated with a deep copy of parent1.

Mutation Function

The very simple mutation function below simply adds a random value in $[-.1,.1]$ and makes sure that x remains within bounds.

```
\GenomeClass::mutator : // Must return the number of mutations
  int nNbMut=0;
  for (int i=0; i<100; i++)
    if (tossCoin(.01)){
      nNbMut++;
      Genome.x[i]+=random(-.01,.01);
      if (Genome.x[i] <= X_MIN) Genome.x[i]=X_MIN;
      if (Genome.x[i] >= X_MAX) Genome.x[i]=X_MAX;
    }
  return nNbMut;
\end
```

In this version, the mutation function walks through all dimensions and mutates each gene with a probability .01, meaning that the probability that the individual will be mutated is 1, because there are 100 dimensions to this problem. The number of mutations must be returned because in some versions of EASEA, this is used for statistics.

Set of Parameters

The task of the EASEA compiler is then to wrap a complete evolutionary algorithm around these problem-specific functions and individual structure definition that can virtually implement any standard or nonstandard evolution engine thanks to a large enough set of parameters and a library of functions.

The typical set of parameters is:

- *Number of generations*: the number of generations for the run to complete.
- *Time limit*: the number of seconds before the run is stopped (0 to cancel this functionality).
- *Population size*: the number of individuals in the population.
- *Offspring size*: the number of children to be created per generation.
- *Mutation probability*: the probability to call the mutation function once a child has been created.
- *Crossover probability*: the probability to call the crossover function to create a child (if the crossover function is not called, the child is a clone of the first parent).
- *Evaluator goal*: to specify whether the evolutionary engine should minimize or maximize the fitness of the individuals.
- *Selection operator*: to be chosen among (Deterministic, Random, Tournament, Roulette). “Roulette” can only be specified if the evaluator goal is “maximize”, and Tournament needs a parameter to specify selection pressure. An n-ary tournament will be implemented for integer values of n greater than 2, and a stochastic tournament will be implemented for real values between .5 and 1.0.

(In a stochastic tournament, the best of two individuals is returned with a probability between 0.5 (random selection) and 1.0 (binary tournament).)

- *Surviving parents*: number of parents to participate in the selection of the individuals that will constitute the next generation.
- *Surviving offspring*: number of children to participate in the selection of the individuals that will constitute the next generation.
- *Reduce parents operator*: how the surviving parents are selected.
- *Reduce offspring operator*: how the surviving children are selected.
- *Final reduce operator*: how the next generation is selected among the surviving parents and offspring.
- *Elitism*: strong (meaning that the elite is selected among the parents) or weak (meaning that the elite is selected among the surviving parents + surviving offspring).
- *Elite*: Elite size (number of best individuals to make it to the next generation).

This rather complete set of parameters allows one to specify virtually any evolution engine, from a generational GA (offspring size equal to population size, 0 surviving parents, 100 % surviving offspring, deterministic selectors) to a steady state GA (offspring size equal to 1, weak elitism) to an evolutionary strategy comma (offspring size greater than population size, 0-surviving parents) or an evolutionary strategy plus (surviving parents + surviving offspring > population size).

Note that strong elitism is used with generational algorithms (generational GA or ES-comma) where the elite is chosen among the parents, while non-generational algorithms such as ES-plus can choose their elite among the offspring or parents (weak elitism).

Then, when the `easea` compiler is invoked on an `.ez` file containing the above-mentioned functions and parameters, a man-made template that contains a complete evolutionary algorithm is specialized with the provided user functions and parameters.

The result is a human-readable indented and commented C++ source file that implements a complete algorithm that evolves solutions to the problem that was specified in the evaluation function, using genetic operators specified by the user.

4 Partial and Full Parallelization on One Machine

Evolutionary algorithms are intrinsically parallel, so in theory, all the different steps are fully parallelizable, meaning that Amdahl's law [3] should not hinder speedup (cf. chapter "Why GPGPUs for Evolutionary Computation?").

However, this is only true for a non-elitist generational GA, where the offspring population replaces the parent population: if one wants to implement an elitist generational GA, then at some point, it will be necessary to find out who was the best parent, to copy it into the next generation.

Unfortunately, finding which individual is the best is not intrinsically parallel, because all different cores must then communicate at some point. This implies some synchronization between the cores as well as exchanging information that will degrade speedup.

Worse: if one wants to implement an evolutionary strategy plus (ES+) with a population of 10,240 parents + 10,240 offspring on a 1,024-core GPGPU, it will be necessary to select 10,240 individuals out of 20,480 individuals to create the next generation. Supposing each of the 1,024 cores was asked to select 10 good individuals out of the 20,480-individual temporary population made of parents + offspring to constitute the next generation of 10,240 individuals, it is extremely probable that some good individuals will be chosen many times. *This means that the new generation of 10,240 individuals will contain many clones which will lead to premature convergence.*

One solution is to implement an intrinsically parallel selector without replacement, as was done with the DISPAR tournament [19], but this selector is not absolutely identical to a tournament selection, which means that the massively parallel algorithm will not be strictly equivalent to its sequential counterpart.

Another solution is to only parallelize the evaluation function, which makes sense if its computation is intensive enough to dwarf the time devoted to execute the evolution engine. Using this solution means that both the sequential and parallel algorithms will be strictly identical.

Below is a test done on the Weierstrass function shown above, on a PC with an Intel Core I7 990X running at 3.46 GHz for an advertised computing power of 107 GFlops with an NVIDIA GTX680 with 1,536 cores running at 1 GHz for an advertised computing power of 3.09 TFlops.

For the test below, the exact code above was inserted into an `.ez` file as well as the parameters below that implement a generational GA.

```
\Default run parameters :
  Number of generations : 10000
  Time limit: 0              // in seconds, 0 to deselect
  Population size : 16384
  Offspring size :  16384
  Mutation probability : 1
  Crossover probability : 1
  Evaluator goal : minimize
  Selection operator: tournament 7
  Surviving parents: 0
  Surviving offspring: 100%
  Reduce parents operator: deterministic
  Reduce offspring operator: deterministic
  Final reduce operator: tournament 2

  Elitism: Strong
  Elite: 0
  ...
\end
```

Compiling the file (under LINUX) with EASEA for CPU (default mode) is done with the following lines:

```
$ easea weierstrass.ez
$ make
...
$
```

Then, execution gives:

```
$ ./weierstrass --nbGen=10
...
Population initialisation (Generation 0)...
GEN     TIME      EVAL    BEST      AVG       STDDEV    WORST
0        85.83     16384  1.11e+02  1.25e+02  3.50e+00  1.39e+02
1       171.44     32768  1.01e+02  1.16e+02  3.35e+00  1.30e+02
2       256.83     49152  9.72e+01  1.09e+02  3.26e+00  1.22e+02
3       342.04     65536  8.81e+01  1.03e+02  3.25e+00  1.16e+02
4       427.01     81920  8.32e+01  9.53e+01  3.15e+00  1.07e+02
5       511.76     98304  7.53e+01  8.73e+01  2.95e+00  9.83e+01
6       596.32    114688  6.77e+01  7.95e+01  2.78e+00  9.17e+01
7       680.66    131072  5.98e+01  7.18e+01  2.57e+00  8.24e+01
8       764.81    147456  5.42e+01  6.46e+01  2.38e+00  7.31e+01
9       848.77    163840  4.72e+01  5.78e+01  2.15e+00  6.58e+01
Current generation 10 Generational limit : 10
Stopping criterion reached
```

Note that the command line parameter `--nbGen=10` overrides the default parameter that was in the `.ez` file that was saying:

```
  Number of generations : 10000
```

One can see that each generation that evaluates 16,384 individuals takes around 85 s to compute, meaning that one evaluation takes less than 5.24×10^{-3} s, because in each generation the time taken by the evolution engine is also included.

After 10 generations and 163,840 evaluations (remember that the algorithm presented above is generational), the value found for the best individual is 47.2.

Now, using the very same `weierstrass.ez` source file, EASEA can wrap around its contents a complete algorithm that performs a massively parallel evaluation on the NVIDIA GPGPU card, if its compiler is invoked with the `-cuda` command line option. Execution time is then different:

```
$ easea weierstrass.ez -cuda
$ make
...
$ ./weierstrass --timeLimit=849
...
Population initialisation (Generation 0)...
card (0) GeForce GTX 680, 16384 individuals: t=1024 b: 16
GEN     TIME      EVAL    BEST      AVG       STDDEV    WORST
0       0.16       16384  1.09e+02  1.25e+02  3.49e+00  1.39e+02
1       0.41       32768  1.04e+02  1.16e+02  3.32e+00  1.28e+02
2       0.66       49152  9.68e+01  1.10e+02  3.24e+00  1.23e+02
3       0.90       65536  9.02e+01  1.03e+02  3.23e+00  1.15e+02
```

```
4       1.14     81920  8.38e+01  9.55e+01  3.13e+00  1.07e+02
5       1.38     98304  7.60e+01  8.75e+01  2.97e+00  1.01e+02
6       1.62    114688  6.92e+01  7.97e+01  2.75e+00  9.12e+01
7       1.86    131072  6.23e+01  7.20e+01  2.56e+00  8.20e+01
8       2.11    147456  5.59e+01  6.48e+01  2.36e+00  7.45e+01
9       2.35    163840  4.90e+01  5.80e+01  2.14e+00  6.59e+01
...
```

On this quite fast evaluation function (less than .005 s on a CPU), it now only takes 2.35 s to perform the same ten generations, meaning that the obtained speedup is around 360×.

Results are different because the algorithm is stochastic: random choices are made when selecting parents, so results are different between executions.

Compiling with the `-cuda` option means that the EASEA platform compiled the evaluation function with the `nvcc` NVIDIA C Compiler. Then, whenever an evaluation phase occurs, the complete population is collected in one chunk of memory, then transferred in one go onto the GPGPU card and evaluated in parallel on the cores of the card.

All this was done thanks to the `-cuda` command line option of the EASEA compiler. The fact that the `weierstrass.ez` file is identical guarantees that the parallel algorithm is identical to the sequential one.

Because the CPU version evolved the ten generations in 848.77 s, we ran the evolutionary algorithm for 849 s (`--timeLimit=849` parameter on the command line) to find out what result would have been found in the same amount of time as the sequential CPU algorithm:

```
3621  845.99  59342848  6.19e-01  8.58e-01  2.38e-01  2.47e+00
3622  846.22  59359232  6.19e-01  8.58e-01  2.35e-01  2.56e+00
3623  846.46  59375616  6.18e-01  8.61e-01  2.44e-01  2.56e+00
3624  846.69  59392000  6.18e-01  8.59e-01  2.40e-01  2.23e+00
3625  846.92  59408384  6.17e-01  8.56e-01  2.37e-01  2.19e+00
3626  847.15  59424768  6.16e-01  8.58e-01  2.38e-01  2.32e+00
3627  847.39  59441152  6.16e-01  8.56e-01  2.36e-01  2.22e+00
3628  847.62  59457536  6.16e-01  8.59e-01  2.39e-01  2.42e+00
3629  847.85  59473920  6.16e-01  8.55e-01  2.36e-01  2.09e+00
3630  848.90  59490304  6.16e-01  8.53e-01  2.38e-01  2.16e+00
Time Over
Time Limit was 849 seconds
```

Because evolutionary computing relies on evaluations to evolve new solutions through genetic operators, Fig. 1 shows that having access to more computing power will allow better solutions to be found in the same amount of time.

One can see that 3,630 generations were evolved thanks to the GPGPU parallelization while only 10 generations were obtained for the sequential algorithm on one of the cores of the CPU. This confirms the 360× speedup that was computed on the difference of execution time between CPU and GPGPU versions, meaning that 10 s on the GPU is equivalent to 1 h execution on the CPU.

Obtaining a value of .616 would therefore have taken 85 h (3.5 days) on the CPU, and 1 day computation on the GPU is equivalent to 1 year (360 days) on the CPU!

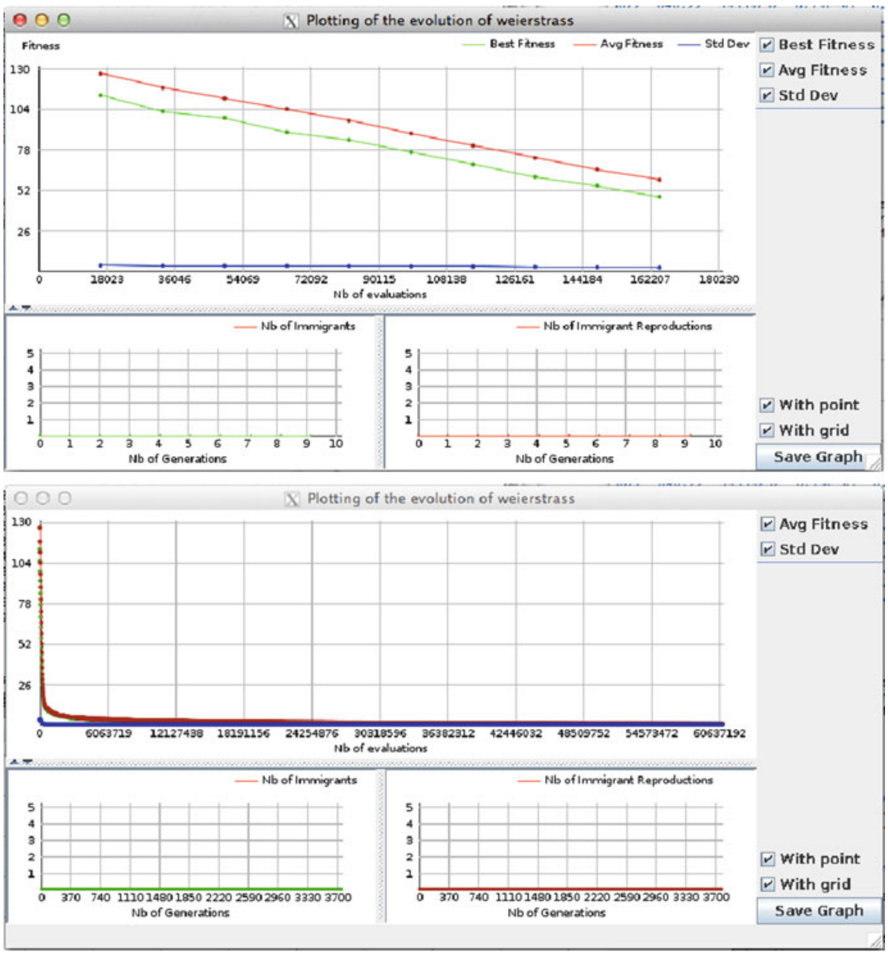


Fig. 1 The EASEA platform provides a graphic display that shows (among other information) the fitness of the best individual, the average of the population and the standard deviation. The *top* and *bottom* figures show the obtained results for the *same* algorithm on CPU and GPGPU, run for the same amount of time (849 s)

The speedup offered by a GPGPU card may allow one to tackle problems that are currently beyond the reach of a sequential algorithm.

Because the parallelization is performed on the evaluation function only (and supposing that parallelization was absolutely perfect over an infinite number of cores), it is important to stress that the obtained speedup depends on how computer intensive is the evaluation function and that for a particular problem, the maximum speedup can only be obtained with population sizes about an order of magnitude greater than the number of cores of the used GPGPU card.

On this example, obtaining a 360× speedup means that the sequential part of the algorithm is 1/360th of the complete algorithm, suggesting that the parallel part of the algorithm represented 99.9972 % of the complete algorithm.

If the EASEA automatic evaluation parallelization is tested on a function that represents less than 99.9972 % (i.e. is shorter to evaluate than .005 s on the CPU), Amdahl's law will apply and the speedup will not be as good as 360×. This is the price to pay for using a strictly identical algorithm to the sequential one.

On a much simpler function, for instance, the obtained speedup can be <1 because of the overhead induced by transferring the evaluation onto the GPGPU card. For instance, the speedup obtained on the following Rosenbrock function:

$$R(x) = \sum_{i=1}^{\dim}[(1-x_i)^2 + 100(x_{x+1} - x_i^2)^2] \tag{1}$$

shown in Fig. 2 top can be as low as 0.65× on an NVIDIA GTX480 card vs. one core of an Intel Core i7 950 CPU running at 3.07 GHz, for 80 dimensions and 10,000 individuals.

In this case, in order to fully exploit the GPGPU card, it is necessary to use a fully parallel implementation on a GPU, using either a parallel simulation of a tournament for the reduction phase (DISPAR tournament [19] in Fig. 2 bottom left) or a generational algorithm (in Fig. 2 bottom right) on which accelerations up to 250× can be obtained even on this very lightweight evaluation function, on a GTX480 card `vs.` one core of a good CPU (Intel Core I7 950) of the same generation.

In order to better understand what happens when parallelizing EAs on GPGPU cards, Fig. 3 shows the time distribution between the different phases.

The top line (ezCPU) shows (from left to right):

1. The initialization phase (nearly invisible)
2. The evaluation phase
3. The parents selection phase (tournament in this case)
4. The variation phase (creation of children using crossover and mutation) and
5. The reduction phase (reducing the parents + children population to create the next generation)

on the same Weierstrass function with parameters tuned so that the disproportion between the evaluation phase and all the other phases is not too accented.

The line below (ezGPU) shows what happens when the same program is compiled by EASEA using the `-cuda` option. The evaluation phase nearly disappears as the evaluation of the 4,800 individuals is parallelized over the 480 cores of the NVIDIA GTX480. What remains is the sequential part of the algorithm, showing Amdahl's law at work.

The only way to obtain a good speedup in this configuration is to parallelize all the phases of the algorithm, which results in the two lines below (gpuDispar and gpuGen), in which the distribution of the different phases is indiscernible.

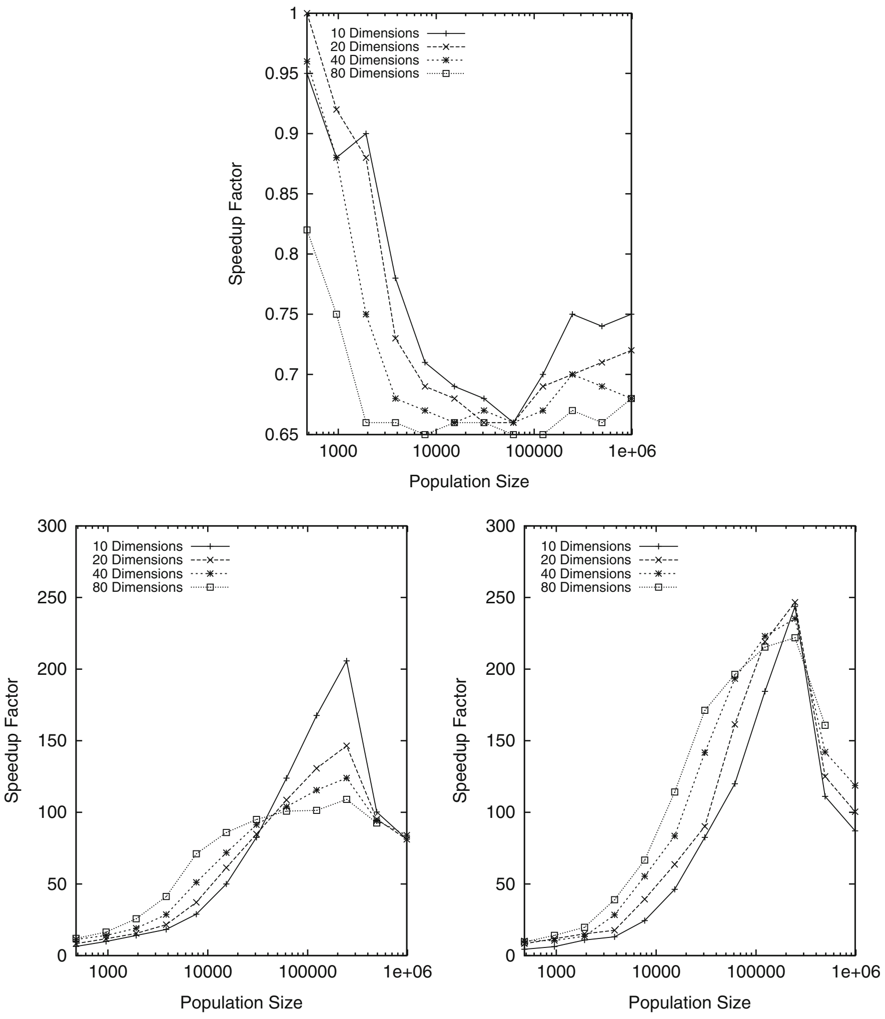


Fig. 2 The *top* figure shows the obtained speedup with a standard EASEA parallelization on the lightweight Rosenbrock function on an NVIDIA GTX480 card for 10–80 dimensions. The *bottom left* figure is obtained with a fully parallel EA using a DISPAR tournament and the *bottom right* figure with a generational fully parallel GA

In order to see something, Fig. 4 shows the same lines as a percentage of the total time. As above, the first (ezCPU) serves as a reference before parallelization. The second line shows the much smaller proportion of the evaluation phase.

The third and fourth lines show completely parallelized algorithms, so if all phases are equally parallelized, the proportions should be similar to those of the first line.

On the third line, the DISPAR tournament used to reduce the population takes a bit more time than the standard tournament, and the initialization phase now

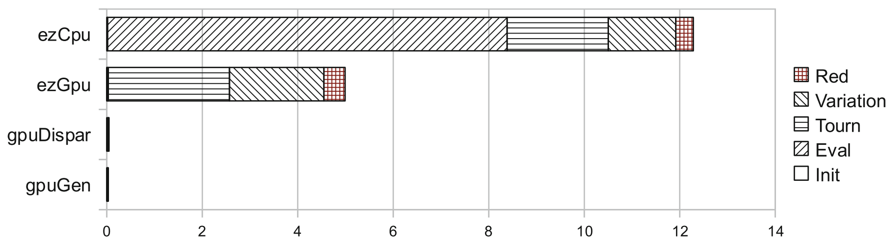


Fig. 3 Time distribution on a GTX480 NVidia card and i7 950 CPU for Weierstrass function with 4,800 individuals, 10 iterations and 50 generations (in seconds), over an average of 20 runs

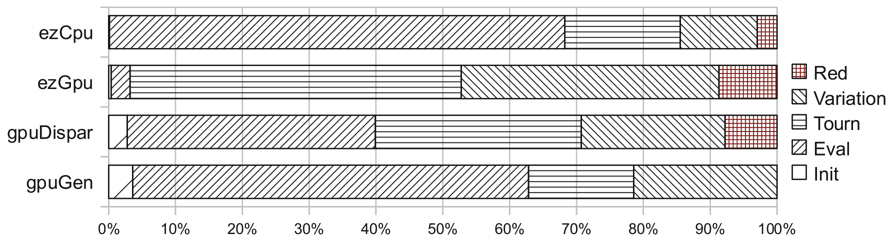


Fig. 4 Same as Fig. 3, but as a percentage of execution time

appears. On the fourth line, the generational replacement uses no time, and the proportions are similar to those of the CPU algorithm. An average was taken over 20 runs because on such small execution times for the parallel version, even small variations will change proportions.

Full parallelization of an evolutionary algorithm using a DISPAR or generational reduction phase will soon be available under EASEA.

5 Island Parallelization and Emergent Behaviour

As was said in the chapter "Why GPGPUs for Evolutionary Computation?", driving through a desert sea of sand dunes can be very tricky, as it is very easy for a vehicle to get stranded in soft sand. In terms of search algorithms, one can say that an evolutionary algorithm can get stranded in a multimodal search space if its population has prematurely converged to a local optimum.

Unfortunately, the task of search algorithms is even more difficult than that of crossing a desert of soft sand dunes because their task is to explore the pits to find the deepest ones. While doing so, the chances that algorithms get stranded (read "premature convergence") are much larger than if they could drive around the ditches.

However, desert drivers found that a complex system was much more efficient to cross the desert. Complex systems can be seen as a group of autonomous entities in

Fig. 5 4WD vehicles implementing an emergent complex system to cross the desert

interaction that implement an emergent behaviour. In Aristotle's terms, the whole is more than the sum of the parts.

Rather than attempting to cross the desert alone, 4WD drivers use several vehicles (cf. Fig. 5). If there were no interaction between the vehicles, the result would be the same as if they used the same vehicle n times: each vehicle would eventually get stranded with the same probability as the other vehicles.

What turns this group of vehicles into a complex system with emergent behaviour is that they carry tow cables and winches that allow them to interact. Indeed, if a vehicle is stranded into a pit, another one can throw it a cable and pull it out of the pit with its winch. The vehicle is not stranded anymore and the group can resume their progression across the desert.

In this case, the emergent behaviour is that where n vehicles will get stranded as easily as one vehicle, n vehicles with tow cables and winches will be able to cross the desert. The whole is more than the sum of the parts.

Evolutionary algorithms already exploit a similar behaviour because they are population-based algorithms. Where one could use n independent simulated annealing algorithms, evolutionary algorithms are more efficient because they use n individuals in interaction (through crossover) to help themselves out of local optima and preserve diversity.

However, even with a large population, algorithms will eventually get stranded as their population will converge into a sweet spot. One can then implement a higher-level kind of parallelism, by getting several evolutionary algorithms to cooperate on the same search space, thereby implementing a multi-level complex system: an island-model evolutionary algorithm [1].

If several independent population-based algorithms (let us call them islands) explore the same search space, they can act like several vehicles exploring an erg and help one another out of local optima in order to find even better areas.

If they get stranded (cf. Fig. 6 left), rather than using a tow cable, an EA island can send its best individual to another island (cf. Fig. 6 right). The individual is not physically sent to the location where the other island resides. The individual is made

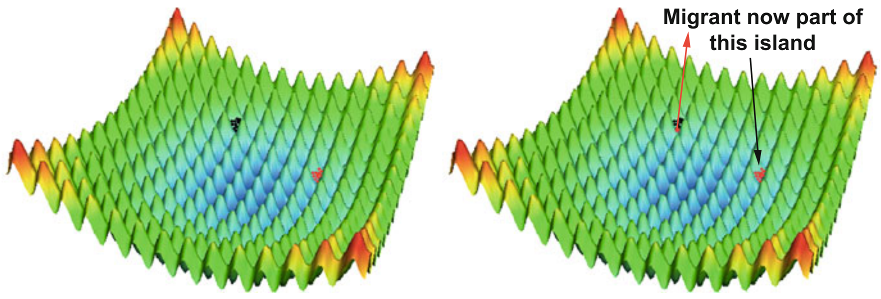


Fig. 6 On this Rastrigin function, two islands have converged towards local optima (*left*). Then, an individual is sent from the left island to the right island. Note that the individual does not physically move: it stays where it is but it is now part of the population of the right island

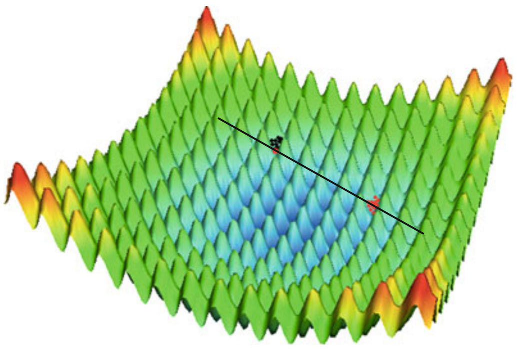


Fig. 7 If the migrant is better than the local individuals of the right island, a barycentric crossover (such as BLX-α with $\alpha = .5$) will distribute offspring across the line. The island, which had prematurely converged, will be pulled out of its local optimum by the immigrant

part of the population of the other island, but it stays topologically where it was in the search space.

If the incoming individual (immigrant) is worse than the individuals in the stranded island, it will get discarded and will not be part of the next generation. If however, it fares well among the population of the island where it was sent or if it is better than the best individuals of the island, this immigrant will implement a tow cable between the two islands:

- It will survive across the generations.
- Other individuals will want to mate with it to create new offspring.

If a barycentric-type crossover is used (a BLX-α [10] with $\alpha = .5$ as in Fig. 7 for instance), some created offspring will keep appearing on a line joining the two islands until a better solution than the one implemented by the migrant is found.

If n stranded islands share the same search space and exchange individuals, children will spawn over $n(n-1)/2$ lines, therefore improving the chances to find better places in the search space, as individuals situated on the lines may also want to create children.

However, individuals should not be exchanged among islands too often, or the multi-island algorithm will become panmictic (with a global population) across the different islands. Ideally, local exploitation should take place in between migrations so that new good spots are well explored before the island is pulled out of its new local optimum by another good migrant.

Typically, search algorithms must stay on the ideal EvE (Exploration vs. Exploitation) compromise. In evolutionary algorithms, this is done by fighting against premature convergence while allowing the algorithm to explore enough local optima.

In an island model, one can implement fast-converging islands (something easy to do because evolutionary algorithms can be made quite convergent if strong selection pressure operators are chosen) that will be periodically pulled out of their local optima by good incoming immigrants.

Needing islands to find the bottom of local optima before they are pulled out of them means that infrequent communication is actually an advantage over frequent communication! This is a very important point, as communication is usually what prevents parallel machines from yielding linear speedup with the number of machines: usually, 10 machines will not go 10× faster than one machine because of the needed synchronization and communication time between machines. Super-fast communication networks between machines are one of the primary costs of supercomputers! Being able to get computers to cooperate over a loosely coupled asynchronous low-bandwidth network means that it is possible to create supercomputers out of standard machines that are cooperating together over the Internet, for instance.

In an island-model parallelization scheme, communication (migration) will actually benefit from taking place asynchronously and in a loosely coupled manner!

A lot of research has already been done on island models. Theoretical studies on the number and the size of populations have been done by Whitley et al. [24]. Other studies show the influence of different island-model parameters such as migration rate and connectivity [6]. Alba and Troya studied asynchronism [2]; Branke et al. studied the influence of heterogeneous networks on island models [5].

Some research has been done on parallelizing evolutionary computation using an island model on a GPGPU card, but this research was limited to a single machine equipped with a GPGPU card that hosted all the islands [23].

Some results obtained with the EASEA island model have been quoted in the Supporting Online Material of a Science paper [16] whose results were obtained with the help of the work described in the chapter "Using Large-Scale Parallel Systems for Complex Crystallographic Problems in Materials Science".

5.1 EASEA Island Model

The default island model implemented by the EASEA language is very basic but very versatile at the same time. It allows the algorithm to periodically send (and receive) one or several individuals to (from) one of several IP addresses and ports listed in a file.

The configuration of the EASEA island model relies on two main parameters of the `.ez` file:

- The name of the file containing the IP addresses and ports (`ip.txt` by default)
- The migration probability

Here is a basic `ip.txt` file where four islands are implemented on the same local machine and four other islands on a distant machine:

```
$ cat ip.txt
127.0.0.1:2929
127.0.0.1:2930
127.0.0.1:2931
127.0.0.1:2932
130.79.92.66:2929
130.79.92.66:2930
130.79.92.66:2931
130.79.92.66:2932
$
```

This allows one to implement several islands on a single machine (for multi-core machines, with different ports on a single IP number) or on different machines (with different IP numbers).

All the islands can use an identical (or a different) `ip.txt` file as when an individual is sent the island makes sure it does not send the individual to itself.

Individuals are sent between islands using a connectionless UDP/IP protocol (this can be seen as individuals sent by mail between the different islands). No connexion means that the process can be asynchronous and therefore quite robust: an island can stop working (due to a hardware problem for instance) and then start again later on. It will receive good individuals from other machines that did not crash and will rapidly find good spots again without disturbing the other islands.

New islands can join the search at any moment.

Sent individuals may get lost, but then the whole EC island paradigm is stochastic, so losing an individual once in a while can be considered as being part of the algorithm.

No time is lost in island synchronization or acknowledgement that a sent individual has effectively been received.

As EASEA can produce code for machines running with or without GPU, running under Linux, Windows or Mac Os X, the EASEA island model can run on any number of heterogeneous machines connected via the Internet worldwide to work on solving the same problem, provided each IP address is present in the IP file of each island and provided they exchange individuals sharing the same structure.

With the EASEA-CLOUD project, this will be extended to Grid and Cloud computing, for a full massively parallel evolutionary computation eco-system.

Algorithms may be different depending on machines provided that they can share the same individuals: slower machines can run exploratory algorithms, while faster ones can concentrate on exploitation of good spots.

5.2 Implementation

Before the evolution begins on an island, the file containing the IP+port addresses of the other islands is parsed, creating a list of clients. The IP+port address of an island can appear several times. Because one IP+port number is picked at random among the number of IP+port entries, replicating an identical IP+port number ten times will give it ten times more chances to be selected than the other islands, therefore allowing one to implement weighted edges between islands.

Then, communication need not be symmetric, as an island can know the IP+port number of another one, but the second may not have the IP+port number of the first. In a heterogeneous environment (machines with GPUs and machines without GPUs, for instance), this will allow one to implement unidirectional flows of individuals between slow exploratory machines that find good spots and fast exploitation machines that concentrate on finding local minima. However, the fast machines may not send back individuals to the slow machines so as to prevent premature convergence in the cluster of slow machines.

Then, at every generation, a migration function is called with a probability p set by the user. A destination island is chosen randomly among the list of clients. Once a destination is chosen, n individuals are selected among the population using a user-defined selector. The selected individuals are then serialized and sent to the destination island.

On every island, at the beginning of the run, a thread is launched that is in charge of receiving incoming individuals. Upon arrival of a newcomer, a user-defined selector decides who in the population will be replaced. The integration process for newcomers is performed at every new generation.

The described implementation of an island model is very simple, robust and versatile, as it allows one to easily design complex topologies. For instance, it would be really easy to create a ring topology by just giving each island the address of the following island. One could also create a square or hexagonal toroidal grid or virtually any other topology. Then by changing the migration probability, the communication rate can easily be increased or decreased: for instance, the centre island in a star-shaped topology may need more frequent communications with other islands, or one could imagine modifying the migration rate depending on some strategy (increase or decrease migration with the number of generations).

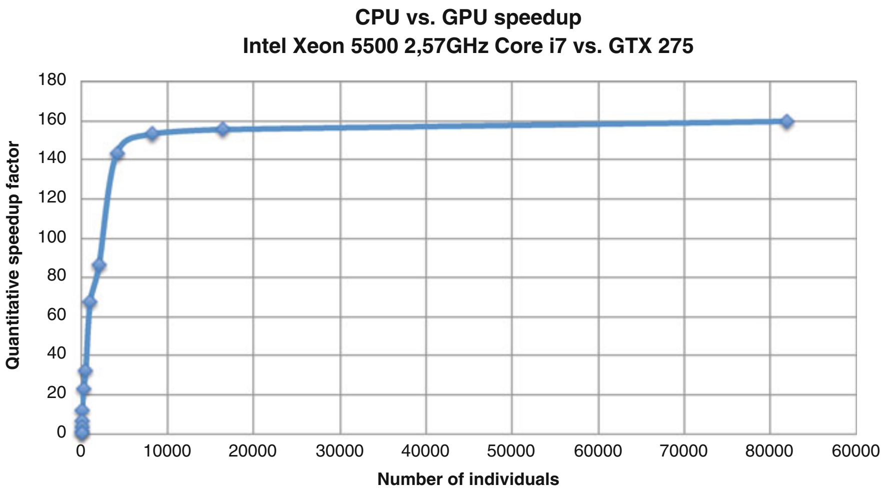


Fig. 8 GPU vs. CPU speedup on a 240-core GTX275 NVIDIA card vs. an Intel XEON 550 CPU of the same generation depending on population size. The Weierstrass benchmark is set up over 1,000 dimensions with 120 iterations and using a Hölder parameter value of 0.35. Greater population sizes give better speedups

6 Experiments

In this section, the EASEA GPU island model is tested with an old cluster (a classroom) of 20 Intel Xeon 5500 Core i7 PCs, each containing a 240-core NVIDIA GTX275 GPU card, for a total computing power of around 20 TFlops.

6.1 Quantitative GPU vs. CPU Speedup

First of all, before the island model is tested, it must be recalled that evaluation can be parallelized on each machine's GPU card, so Fig. 8 presents the obtained speedup by comparing the standard sequential implementation of the Weierstrass problem by the EASEA compiler on one core of an Intel Xeon 5500 2.57 GHz Core i7 CPU with 8 GB Ram vs. the same code compiled with the `-cuda` command line parameter, using an NVIDIA GTX275 card of the same generation with 240 cores.

Different population sizes were used ranging from 8 to 81,920, showing that the obtained speedup reaches a plateau at around 160× for a population size larger than 8,192, while a still-reasonable speedup can be obtained with 4,096 individuals only.

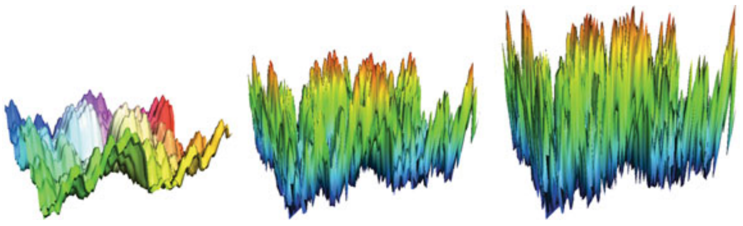


Fig. 9 Weierstrass function for Hölder coefficients 0.9, 0.5 and 0.35 from left to right

6.2 Qualitative Speedup for the Island Model

As a benchmark function, we used the same Weierstrass function, whose irregularity can be tuned thanks to its Hölder coefficient h.

Because we are interested in results in this section, we needed to tune the difficulty of the benchmark in order to show the *qualitative* speedup that can be obtained using an island model.

Usually, a Hölder coefficient of 0.5 is used but it turned out that this value created a too-simple function for the 20 TFlop cluster, so irregularity was increased by using a 0.35 Hölder coefficient (irregularity increases as h decreases) and 120 iterations (cf. Fig. 9).

Then two dimensions only was too small a search space for a 20 TFlops cluster, so a 1,000-dimensional problem was used to create a tough enough problem on which conclusions could be drawn.

Knowing that quantitative computing power is exactly linear with the number of machines, it is interesting to have a look at the obtained *quantitative* speedup. How much faster will it be to find the same result on 5, 10 or 20 machines than on one single machine ?

Looking at Fig. 8, it was decided that 4,096 was the best compromise between population size and obtained GPU speedup.

In order to only examine the influence of the island model, all experiments are done using not only identical parameters but also constant population size because population size will play a role in exploration and premature convergence. Indeed, if premature convergence is what an island model is intended to prevent, it would be unfair to compare:

- A single population of 4,096 individuals on one machine for 2,000 generations, to
- 20 populations of 4,096 individuals on 20 machines for 100 generations,

because evolving 81,920 individuals for 100 generations would not converge as fast as 4,096 individuals over 2,000 generations.

So in order to keep things fair, the setup is the following: one population of 81,920 individuals on one machine will be compared to:

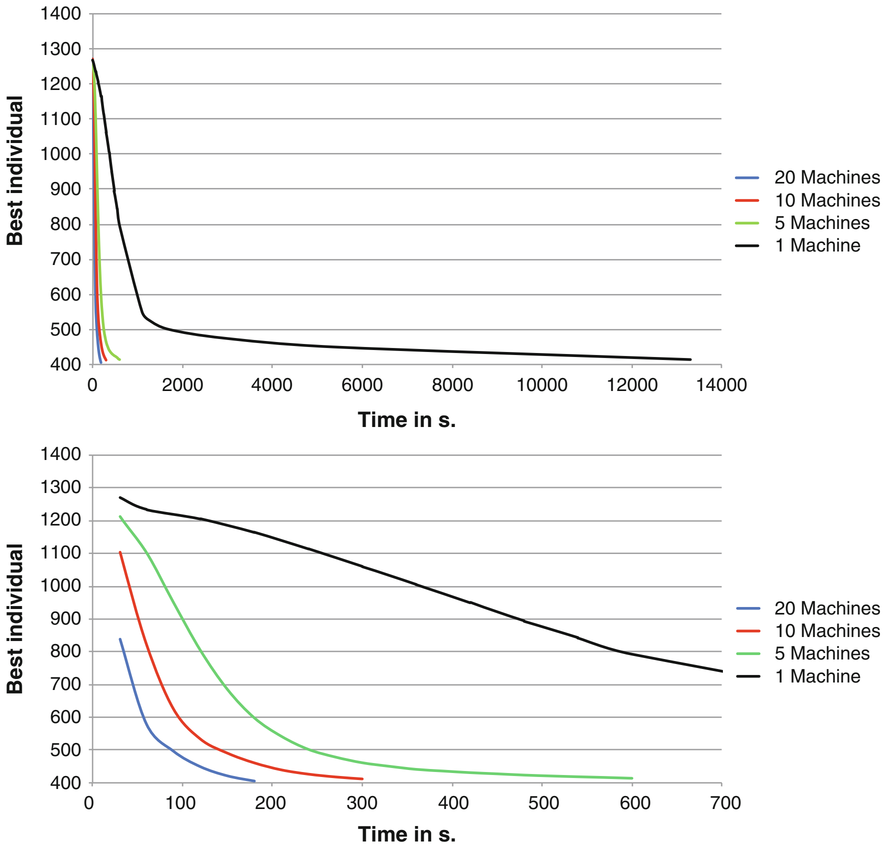


Fig. 10 Evolution of the best individual on different cluster sizes, averaged over 20 runs. The *bottom* figure is a zoom of the *top* figure, so that the different curves for 5, 10 and 20 machines can be seen correctly

- 5 populations of 16,384 individuals on 5 machines
- 10 populations of 8,192 individuals on 10 machines
- 20 populations of 4,096 individuals on 20 machines

The topology used for the experiments is a very straightforward fully connected network (each island uses the same IP file containing the IP numbers of all participating machines, and no machine can send an individual to itself).

Figure 10 top shows four curves. The black curve presents the evolution of the average fitness of the best individual of 20 runs for 14 h on the Weierstrass benchmark using a population of 81,920 individuals. The average best value found over 20 runs was 414.

Then, the three other curves show the average fitness of the best individuals of 20 runs on the same benchmark with 5 machines and 16,384 individuals per machine,

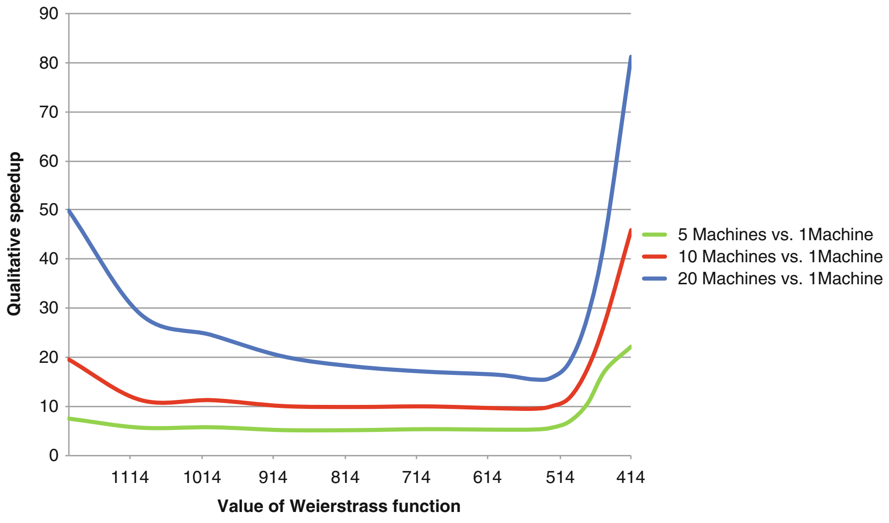


Fig. 11 Qualitative speedup factor determined by comparing one machine vs. 5, 10 and 20 machines in order to reach the same fitness value. These results represent an average of 20 runs

10 machines and 8,192 individuals per machine and finally, 20 machines and 4,096 individuals per machine until the value 414 is reached.

What is interesting to see is that the black curve for one machine shows a marked knee at around fitness 520. The huge 81,920-individual panmictic population quickly gets to this value, after which things get really difficult. This knee very probably shows that the whole population has converged to a local optimum. What happens next is probably only due to a random search implemented by the mutation function (Schwefel adaptive mutation in this algorithm), hence the very slow slope.

On the contrary, no marked knee is seen for the multi-island implementation (cf. Fig. 10 bottom that show the same curves, but over the first 700 seconds only so that the multi-island curves are more visible).

Search becomes more difficult, but no real knee is to be seen, and certainly not around value 500 even though the global population is exactly the same size.

As expected, 20 machines obtain results faster than one machine only, but this figure does not allow one to visualize the obtained speedup well.

In order to better see what is going on, Fig. 11 shows how long it takes each configuration to reach predefined values such as 1,100, 1,000, 900, …, by horizontally slicing Fig. 10 and looking at how much faster five machines reach these values than one machine and the same for 10 and 20 machines.

It can then be said that Fig. 11 shows *qualitative* speedup curves, i.e. how much faster a particular island configuration finds the same value compared to one panmictic population of the same size on a single machine.

Figure 11 can then be read the following way: it was about six times faster for a five-island configuration (one island per machine) to obtain a fitness of

1114, compared to one machine only, meaning that a slightly super-linear speedup is obtained for five machines. Indeed, five machines have exactly five times the computing power of one machine, so a perfectly linear speedup should have shown an exact 5× speedup.

After value 1114, five islands on five machines show a relatively constant speedup of 5× until value 520 is reached, after which qualitative speedup increases up to more than 20 for value 414 (these are all average values over 20 runs).

The speedup factor for a 10-machine cluster is quite similar: after a good start, speedup stabilizes at around 10× up to value 520 after which speedup rises to around 45×.

With 20 machines, speedup starts with a super-linear speedup of around 50× before going down to nearly 15× and rising again at around value 520, to slightly above 80×.

What probably happens here is that speedup is roughly linear with the number of machines until value 520, after which single machines with one panmictic population of 81,920 individuals get stranded in a sand dune pit. At this point, multi-island models show their advantage as they help each other out of local optima and continue their progression.

Because the speedup keeps rising after value 520, we call it a supra-linear speedup because it rises steeply.

Twenty machines get down to values around 300 in around 10 min, whereas it could take days or months for one machine to get there using random search only.

Interestingly enough, one sees an inflection on the five-machine configuration around value 450, probably because the five islands are getting stranded too, which is not the case for the 10- and 20-island configurations.

Finally, one can see that beyond value 500 (i.e. after single islands tend to get stuck in local optima), the speedup between multi-island configurations remains roughly the same: for value 414, the speedup of a five-machine cluster is slightly above 20×, while it is roughly 45× for 10 machines and above 80× for 20 machines, which is quite satisfying. The reader must be reminded that these curves are for GPU-islands for which Fig. 8 shows that they are already 160× faster than a sequential execution of the same code on a CPU. Therefore, during the linear speedup phase, the speedup of the 20 GPU cluster vs. one single machine is of $\sim 160 \times 20 = 3{,}200$, reaching $\sim 160 \times 80 = 12{,}800$ when single machines get stranded. On this problem, a one-day run on this cluster would therefore be equivalent to at least 35 years on a single island but probably much more as value 414 was obtained in less than 3 min only, and the speedup slope is quite steep.

This island-model speedup helped on the real-world problem of zeolite structure determination described in the chapter "Using Large-Scale Parallel Systems for Complex Crystallographic Problems in Materials Science", where 11 years of calculation were reduced to 3 days on the cluster of 20 machines described above (not taking supra-linear acceleration into account).

7 Conclusion

The EASEA platform is an easy way to test massive parallelization of evolutionary algorithms over one or several GPGPU cards and possibly over several homogeneous or heterogeneous computers equipped (or not) with such cards. The EASEA-CLOUD project will extend this to Grid computing and to the Cloud, so that, in the end, it is possible to use a supercomputer such as TITAN or a computational ecosystem to solve a single large problem.

EASEA is open source and is accessible through its web page: http://easea.unistra.fr.

It produces human-readable code, meaning that the EASEA platform can be used as a primer, to start on a massively parallel implementation (implement a basic program that implements your problem and then modify the C++ code to exactly suit your needs) or to code a complete project from A to Z.

EZ source files are portable, meaning that you can try them for yourself and exchange files on collaborative projects. This is why EASEA was included in the tutorial part of this book and why several chapters are based on it.

References

1. Alba, E., Tomassini, M.: Parallelism and evolutionary algorithms. IEEE Trans. Evol. Comput. **6**(5), 443–462 (2002)
2. Alba, E., Troya, J.: An analysis of synchronous and asynchronous parallel distributed genetic algorithms with structured and panmictic islands. In: Rolim, J., Mueller, F., Zomaya, A., Ercal, F., Olariu, S., Ravindran, B., Gustafsson, J., Takada, H., Olsson, R., Kale, L., Beckman, P., Haines, M., ElGindy, H., Caromel, D., Chaumette, S., Fox, G., Pan, Y., Li, K., Yang, T., Chiola, G., Conte, G., Mancini, L., Mery, D., Sanders, B., Bhatt, D., Prasanna, V. (eds.) Parallel and Distributed Processing. Lecture Notes in Computer Science, vol. 1586, pp. 248–256. Springer, Berlin (1999)
3. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18–20, 1967, Spring Joint Computer Conference, pp. 483–485. ACM, New York (1967)
4. Beyer, H.G., Schwefel, H.P.: Evolution strategies: a comprehensive introduction. Nat. Comput.: Int. J. **1**(1):3–52 (2002)
5. Branke, J., Kamper, A., Schmeck, H.: Distribution of evolutionary algorithms in heterogeneous networks. In: Genetic and Evolutionary Computation? GECCO 2004. Lecture Notes in Computer Science, vol. 3102, pp. 923–934. Springer, Berlin (2004)
6. Cantu-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers, Norwell (2000)
7. Collet, P., Schoenauer, M.: GUIDE: unifying evolutionary engines through a graphical user interface. In: Liardet, P., et al. (eds.) EA'03, Marseilles. Lecture Notes in Computer Science, vol. 2936, pp. 203–215. Springer, Berlin (2003)
8. Collet, P., Lutton, E., Schoenauer, M., Louchet, J.: Take it EASEA. In: Schoenauer, M., et al. (ed.): Proceedings of the 6th Conference on Parallel Problems Solving from Nature, LNCS 1917, pp. 891–901. Springer, Berlin (2000). http://sourceforge.net/projects/easea
9. Cramer, N.L.: A representation for the adaptive generation of simple sequential programs. In: Proceedings of an International Conference on Genetic Algorithms and their Applications, pp. 183–187 (1985)

10. Eshelman, L., Schaffer, J.D.: Real-coded genetic algorithms and interval-schemata. In: Whitley, L.D. (ed.) Foundations of Genetic Algorithms 2, pp. 187–202. Morgan Kaufmann, Los Altos (1993)
11. Fogel, D.B.: An analysis of evolutionary programming. In: Fogel, D.B., Atmar, W. (eds.) Proceedings of the 1st Annual Conference on Evolutionary Programming, pp. 43–51. Evolutionary Programming Society, La Jolla (1992)
12. Fogel, D.B.: Evolutionary Computing: The Fossil Record. IEEE Press, Los Alamitos (1998)
13. Fogel, L.J., Owens, A.J., Walsh, M.J.: Artificial Intelligence Through Simulated Evolution. Wiley, New York (1966)
14. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading (1989)
15. Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor (1975)
16. Jiang, J., Jorda, J.L., Yu, J., Baumes, L.A., Mugnaioli, E., Diaz-Cabanas, M.J., Kolb, U., Corma, A.: Synthesis and structure determination of the hierarchical meso-microporous zeolite itq-43. Science **333**(6046), 1131–1134 (2011)
17. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Evolution. MIT Press, Cambridge (1992)
18. Maitre, O., Kruger, F., Querry, S., Lachiche, N., Collet, P.: Easea: specification and execution of evolutionary algorithms on GPGPU. J. Soft Comput. **16**(2), 261–179 (2012)
19. Maitre, O., Lachiche, N., Collet, P.: Two ports of a full evolutionary algorithm onto GPGPU. In: Hao, J.K., Legrand, P., Collet, P., Monmarche, N., Lutton, E., Schoenauer, M. (eds.) Artificial Evolution. Lecture Notes in Computer Science, vol. 7401, pp. 97–108. Springer, Berlin (2012)
20. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. In: Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk (With contributions by J. R. Koza) (2008)
21. Rechenberg, I.: Evolutionstrategie: Optimierung technischer Systeme nach Prinzipien des biologischen Evolution. Frommann-Holzboog Verlag, Stuttgart (1973)
22. Schwefel, H.P.: Numerical Optimization of Computer Models. Wiley, New York (1981) [1995—2nd edn.]
23. Van Luong, T., Melab, N., Talbi, E.-G.: GPU-based Island Model for Evolutionary Algorithms. In: Genetic and Evolutionary Computation Conference (GECCO), Portland, USA (2010)
24. Whitley, D., Rana, S., Heckendorn, R.B.: The island model genetic algorithm: on separability, population size and convergence. J. Comput. Inform. Technol. **7**, 33–48 (1999)

Part II
Implementations of Various EAs

Generic Local Search (Memetic) Algorithm on a Single GPGPU Chip

Frédéric Krüger, Ogier Maitre, Santiago Jiménez, Laurent A. Baumes, and Pierre Collet

Abstract Memetic algorithms (MAs), evolutionary algorithms coupled with a local search routine, have been shown to be very efficient in solving a great variety of problems. This chapter presents the first implementation of a generic parallel MA on a general-purpose graphics processing unit card. An upgrade of the EASEA platform provides an automatic generation and parallelization of an MA for both novice and experienced users. Experiments on a benchmark function and a real-world problem reveal speedups ranging between ×70 and ×120, depending on population size and number of local search iterations.

1 Introduction

Nature is a great source of inspiration. Darwin was inspired by nature when he wrote his thesis about evolution, and evolutionary algorithms (EAs) are directly inspired by Darwin's theory of evolution. If in real life living organisms are being evolved in a process that takes millions of years, in artificial evolution solutions to a problem are being evolved to reach an optimum. These solutions participate in an evolutionary process of refinement of their genetic material through reproduction and selection.

Where in an evolutionary algorithm an individual is able to reproduce right after its creation, in real life an individual will acquire some experience, age, mature, and "improve" himself before he starts participating in the great process

F. Krüger (✉) · O. Maitre · P. Collet
University of Strasbourg, ICUBE, Illkirch, France
e-mail: Frederic.Kruger@etu.unistra.fr; Ogier.Maitre@unistra.fr; Pierre.Collet@unistra.fr

S. Jiménez · L.A. Baumes
Insituto de Tecnologia Quimica, UPV-CSIC, Valencia, Spain
e-mail: sanjiser@gmail.com; baumesl@gmail.com

of evolution, hence going through some sort of “local optimization.” Locally optimizing an individual in an evolutionary algorithm transforms the latter into a memetic algorithm (MA). The literature is full of papers from different fields that report on the application of MAs in problem solving.

Desktops and even laptop computers are nowadays routinely equipped with very powerful general-purpose graphics processing unit (GPGPU) cards. The latter are starting to revolutionize the field of evolutionary computation. Recent papers report great speedups obtained by parallelizing the evaluation function on GPGPU cards [9, 11]. Considering these astonishing results, it only seems natural to see what parallelizing the local search can bring as speedups for MAs.

The aim of this chapter is to present the implementation of a generic MA (i.e., an evolutionary algorithm coupled with a local search [6]) parallelized on a GPGPU card along with the obtained speedups on a benchmark function and a real-world problem. This chapter also reveals how the parallel MA was integrated into the EASEA[1] platform [10]. This important step performs automatic generation and parallelization for researchers without their having to master the programming of GPU cards.

2 Memetic Algorithm

Memetic algorithms have already been successfully applied to a multitude of real-world problems. Also known as “hybrid algorithms”, their uniqueness comes from the combination of a population-based global search approach and a separate individual local search process.

2.1 The Origin of Memetic Algorithms

The term “meme” was first introduced by the British scientist Richard Dawkins in his 1976 book *The Selfish Gene* [4]. In the Oxford English Dictionary, it is defined as:

> An element of culture that may be considered to be passed on by non-genetic means.

The first appearance of the term MA was in a technical report of 1989 written by Moscato [12]. MAs are often referred to as “Lamarckian evolutionary algorithms” or “Baldwinian evolutionary algorithms.” These names are a tribute to two great personalities who greatly influenced the field of evolution: Lamarck and Baldwin. Jean-Baptiste Pierre Antoine de Monet, Chevalier de Lamarck (1744–1829), known

[1] http://easea.unistra.fr

simply as Lamarck, was a French naturalist widely remembered for his theory of inheritance of acquired characteristics, also called Lamarckism.

The inheritance of acquired characteristics stipulates that physiological changes (phenotypical changes) acquired by an organism during its lifetime can be transmitted in genetic form to its offspring. For instance, a blacksmith who strengthens his hands through repeated use will pass on this characteristic to his children.

James Mark Baldwin (1861–1934) was an American philosopher and psychologist whose most important legacy is the concept of the Baldwin effect or "Baldwinian evolution," presented in an 1896 paper *A New Factor in Evolution*. Baldwin believed that human behavioral decisions made and sustained over generations as "traditions" ought to be considered among the factors influencing the human genome. In his paper, Baldwin suggested that during evolution individuals are selected for their potential in learning new skills, rather than their fixed genetically coded abilities.

To summarize:

- In Lamarckian evolution, individuals are selected for their genome that is the result of the refined phenotype of their ancestor.
- In Baldwinian evolution, individuals are selected for their potential to improve themselves.

2.2 Memetic Algorithms in Artificial Evolution

To understand the mechanism behind an MA, one has to remember the mechanism of a standard evolutionary algorithm (SEA) [5]. The algorithm starts by creating, initializing, and evaluating a population of individuals referred to as the parent population. Then, until the stopping criterion is met, the algorithm will repeat the following process:

1. Select individuals from the parent population.
2. Create new individuals from the selected parents using crossover and mutation operators.
3. Evaluate the new individuals.
4. Insert the new individuals into a new population referred to as the offspring population.
5. Replace less-fitted parents with better-fitted children.

Figure 1 depicts the flowchart of the EASEA-generated evolutionary algorithm. The mechanism behind MAs is very similar to the routine described above. Being population-based algorithms, they also manage a parent and an offspring population. New individuals are created using crossover and mutation operators.

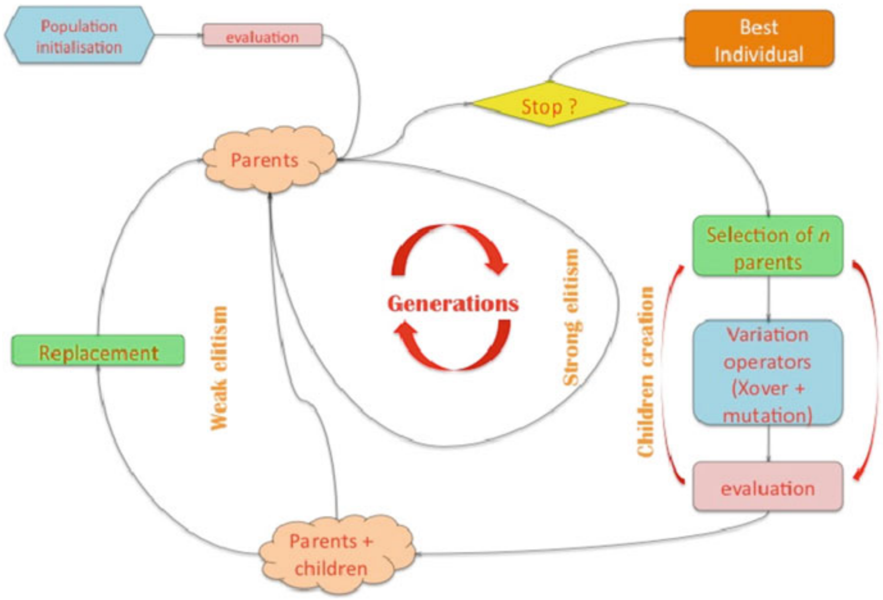


Fig. 1 Flowchart of the standard evolutionary algorithm

Algorithm 1 Local search algorithm applied to each selected individual

```
function LocalSearch(individual)
    Evaluate(individual)
    while Termination Criterion not met do
        newIndividual ← Improve(individual)
        Evaluate(newIndividual)
        if newIndividual better than individual then
            individual ← newIndividual
        end if
    end while
    return individual
end function
```

Individuals with better fitness replace less fit individuals. The main difference lies in the addition of a local search that tries to improve the newcomers. The local search is given an individual as input. The latter is evaluated. Then, until the stopping criterion is met, an individual from the neighborhood of the original individual is picked. The new individual is evaluated and tested against the original one. If the new individual has better fitness than the original, he will take its place. If the new individual does not have better fitness than the original one, he is discarded. Once the stopping criterion is met, the new individual (or original individual if no improvement has been made) is returned. Algorithm 1 gives an overview of the local search.

The two evolutionary trends presented in the previous section, namely, Lamarckian evolution and Baldwinian evolution, can be directly applied to the implementation of an MA. As explained in Sect. 2.1, Lamarckian evolution selects individuals who are best fitted by local improvement for a specific task, whereas Baldwinian evolution selects individuals with the best potential.

In artificial evolution, these trends trigger different behaviors. If we want to make the MA behave in a Lamarckian way, the local search has to return the improved genome of the individuals. The genome of the original individual will be lost, and the next generations will be directly influenced by the result of the local search.

If we want the MA to behave in a Baldwinian way, then the local search has to return the fitness of the improved individual and the genetic material of the original individual. The genome of an individual is not changed, and its potential is given by the fitness it could achieve if improved (fitness of the individual found by the local optimization). Deciding whether to perform Lamarckian or Baldwinian evolution necessarily has an impact on the direction of the global search, on the convergence of the population, and on the results. Some papers show the influence of both trends on the evolution of busy beavers [14], but no such studies have been made for this research.

2.3 Parallel Memetic Algorithms

Some papers have been written about parallel MAs, but they mainly concern very specific problems. Munawar et al. solved the Max-SAT problem by adding a local search routine to an evolutionary algorithm [13]. They obtained a speedup of $\times 25$ with an expensive TESLA machine. Wong et al. present a speedup of $\times 4.24$ [17]. Luo et al. report solving a similar problem to Max-SAT using local search procedures and graphics hardware [8]. However, no work on the generic implementation of a parallel MA or of an evolutionary algorithm coupled with a local search routine was found.

3 Implementation of a Parallelized Memetic Algorithm

The parallelized MA presented in this chapter is based on the GPU parallelized evolutionary algorithm generated by the EASEA platform. It was first developed separately from EASEA and integrated into the language later on. This section gives a brief overview of what EASEA is all about and why it was chosen as the host for the parallelized MA. The different modifications made to the EASEA language are then described.

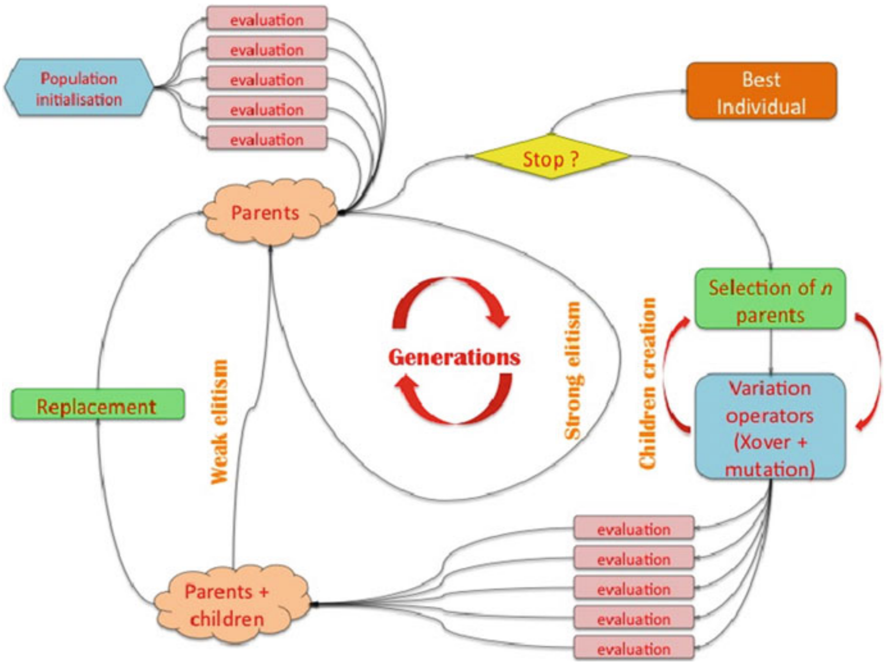


Fig. 2 Flowchart of the GPU parallelized evolutionary algorithm generated by the EASEA platform

3.1 The EASEA Platform

The EASEA platform, mainly composed of the EASEA language [2], was designed to allow “anybody” with a bit of programming skill to describe, implement, and generate their evolutionary algorithm, whether for experimenting, teaching, or real-world problem solving. The user only has to specify problem-dependent functions and parameters, such as genome structure and fitness function, and EASEA takes care of generating the evolutionary algorithm that encapsulates the problem. The user is relieved of the tedious task of having to program the algorithm himself and is provided with the source code for a full evolutionary algorithm.

To transcribe the EASEA language into C++ and CUDA source code, the EASEA platform has a compiler at its disposal that uses basic functions implemented in the EASEA library, and user-readable templates of evolutionary algorithms. A recent addition to the EASEA platform [9] is the possibility to parallelize the evaluation function on one or several GPU cards by using the `-cuda` option.

Figure 2 shows the flowchart of the EA parallelized on a GPU as it is generated by EASEA. Similar to the algorithm presented in Fig. 1, EASEA keeps on the CPU the evolutionary engine that has the task of generating and managing the population of individuals to evaluate. The evaluation function however is compiled for the GPU using the `nvcc` (nVidia C Compiler) compiler and transferred to the GPU card. Then, whenever a population of individuals has to be evaluated, the evolutionary engine transfers it onto the GPU where all the individuals are evaluated in parallel. Once the GPU is done evaluating each individual, it returns an array containing the fitnesses to the engine on the CPU to continue the evolutionary loop.

3.2 Implementing a Parallel Memetic Algorithm

The parallel MA presented in this section relies greatly on the parallelized evolutionary algorithm generated by EASEA (see Fig. 2). The aim of the implementation was not to parallelize the complete MA on the GPU but only to port the local search routine to the graphic card.

As can be seen in Algorithm 1, the local search procedure is mainly composed of an optimization function and the evaluation function that is called right after. The process is repeated for a certain number of iterations. Figure 3 gives a good overview of how the parallel MA works (with the exception that the initial population also goes through a local optimization).

Porting the local search algorithm to the GPU in a generic way required some choices to be made. In a standard MA a subset of the population is selected to go through a local search process. We chose in our implementation to give each individual the chance to improve. This choice was made to fill the GPU cores as much as possible in order to achieve the best possible speedup.

We did not want to have to decide between turning our parallel MA into a Lamarckian evolution or a Baldwinian evolution. Both directions can be useful for future users, and we wanted to keep things as generic as possible. For that reason we decided to implement both tendencies. The main difference between the two implementations lies in the final transfer from the GPU to the CPU. For a Lamarckian evolution, the genomes of the locally optimized individuals are transferred alongside the fitness, and the original individuals are replaced with the new ones. For a Baldwinian evolution, only the fitnesses of the locally optimized individuals are transferred back to the CPU and these are assigned to their respective original individuals.

The main challenge resided in keeping the local search algorithm generated as generic as possible. The goal was to be able to give users as much freedom as possible in designing their local search algorithm but also relieving them from the laborious population and memory management.

As a matter of fact, the local search algorithm requires a lot of memory management: three different versions of each individual have to be stored:

- The original individual
- A current best optimized version of the original individual

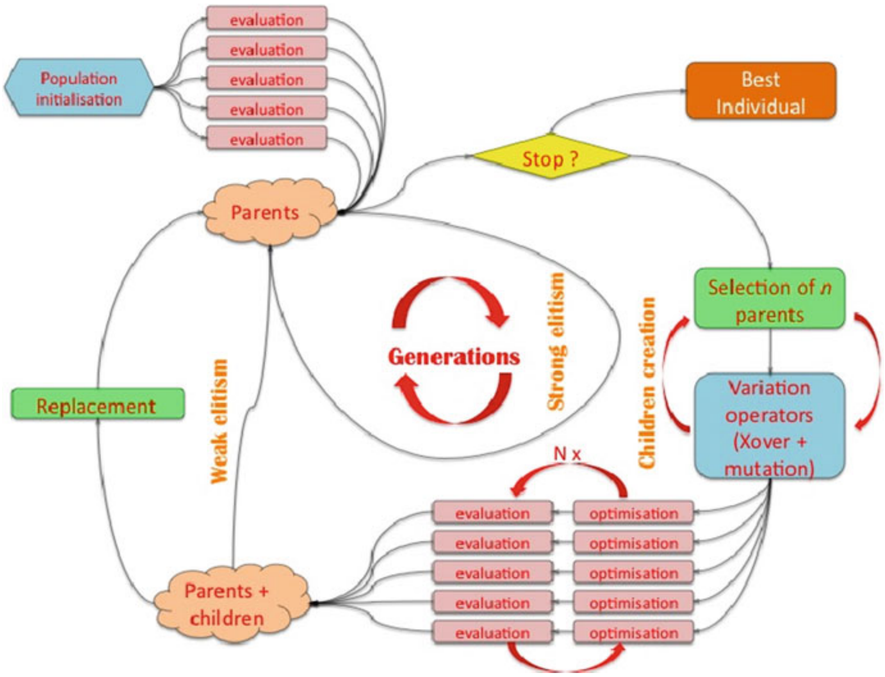


Fig. 3 Flowchart of the GPU parallelized MA generated by the EASEA platform

- A newest "improvement" of the best individual (challenger to the best current individual)

All these data have to be present in the global memory of the GPU at all times. The idea of using the GPU core's local memory to boost the search even more was quickly dropped in order to keep things as generic as possible. Loading the individuals into the very limited local memory of a GPU core implied restricting the maximum genome size an individual could have.

A difficult choice to make was the implementation of the local search loop. Letting EASEA design the local search loop helps the user in that he only has to focus on writing the function that improves the individual. EASEA then takes care of calling the evaluation function at the right time, of comparing individuals, and of keeping an up-to-date version of the best genome. That solution may seem very attractive but restricts the user to very basic optimization functions.

Letting the user write the complete local search algorithm opens a lot of possibilities to more complex optimization functions. But this freedom comes with a price: the responsibility for making sure memory is allocated, individuals are not dropped, and they are stored in the right place to be transferred back to the CPU lies with the user. It also requires the latter to be familiar with the

Algorithm 2 Local search algorithm generated by EASEA

```
function GPULocalSearch(population)
    for all $individual$ in population do in parallel
        Call CustomEvaluation(individual)
        for $i = 1 \rightarrow numberOfIterations$ do
            $newIndividual \leftarrow$ CustomOptimizationFunction(individual)
            Call CustomEvaluation(newIndividual)
            if newIndividual better than individual then
                $individual \leftarrow newIndividual$
            end if
            $i \leftarrow i + 1$
        end for
        $individual \rightarrow optimizedPopulation$
    end for
    return optimizedPopulation
end function
```

restrictions of GPU programming. Give experienced users more freedom at the cost of frightening the novice? Or set boundaries in order to relieve users as much as possible of uninteresting tasks? We did not want to compromise and chose to offer both alternatives. The solution to this problem lies in the `Number of Search Iterations` parameter. Algorithm 2 gives a clear overview of the generated local search algorithm as well as the call sequence of the different user-created functions. If the user only wants to write the optimization function, he sets the desired number of search iterations, and EASEA takes care of managing the rest of the local search algorithm. If the user wants to write a more elaborate local search algorithm (for instance, create n variations of an individual instead of just 1), he simply sets the `numberOfIterations` parameter to 1 and writes a brand new local search loop in the custom optimization function (that loop will require an independent number of search iterations). The custom optimization function is only called once by the EASEA-generated local search algorithm.

3.3 Changes Made to EASEA

Several additions were made to the EASEA language in order to let it automatically generate parallel MAs. First, a couple of MA-specific parameters were added to the language syntax:

- A boolean that specifies whether the evolution is Lamarckian or Baldwinian
- The number of local search iterations to perform

A parameter (`-memetic`) was also added to the EASEA compiler to identify the MA template as the template to be used. Several functions had to be added to the EASEA library such as the local search loop management function and a transfer function for the locally optimized population from the GPU back to the CPU. Finally a new field had to be added to the EASEA language where the user can either specify the improvement function only or a complete local search algorithm. In the end, the new MA fitted pretty well into the EASEA language, and the EASEA platform can now be used to automatically generate a GPU-parallelized MA.

4 Designing the Local Search Algorithm

At the time of the experiments, GPGPU cards were not supplied with random number generators. Even though pseudorandom generators had been successfully implemented [7], we chose to design a deterministic local search algorithm. Newer CUDA environments are fitted with a proper random number generator.

The local search algorithm presented in this section was used for the experiments described later on in this chapter. As explained above the local search is deterministic for lack of a proper random number generator. It requires as input a specific step and a specific number of search iterations. The main engine is pretty straightforward. Until the maximum number of search iterations is reached, the algorithm adds the step value to the first dimension of the individual, looking into the neighborhood of that dimension. The modified individual is then evaluated and its fitness compared to the fitness of the best known individual. If the fitness improved, the best individual is replaced by the improved one.

If the fitness of the individual improved on the first try, the algorithm keeps adding the step value to the same dimension until the individual stops improving, in which case the process is repeated on the next dimension. If the fitness of the individual did not improve on the first try, the algorithm will start exploring in the opposite direction. Once the algorithm has browsed through all the dimensions, it starts over on the first dimension and repeats the process described above until the maximum number of search iterations is reached. Algorithm 3 gives a more detailed description of the local search algorithm's modus operandi.

The function presented in this section is very primitive in that the step added to the dimension is not adaptive and the step is not changed once all the dimensions have been browsed. But the aim of this research was not to find the best possible local search algorithm that would solve any problem. Its goal was to give a simple idea of what kind of local optimization functions can be written with the EASEA language and observe the speedup obtained for parallelizing the local search on a GPU card.

Algorithm 3 Custom local search loop used for the experiments

```
function CUSTOMLOCALSEARCHLOOP(individual)
    // initializing variables
    $step \leftarrow 0.001$
    $N \leftarrow genomeSize$
    $currentDimension \leftarrow 0$
    $firstAttempt \leftarrow$ TRUE

    // initializing genomes
    $tempGenome \leftarrow inidividual.Genome$
    $originalFitness \leftarrow$ Evaluate($genome$)
    $tempFitness \leftarrow originalFitness$

    for $i \leftarrow 0$ to $numberOfIterations$ do
        $tempGenome[currentDimension] \leftarrow Genome[currentDimension] + step$
        $tempFitness \leftarrow$ Evaluate($tempGenome$)
        if $tempFitness < originalFitness$ then
            // replace genome
            $originalFitness \leftarrow tempFitness$
            $individual.genome\ [currentDimension] \leftarrow tempGenome\ [currentDimension]$
            if $firstAttempt$ =TRUE then
                $firstAttempt \leftarrow$ FALSE
            end if
        else
            // readjust step
            if $firstAttempt$ =TRUE then
                if $step < 0$ then
                    $currentDimension \leftarrow (currentDimension + 1)\%N$
                end if
                $step \leftarrow step \times (-1)$
            else
                $currentDimension \leftarrow (currentDimension + 1)\%N$
                $firstAttempt \leftarrow$ TRUE
            end if
        end if
    end for
    return $individual$
end function
```

5 Experiments

The purpose of the experiments was not to measure the efficiency of the MA or of the local search algorithm in finding the global optimum but rather to measure the speedup that parallelizing the local search on a GPU chip brings.

5.1 Experimental Conditions

The experiments were performed on "one half" of a GTX295 nVidia card[2] *versus* a 3.6 GHz Pentium IV with 3 GB RAM. The machine was running under Linux 2.6.27 32 bits with nVidia driver 190.18 and CUDA 2.3. "One half" of a GTX295 means that only one of the two GPUs present on the GTX295 was used for the experiments. This decision was made in order to have a fair one-to-one comparison: one GPU chip *versus* one CPU core. The timings used to compute the speedups were performed with the `gettimeofday()` POSIX function. Two different sets of speedups were measured:

1. Speedup on the whole evolutionary algorithm
2. Speedup on the local search algorithm only

We chose a Lamarckian evolution for the experiments so as to include the final transfer of the population from the GPU back to the CPU in the timing measurements. When only the local search algorithm was timed, what was really measured was:

- For the CPU run: the time needed to perform the local search for each individual
- For the GPU run: the time needed to perform the local search for each individual in parallel and population transfer time from and to the GPU (added for fairness)

5.2 Rosenbrock Function

The first set of experiments was performed on the Rosenbrock function. The latter is a non-convex function, often used as a test problem to measure the performance of optimization algorithms. The Rosenbrock function was first introduced by Howard H. Rosenbrock in 1960 [15]. The function has the following definition:

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2 \tag{1}$$

Figure 4 shows that the global minimum is located inside a long parabolic-shaped valley whose landscape is flat. The challenge associated with this function is not finding the valley but converging to the global minimum located at $(x, y) = (1, 1)$. The Rosenbrock function has several multidimensional generalizations, one of which was used to test the MA [16]. It has the following definition:

$$f(x) = (x_1, x_2, \dots, x_N) = \sum_{i=1}^{N/2} [100(x_{2i-1}^2 - x_{2i})^2 + (x_{2i-1} - 1)^2] \tag{2}$$

[2] www.gefore.com/Hardware/GPUs/gefore-gtx-295

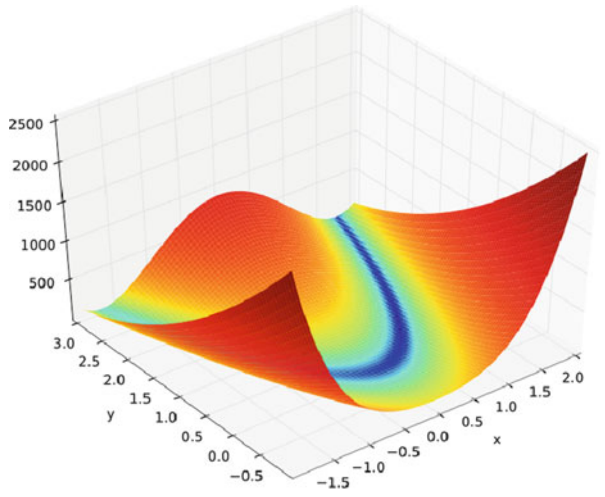


Fig. 4 Plot of the Rosenbrock function for two dimensions

This equation shows that this multidimensional generalization is only defined for an even number N. We chose the Rosenbrock function mainly because it is fast to compute. The idea was to expose the overheads induced by the very fast benchmark function as much as possible. Therefore, the speedups exposed in the following section are amongst "*the worst*" that could be achieved, but give a fair idea of the advantages of parallelizing the local optimization function on the GPU.

5.3 *Results*

Several experiments were performed to measure the different speedups achieved by parallelizing the local search on a GPU chip. First, we measured the speedup achieved on the whole MA, i.e., the evolutionary algorithm running on the CPU and the local search parallelized on the GPU in one case *versus* both the evolutionary algorithm and the local search running on the CPU.

Then, we measured the speedup achieved on the local search only. Therefore, we compared the time required by the local search on the CPU with the time required by the parallelized local search on the GPU. Finally, we measured the impact of the population transfer time to and from the GPU. The local search algorithm used during the experiments was the one described in Sect. 4.

5.3.1 Speedup on the Whole Algorithm

Figure 5 shows the speedup obtained for the complete MA generated by the EASEA platform. The influence of two parameters was observed: the size of the population

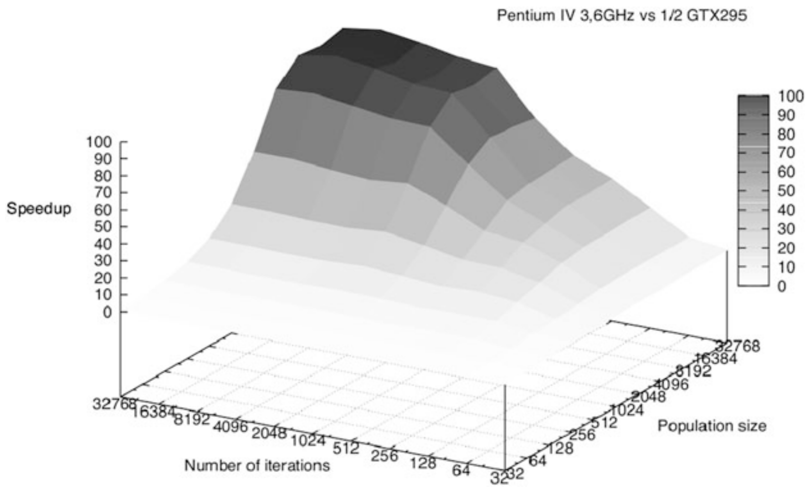


Fig. 5 Speedup for the whole algorithm

and the number of iterations the local search had to perform for each individual. The maximum speedup reaches a plateau at around ×91 for a population size of 32 K individuals and 32K iterations. Figure 5 reveals that for a population size of fewer than 2,048 individuals and fewer than 256 iterations, the achieved speedup is not considerable. The cores of the GPU are not fully loaded for fewer than 2,048 individuals.

A reasonable speedup of ×47 is obtained for 2,048 individuals and 2,048 local search iterations. We can see that a large number of individuals and local search iterations is required to overcome the overhead of the evolutionary algorithm running on the CPU. The maximum speedup of ×95 is obtained for 32 K individuals and 16 K local search iterations.

5.3.2 Speedup on the Local Search Only

Figure 6 shows the speedup obtained on the local search only. What was actually measured was:

- The population transfer time from the CPU to the GPU
- The improvement and evaluation of all the individuals for a certain number of iterations
- The population transfer time from the GPU to the CPU

For this experiment, the time required by the evolutionary algorithm running on the CPU was ignored. The maximum speedup reaches a plateau at around ×120 for a population of 32 K individuals and numbers of local search iterations starting at 256. Two thousand and forty-eight individuals are already sufficient to achieve

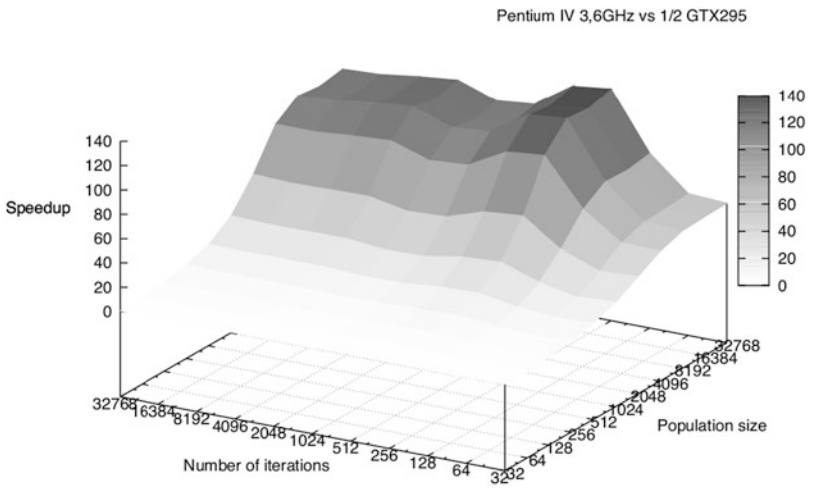


Fig. 6 Speedup for the local optimization only

maximum speedup, but below this number of individuals, the cores of the GPU card are not fully loaded. It is important to remember that the benchmark function used for these experiments is very fast to compute and was chosen to maximize the influence of the overhead. The presented speedups are "specific" to this function, and different speedups will be obtained for different functions.

Figure 6 shows that a reasonable speedup of ×48 is reached for 2,048 individuals and 256 iterations. No reasonable explanation was found for the "pass" observed for 1,024 and 2,048 iterations and for population sizes greater than 8 K.

Comparing Figs. 5 and 6 reveals that the number of local search iterations does not influence the speedup obtained for the local search as much as it influences the speedup obtained for the whole MA. In fact, more local search iterations are required to get a reasonable speedup for the whole algorithm. More local search iterations imply more calls to the evaluation function, which is usually very time consuming. In order to overcome the overhead of the evolutionary algorithm running on the CPU, more local search iterations are required to achieve an interesting speedup.

What both Figs. 5 and 6 reveal is that one must use large populations to benefit from the power of the GPU cards. Since their power comes from their parallel architecture, the more individuals are used, the more the cores of the card are filled, making it possible to achieve great speedups.

5.3.3 Influence of Population Size on Transfer Time

The GPU parallelization of the local search has to deal with different population transfers: an initial transfer of the population from the RAM to the GPU global memory and a final transfer of the local optimization results from the GPU back to

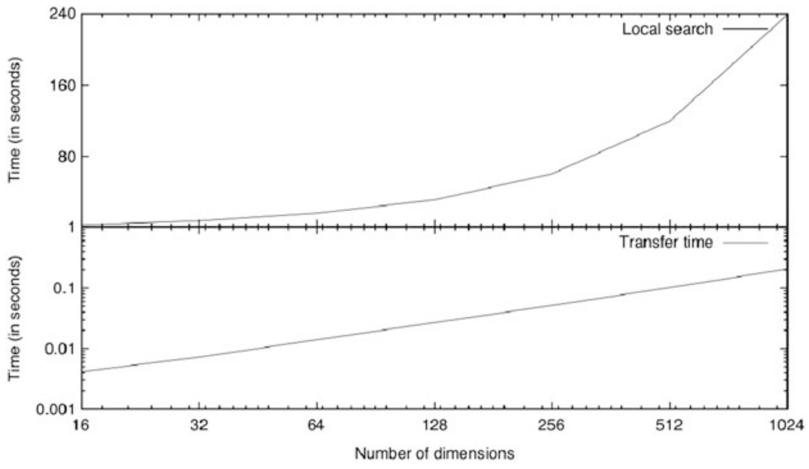


Fig. 7 Influence of genome transfer time

the RAM. It is therefore relevant to verify the influence of the time required by the memory transfers.

The experiment presented in this section was performed for a Lamarckian evolution. Therefore, on the final transfer of the population, the genome of the locally optimized individuals was transferred alongside the fitnesses.

Figure 7 (top) shows the time required by the local search for 32 K individuals and 32 K iterations with regard to the number of dimensions of the problem. The genome size in bytes is obtained by multiplying the number of dimensions by 4. Figure 7 (bottom) shows the time required for the transfer of the population of 32 K individuals to the GPU and back. The scale of the plot is logarithmic.

Figure 7 shows that while the time required by the local search ranges from 3.44 s for 16 dimensions to 238.9 s for 1,024 dimensions, the transfer time does not exceed 0.2 s for 1,024 dimensions. 32 K Individuals each with 1,024 dimensions represent 128 KBytes! The very fast transfer time is explained by the fact that GPU cards are designed to transfer millions of pixels per image frame, hence having to deal with huge throughput. Figure 7 demonstrates that the transfer time is negligible compared to the time taken by the local search.

6 Real-World Experiment

The experiments with the Rosenbrock function revealed some very interesting speedups. But running a benchmark function is never the same as running a real-world problem fitness function. Therefore, we decided to run the same setup used for the Rosenbrock benchmark on a chemistry problem: crystal structure prediction.

6.1 Crystal Structure Prediction: Zeolites

Zeolites are porous crystalline structures with important applications in industry, such as filtering, catalysis, and energy storing, and medicine. The crystal is made out of tetrahedrons of oxygen that contain a single silicon or aluminum atom in their center. When the quality of the crystal is measured using its structure, only a small set of atoms is used for the actual work. Indeed, the structure of the crystal is a combination of different symmetry operations applied on smaller cells. The small set of atoms used in the case of this research represents the smallest cell in which there is no element of symmetry. The structure of the crystal is reconstructed using the small set of atoms along with various relevant information concerning the zeolite we are looking for.

The fitness function for this complex problem evaluates the probability for a crystal structure to be a zeolite. It takes a lot more time to evaluate than the very fast Rosenbrock benchmark. It can be expressed by the following equation:

$$F(a_x, a_y, a_z, \ldots, n_x, n_y, n_z) = (F_a + F_b + F_c)/1000 \times (1 + F_d) \qquad (3)$$

where F_a, F_b, F_c, and F_d are functions of the atoms' distances, angles, average angle, and connectivity, respectively. $(a_x, a_y, a_z, \ldots, n_x, n_y, n_z)$ are the (x, y, z) coordinates of the n atoms of the crystal structure. More information about the fitness function can be found in other papers [1,3]. Behind this elementary equation lie more than 400 lines of source code that incorporates a huge chemistry library.

For the experiments we used a structure with four atoms. The genome of an individual was therefore made out of 12 floats. The chosen zeolite represented an "average" structure in terms of fitness evaluation time and number of atoms.

6.2 Results

As for the benchmark function, the experiment does not reveal the quality of the results obtained with a parallelized local search function but simply measures the speedup gained with the use of a GPU card.

Figure 8 shows the speedup obtained for the zeolite problem with regard to population size and number of local search iterations. The figure reveals a much more predictable surface where the number of local search iterations does not impact the speedup very much. This can be explained by the fact that the evaluation function takes much more time to compute than the very fast Rosenbrock benchmark, hence minimizing the overhead of the evolutionary algorithm. Thirty-two local search iterations are already enough to keep the GPU cores busy.

As for the results presented in Sects. 5.3.1 and 5.3.2, a population of at least 2,048 individuals is required to fill the GPU card and get a fair speedup of $\times 71$. Then, the more individuals are added to the population, the better the card will optimize its scheduling and the better the speedup will get, achieving a maximum value of $\times 91$

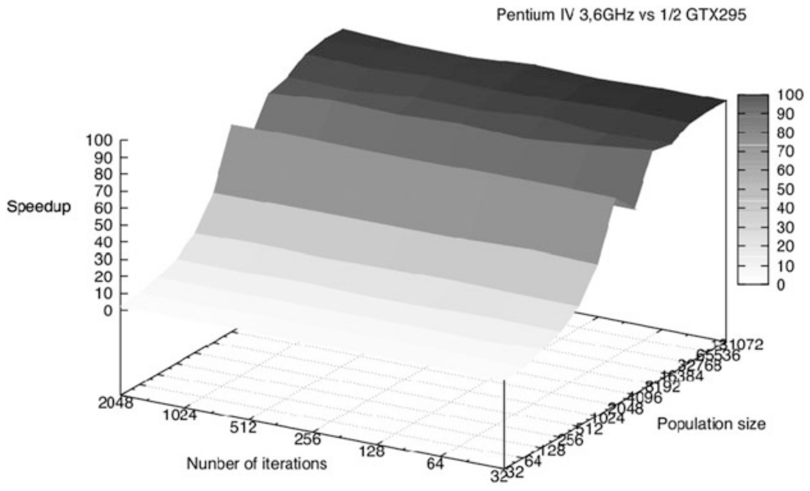


Fig. 8 Speedup on a real-world problem

for 65 K individuals and 32 local search iterations. No explanation was found for the drop for 4,096 individuals.

7 Conclusion

Memetic algorithms can take full advantage of being executed on highly parallel GPGPU cards. Speedups of $\times 120$ for a benchmark function and of $\times 91$ for a real-world problem support that fact. But to achieve such accelerations, *huge* population sizes have to be used in order to maximize the scheduling of the GPU cores. Speedups of $\times 47$ can already be obtained for populations with as few as 2,048 individuals.

The tests done for this research only involved one of the two GPU cards present in the GTX295. But current motherboards can welcome up to four GPGPUs. The version of EASEA used at the time of the experiments did not allow the parallelization of the evaluation function over several GPU cards. That feature was added in later versions and is now also available for the MA. By multiplying the population size by the number of available GPUs, the population is split between the cards, and the obtained speedups can be expected to rise according to the number of cards (a local search carried out on two cards can be expected to be two times faster than a local search on a single card).

To allow the EC community to benefit from the parallel MA without having to go through the tedious task of writing it, the latter was integrated into the EASEA platform. The local search algorithm detailed in Sect. 4 is included as an example (`rosenbrock.ez`) with the EASEA bundle and free for the community to use as a starting point to implement their own problems. The same EASEA project can be compiled for CPU if no option is chosen and for GPU (if the computer is equipped with a GPGPU card) if the `-cuda` option is chosen.

Another recent addition to the EASEA language creates a lot of opportunities for future studies: an asynchronous island model for heterogeneous parallel machines and algorithms. Study of the cohabitation of purely evolutionary environments and memetic environments or the impact of exchanges between Lamarckian and Baldwinian algorithms on the evolution of several islands could bring interesting insights into the potential of MAs.

References

1. Baumes, L.A., Krüger, F., Collet, P.: Using large scale parallel systems for complex crystallographic problems in materials science. In: Tsutsui, S., Collet, P. (eds.) Massively Parallel Evolutionary Computation on GPGPUs. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37959-8
2. Collet, P., Lutton, E., Schoenauer, M., Louchet, J.: Take it EASEA. In: PPSN 2000. Lecture Notes in Computer Science, vol. 1917, pp. 891–901. Springer, Berlin (2000)
3. Corma, A., Moliner, M., Serra, J.M., Serna, P., Diaz-Cabana, M.J., Baumes, L.A.: A new mapping/exploration approach for HT synthesis of zeolites. Chem. Mater. **18**, 3287–3296 (2006)
4. Dawkins, R.: The Selfish Gene. Oxford University Press, Oxford (1989).
5. De Jong, K.A.: Evolutionary Computation, a Unified Approach. MIT Press, Cambridge (2006). ISBN 0-262-04194-4
6. Hart, W.E., Krasnogor, N., Smith, J.E.: Recent Advances in Memetic Algorithms. Springer, Heidelberg (2005)
7. Langdon, W.B.: A fast high quality pseudo random number generator for graphics processing units. IEEE World Congress on Computational Intelligence, Hong Kong, pp. 459–465 (2008)
8. Luo, Z., Liu, H.: Cellular genetic algorithms and local search for 3-SAT problem on graphic hardware. IEEE Congress on Evolutionary Computation CEC 2006, Vancouver, pp. 2988–2992 (2006)
9. Maitre, O., Baumes, L.A., Lachiche, N., Corma, A., Collet, P.: Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA. In: GECCO, pp. 1403–1410 (2009)
10. Maitre, O., Kruger, F., Querry, S., Lachiche, N., Collet, P.: EASEA: specification and execution of evolutionary algorithms on GPGPU. J. Soft Comput. **16**(2), 261–179 (2012)
11. Maitre, O., Lachiche, N., Collet, P.: Fast evaluation of GP trees on GPGPU by optimizing hardware scheduling. In: EuroGP 2010. Lecture Notes in Computer Science, vol. 6021, pp. 301–312. Springer, Heidelberg (2010)
12. Moscato, P.: On evolution, search, optimization, genetic algorithms and martial arts: towards memetic algorithms. Caltech Concurrent Computation Program (report 826) (1989)
13. Munawar, A., Wahib, M., Munetomo, M., Akama, K.: Hybrid of genetic algorithm and local search to solve MAX-SAT problem using NVIDIA CUDA framework. Genet. Program. Evol. Mach. **10**(4), 391–415 (2009)
14. Pereira, F.B., Costa, E.: Understanding the role of learning in the evolution of Busy Beavers: a comparison between the Baldwin Effect and a Lamarckian strategy. In: Proceeding of the Genetic and Evolutionary Computation Conference (2001)
15. Rosenbrock, H.H.: An automatic method for finding the greatest or least value of a function. Comput. J. **3**, 175–184 (1960). doi:10.1093/comjnl/3.3.175
16. Shang, Y.W., Qiu, Y.H.: A note on the extended Rosenbrock function. Evol. Comput. **14**(1), 119–126 (2006)
17. Wong, M., Wong, T.: Parallel hybrid genetic algorithms on consumer-level graphics hardware. IEEE Congress on Evolutionary Computation CEC 2006, pp. 2973–2980 (2006)

arGA: Adaptive Resolution Micro-genetic Algorithm with Tabu Search to Solve MINLP Problems Using GPU

Asim Munawar, Mohamed Wahib, Masaharu Munetomo, and Kiyoshi Akama

Abstract In this chapter, we propose a new approach to solve the most general form of global optimization problems, namely, non-convex Mixed-Integer Nonlinear Programming (MINLP) and non-convex Nonlinear Programming (NLP) problems. The target is to solve MINLP/NLP problems from different domains of research in fewer fitness evaluations as compared to other state-of-the-art stochastic algorithms in the area. The algorithm is GPU compatible and shows remarkable speedups over nVidia's CUDA-compatible GPUs. MINLP problems involve both discrete and continuous variables with several active nonlinear equality and inequality constraints making them extremely difficult to solve. The proposed algorithm is named the adaptive resolution Genetic Algorithm (arGA) as it exploits the concept of controlling the search space size and resolution in an adaptive manner. Using entropy measures, the proposed algorithm adaptively controls the intensity of the genetic search in a given sub-solution space, i.e., promising regions are searched more intensively as compared to other regions. The algorithm is equipped with an asynchronous adaptive local search operator to further improve the performance. Niching is incorporated by using a technique inspired from the Tabu search. The algorithm reduces the chances of convergence to local optima by maintaining a list of already visited optima and penalizing their neighborhoods. The proposed technique was able to find the best-known solutions to extremely difficult MINLP/NLP problems in fewer fitness evaluations and in a competitive amount of time. The results section discusses the performance of the algorithm and the effect of different operators by using a variety of MINLP/NLPs from different problem domains. GPU

A. Munawar · M. Wahib
Graduate School of Information Science & Technology, Hokkaido University, Sapporo, Japan
e-mail: asim@ist.hokudai.ac.jp; wahib@ist.hokudai.ac.jp

M. Munetomo (✉) · K. Akama
Information Initiative Center, Hokkaido University, Sapporo, Japan
e-mail: munetomo@iic.hokudai.ac.jp; akama@iic.hokudai.ac.jp

implementation shows a speedup of up to 42× for single precision and 20× for double precision implementation over the nVidia C2050 GPU architecture.

1 Introduction

Mixed-Integer Nonlinear Programming (MINLP) problems are the most generalized form of single-objective global optimization problems. They contain both continuous and integer decision variables and involve nonlinear objective functions and constraints setting no limit on the complexity of the problems. MINLPs are difficult to solve as [2]:

1. They involve both discrete (integer) and continuous (real) variables.
2. They involve active equality and inequality constraints.
3. Objective function and constraints are nonlinear, generating potential non-convexities.

Many real-world constrained optimization problems are modeled as MINLPs, e.g., heat and mass exchange networks, batch plant design and scheduling, and the design of interplanetary spacecraft trajectories. In a mathematical form, an MINLP problem can be given as

$$\begin{array}{lll}
\text{Minimize} & f(x,y) & x \in \mathbb{N}^{n_{\text{disc}}}, y \in \mathbb{R}^{n_{\text{cont}}}, n_{\text{disc}} \in \mathbb{N}, n_{\text{cont}} \in \mathbb{N} \\
\text{Subject to:} & g_i(x,y) = 0, & i = 1,\ldots,m_{\text{eq}} \in \mathbb{N} \\
 & g_i(x,y) \geq 0, & i = m_{\text{eq}}+1,\ldots,m \in \mathbb{N} \\
 & x_{\text{l}} \leq x \leq x_{\text{u}}, & x_{\text{l}}, x_{\text{u}} \in \mathbb{N} \\
 & y_{\text{l}} \leq y \leq y_{\text{u}}, & y_{\text{l}}, y_{\text{u}} \in \mathbb{R}
\end{array}$$

where $f(x,y)$ is the objective function; x is a vector of n_{disc} discrete variables; y is a vector of n_{cont} continuous variables; m_{eq} and m are the number of equality and total constraints, respectively; and $x_{\text{l}}, x_{\text{u}}, y_{\text{l}}, y_{\text{u}}$ are the lower and upper bounds for the discrete and continuous variables, respectively.

Genetic algorithms (GAs) are population-based search and optimization methods that mimic the process of natural evolution. They fall into the category of stochastic global optimization algorithms. Over recent years GAs have been successfully applied to solve different MINLPs [2, 5, 21]. GAs are easy to implement and are black-box optimizers (BBOs) as they do not require any auxiliary information like continuity or differentiability of functions. They are robust and usually do not get trapped in a local optima. However, like other stochastic methods GAs may need a large number of fitness evaluations because of the combinatorial nature of sampling multidimensional space. Nonetheless, GAs have proven effective for the solution of MINLPs [2, 21].

In this chapter we propose an advanced GA based on an adaptive resolution technique. The algorithm exploits the information hidden in the population to intensify the search in promising regions of solution space while reducing the intensity in other areas. The main motivation behind this research is to develop a method that can solve MINLP/NLP problems in fewer fitness evaluations as compared to the existing algorithms in the area. The algorithm should be generalized enough to solve difficult optimization problems taken from different problem domains in a black-box fashion without any user intervention. This design limitation restricts us from using special operators that are designed for a specific class of problems and are difficult to implement over many-core processors like GPUs.

Most of the existing GAs to solve MINLPs concentrate on a given set of benchmark problems from a particular domain and carry no promise to perform well on a problem from an entirely new domain. We address the problems that are far more difficult than the MINLP problems solved in the relevant literature. Our technique is based on a recursive adaptive resolution micro GA (arGA) with local search (LS). The basic idea is to locate the regions of interest and intensify the genetic search in those areas without revisiting the same areas redundantly. We use the entropy measure of each continuous variable to determine the size of the critical area around a promising solution. The entropy measure is also used to perform an adaptive resolution-based local search. This local search tries to find a better solution in the neighborhoods of an individual using multiple resolutions. In order to avoid revisits to previously visited local optima, we use a technique inspired from the Tabu search. We maintain a finite list of visited local optima and penalize their neighborhoods for a specific number of iterations. This generates a niching effect and encourages the algorithm to search in unexplored areas. We have used the oracle penalty method [19] for constraint handling. The oracle penalty method is an advanced penalty method that depends on a single easily controllable input parameter called Ω.

The algorithm is designed in a way to support modern many-core processors or accelerator chips. In order to demonstrate the ability of the algorithm to run in an SIMD environment, we have implemented it over a GPU that supports the nVidia Fermi architecture. In the results section we discuss both the quality of the algorithm and the significant speedups we get by GPU implementation. Instead of using test problems from a single problem set, we have tried to solve problems from a wide variety of problem domains.

The rest of the chapter is organized as follows: in the next section we will discuss some of the existing GAs for solving MINLPs. In Sect. 3 we explain the proposed arGA and the arLS operators. In Sect. 4 we discuss the GPU implementation of the proposed algorithm. Section 5 gives some results to show the advantage of using the proposed algorithm. In the results section we also discuss the GPU-based implementation of the algorithm and its benefits. We conclude the chapter in Sect. 7 with some guidelines for possible improvements in the algorithm.

2 Related Work

The algorithms for MINLP/NLP problems can be categorized as deterministic and stochastic methods. Deterministic techniques have been extensively used to solve MINLPs. Branch and bound, outer approximation, and extended cutting plan methods are some of the famous deterministic techniques. Grossman [9] gives a detailed review of the deterministic techniques for solving MINLPs. Deterministic methods usually guarantee the global optimality at the expense of long execution times depending on the problem complexity. Deterministic methods are usually not BBOs as they often require the problem to be reduced in a particular form, e.g., removal of non-convexities and initialization of optimizers. This often requires the knowledge about the problem structure.

Stochastic methods based on metaheuristic search techniques are true BBOs as they do not require any information about the mathematical model of the optimization problem. Although such algorithms carry no guarantee for reaching global optimality, due to their robustness and ease of implementation, they are widely used to solve difficult optimization problems, yet their applications in MINLPs remain small. A recent approach on MINLPs by ant colony optimization (ACO) is done by Midaco [18].

Two main concepts of evolution, natural selection and genetic dynamics, inspired the development of GAs. Basic principles of GAs were laid down by J.H. Holland [10] and his colleagues in 1975 and were elaborated in detail by D.E. Goldberg [7]. GAs are flexible and can easily be used with other algorithms in a hybrid fashion [3, 4].

2.1 GAs for Solving MINLP Problems

Simple GA is not able to solve even the easiest MINLP problems. There are two approaches that can be used to enable a GA to solve difficult MINLP problems: The first approach is the use of advanced genetic operators to ensure the desired convergence of the algorithm. The second approach is hybridization of GAs with deterministic or LS methods [3].

A. Ponsich et al. [17] gives some guidelines for GA implementation in batch plant design problems. Batch plant design problem is a real-world problem that is modeled as a non-convex MINLP. Two main issues discussed in this study are the specific encoding methods and efficient constraint handling. The research uses similar encoding for both integer and continuous part and claims the mixed real-discrete encoding method to be the best option. For constraint handling the paper suggests elimination for small problems but appropriate penalization for the complex problems. However, finding appropriate penalization factor is not always easy for the case of MINLPs where the constraints are non-convex and numerous. Another research on the use of GAs for similar problem is given in M. Danish et al. [2]. The paper uses tournament selection, SBX crossover, polynomial mutation, and variable elitism operator along with distance-based dynamic penalty with anti-distortion. The authors claim to solve six difficult MINLP problems by using the proposed method. T. Young et al. [21] suggests an information-guided GA (IGA) approach. It implements the information theory to the mutation stage of the GAs to refresh the premature population. Local search is also performed to increase the efficiency. The paper uses an adaptive penalty scheme to handle constraints [12] and solves five popular benchmark problems using the suggested scheme. V.B. Gantovnik et al. [5] use GAs to solve a problem to design of fiber-reinforced composite shell. The suggested approach tries to reduce the number of fitness and constraint function evaluations by using tree-based data structures for efficient search in the memory to avoid redundant fitness calculations. It suggests the use of multivariate approximation for continuous variables to avoid unnecessary exact analyses for points close to previous values. Apart from the research work done in the area there is at least one commercial product that uses GAs to solve MINLP problems. The product is known as GENO or General Evolutionary Numerical Optimizer [6].

As opposed to the existing techniques, the objective of this research is to solve difficult MINLP problems using techniques involving simple genetic operators that are easy to implement over many-core processors. The operators should not be biased towards a set of problems and should be applicable to problems from a wide variety of problem domains. In the current research, we try to solve some of the extremely difficult problems that to the best of author's knowledge have never attempted before using GAs.

3 The Proposed Algorithm

The proposed approach adopts three guiding principles: (a) areas around better solutions have a greater chance of having an even better solution, (b) constraints must be handled using an advanced penalty method that does not get trapped in a single feasible region, and (c) revisiting local optima results is a waste of time and therefore must be avoided. Using these principles as guidelines, the algorithm uses a hierarchical approach for vigorously searching the promising sub-solution spaces by adaptive resolution GA combined with an adaptive resolution LS operator. In order to eliminate redundancy in revisiting already visited local optima we have used an operator inspired from the working of Tabu search. The oracle penalty method is used to handle constraints.

3.1 Variables Encoding and Genetic Operators

Encoding is one of the most important design factors for GAs, as it limits the kind of genetic operators that can be used by the algorithm. In our approach, we use different encoding for the real and the integer part. In the proposed approach the continuous part of the problem is encoded as real numbers with double precision, while the discrete part is encoded as binary numbers of fixed user-defined length. This kind of encoding allows performing real genetic operators on the continuous variables while binary genetic operators are applied to the discrete part.

We employ simple one-point crossover for the discrete part while SBX crossover [1] is applied to the real part of the chromosome. Binary mutations are achieved by simple bit flipping operation, while polynomial mutation is used for the continuous part of the individuals. Tournament selection of size T is used as a selection operator along with sharing operator that acts as a niching technique to avoid early convergence. Elitist replacement is used for the insertion of new individuals in the population.

3.2 Constraint Handling

Penalty methods are used to handle the problem constraints. Such methods transform a constrained problem to an unconstrained problem by adding the weighted sum of constraint violations to the original fitness function. Death or static penalty methods are the most commonly used penalty methods. Although easy to use, these methods get stuck in a single feasible region and therefore are not able to achieve good performance for tightly constrained problems. We have used oracle penalty method [19] for constraint handling. Oracle method depends on a single parameter, named Ω, which is selected as the best equivalent or just slightly greater than the optimal (feasible) objective function value for a given problem. As for most real-world problems this value is unknown a priori, we start with a value of $\Omega = 1e^6$. We keep on improving the value of Ω by assigning the best known feasible fitness value of the previous run. Mathematically the oracle penalty function can be represented as

$$p(x) = \begin{cases} \alpha \cdot |f(x) - \Omega| + (1-\alpha) \cdot \mathrm{res}(x) & \text{, if } \ f(x) > \Omega \ \text{ or } \ \mathrm{res}(x) > 0 \\ -|f(x) - \Omega| & \text{, if } \ f(x) \le \Omega \ \text{ and } \ \mathrm{res}(x) = 0 \end{cases}$$

where α is given by

$$\alpha = \begin{cases} \frac{|f(x)-\Omega| \cdot \frac{6\sqrt{3}-2}{6\sqrt{3}} - \mathrm{res}(x)}{|f(x)-\Omega| - \mathrm{res}(x)} & \text{, if } \ f(x) > \Omega \ \text{ and } \ \mathrm{res}(x) < \frac{|f(x)-\Omega|}{3} \\ 1 - \frac{1}{2\sqrt{\frac{|f(x)-\Omega|}{\mathrm{res}(x)}}} & \text{, if } \ f(x) > \Omega \ \text{ and } \ \frac{|f(x)-\Omega|}{3} \le \mathrm{res}(x) \le |f(x) - \Omega| \\ \frac{1}{2}\sqrt{\frac{|f(x)-\Omega|}{\mathrm{res}(x)}} & \text{, if } \ f(x) > \Omega \ \text{ and } \ \mathrm{res}(x) > |f(x) - \Omega| \\ 0 & \text{, if } \ f(x) \le \Omega \end{cases}$$

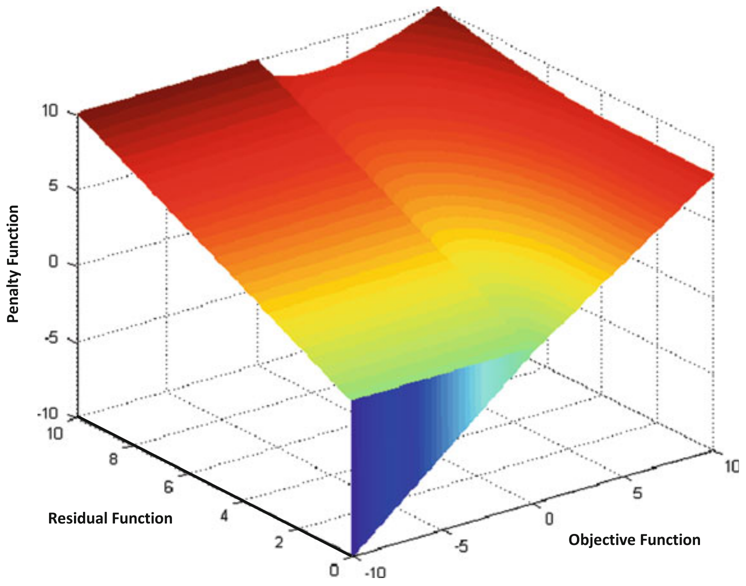


Fig. 1 The oracle penalty function [19]

Shape of the oracle penalty function is shown in Fig. 1. The oracle penalty function is good at dealing with the non-convexities in both equality and inequality constraints.

3.3 Micro GA

The algorithm relies on micro GAs [11] as opposed to the conventional GAs. Micro GAs maintain a very small population ($\lessapprox 20$) that is re-initialized after every few generations (between 10 and 100). We call this re-initialization of the population a restart. A restart can inherit some information from the previous run in order to improve the performance. This re-initialization of the population every few generations can be considered as a mutation operator. The re-initialization of the population may or may not be completely random. We define a proximity parameter that determines the proximity of the newly initialized individuals to the best known individual of the previous restart. Conventional GAs maintain a large population with sizes of approximately $(1/k) \cdot 2^k$ [8] for binary encoding, where k is the average size of the schema of interest (effectively the average number of bits per parameter, i.e., approximately equal to $nchrome/nparam$, rounded to the nearest integer) [8]. The large population size ensures with high probability that the required genetic material is present in the initial population. In our observation, for extremely complex MINLP/NLPs, a large population slows down the overall process as a convergence to local optima may lead the whole population to converge to this point, while in the case of micro GAs, a small population may converge to a local optima but the next restart will have a good chance of jumping to another area. Moreover, as the population size is very small, this process happens very quickly. Therefore, even though the micro GAs require many restarts before they succeed to achieve all the required genetic information to reach the best or near best solution, the process is much faster than their conventional counterparts. Individuals carried on from the previous run allow the building blocks of two different restarts to mix with each other, a step required by the schema theorem of GAs. In the results section we discuss the effects of population sizing on the solution quality.

3.4 Adaptive Resolution Approach

Adaptive resolution is a recursive approach to divide the solution space in search for a better solution. The recursion terminates when no better solution is found. The size of the sub-solution space is calculated by using the entropy value of each variable. The recursive nature of arGA is shown in Fig. 2. The probability of adaptive resolution is proportional to the fitness of the individuals, i.e., an individual with a good overall fitness has a greater chance of getting selected for the adaptive resolution search.

3.4.1 Using Entropy to Control Resolution

In order to control the size of the sub-solution space a vector of real numbers γ is defined; γ is the same size as the number of continuous variables. γ value for each variable is calculated using the information entropy. According to Shannon's definition of information entropy [20], for a variable V which can randomly take a value v from a set $\mathbb{V}$, the information entropy of the set $\mathbb{V}$ is

$$E(V) = -\sum_{v \in \mathrm{V}} p(v) \ln p(v)$$

If V can only take a narrow range of values, $p(v)$ for these values is $\approxeq$ 1. For other values of V, $p(v)$ is close to zero. Therefore, $E(V)$ will be close to zero. In contrast, if V can take many different values each time with a small $p(v)$, $E(V)$ will be close to 1.

Measuring entropy using the above equation is simple for discrete variables, but the entropy must be redefined for real numbers by discretizing the range of each variable. If we have $i = 0, \ldots, I$ real variables with lower and upper bound (L_i, U_i) such that $L_i \leq V_i \leq U_i$. For each variable V_i, we divide the solution space

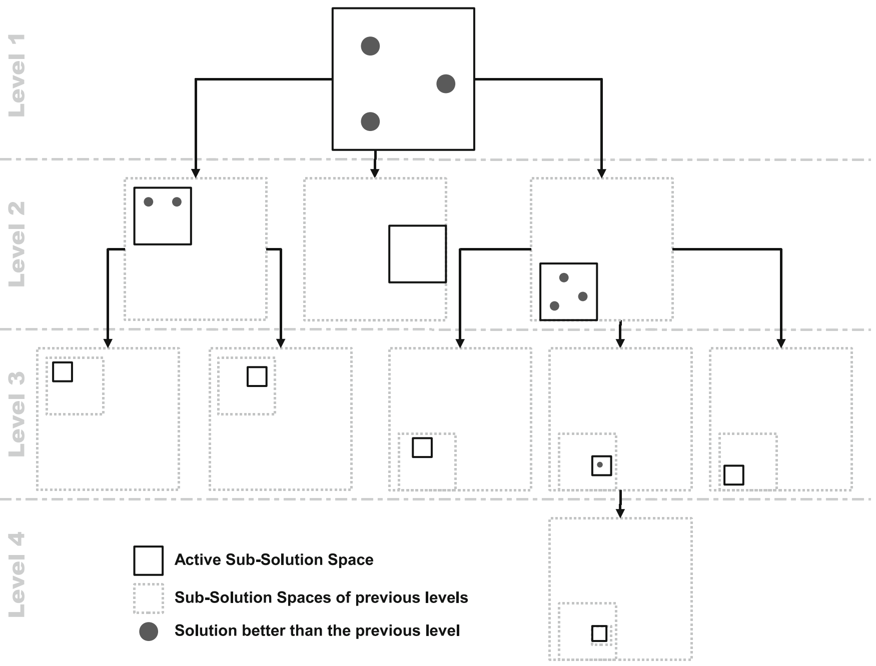

Level 1
Level 2
Level 3
Level 4
Active Sub-Solution Space
Sub-Solution Spaces of previous levels
Solution better than the previous level

Fig. 2 Recursive behavior of adaptive resolution technique

into R sections of equal size. Let $S = \{s_{r,i} | i = 1, \ldots, I, r = 1, \ldots, R\}$ and $s_{r,i} = [L_i^r, U_i^r]$,
where

$$L_i^r = L_i + \frac{r-1}{R}(U_i - L_i) \qquad U_i^r = U_i - \frac{R-r}{R}(U_i - L_i)$$

for $i = 1, \ldots, I$ and $r = 1, \ldots, R$. The probability that the variable V_i takes the value in subspace $s_{r,i} = [L_i^r, U_i^r]$ is given by $P_{r,i} = P(V_i = v_i | v_i \in s_{r,i})$. The total entropy of the set V_i is

$$E(V_i) = -\sum_{r=1}^{R} P_{r,i} \log(P_{r,i})$$

So, for a variable V_i we define $\gamma_i = E(V_i)$. Furthermore, in each iteration of arGA, the value of γ for each variable is halved, hence reducing the size of sub-solution space to half. This technique is similar to the well-known bisection-based reduction technique.

Algorithm 1 Adaptive resolution local search algorithm

```
inputs
    $X = [X_{bin}, X_{real}]$ {$X_{bin}$ is binary vector of size $N_{bin}$, $X_{real}$ is real vector of size $N_{real}$}
    $\gamma$ {A real number vector of size $N_{real}$. contains the resolution for each real variable}
    $N_{LS}$, $N_{rLS}$ {number of local search iterations, number iterations for each real variable}
-----------------------------------------------------------------
for $i = 1$ to $N_{LS}$
    nextEval = INFINITY;
    ------------- binary local search -------------
    for $j = 1$ to $N_{bin}$
        $\neg X_{bin}[j]$;
        if nextEval > objFunc(X) then    nextNode ← X,    nextEval ← objFunc(X);
        $\neg X_{bin}[j]$;
    end for
    -------------- real local search --------------
    for $j = 1$ to $N_{real}$
        for $k = 1$ to $N_{rLS}$
            $X_{real}[j] -= k \cdot \gamma[j]$;
            if nextEval > objFunc(X) then    nextNode ← X,    nextEval ← objFunc(X);
            $X_{real}[j] += 2 \cdot k \cdot \gamma[j]$;
            if nextEval > objFunc(X) then    nextNode ← X,    nextEval ← objFunc(X);
            $X_{real}[j] -= k \cdot \gamma[j]$;
        end for
    end for
    if objFunc(X) > nextEval then    X ← nextNode;    else    break;
end for
```

3.5 Local Search

We have used an asynchronous local search to improve the efficiency of the algorithm. LS is applied only to a specified percentage of the individuals. This probability of local search is kept low (0.01–0.1). The proposed local search is adaptive resolution version of the widely used hill-climbing algorithm. The algorithm for the local search used in arGA is shown in Algorithm 1. The effect of this asynchronous operator is discussed in the results section.

3.6 Avoiding Redundancy

Avoiding redundant search near already visited local optima is a key to better and faster search. We have used a simple technique inspired from Tabu search to

Algorithm 2 Algorithm for avoiding redundancy

```
Initialize counter
loop until the maximum number of restarts is achieved
    Run GA
    if no better solution is found in the current restart
        Increment the counter
        if counter exceeds the maximum number of similar runs ($M_r$)
            Apply arGA to the neighborhood to make sure that the point is the local optima
        if a better solution is found by the arGA
            Reset counter and continue the loop
        else
            Insert the solution into a finite size Queue
    end If
end loop
```

avoid this redundancy. The algorithm used for avoiding redundancy is shown in Algorithm 2. Whenever a promising individual is located the algorithm intensifies the search around that individual in order to find an even better solution. This is done by calculating the entropy and then selecting a sub-solution space around that individual based on the entropy measures. When the algorithm reaches the best solution in this small sub-solution space, it is assumed with high probability that the individual found is the best individual in that sub-solution space. Algorithm maintains a finite list of such visited local optima. The neighborhoods of these individuals are penalized in every fitness evaluation. As the list is of finite size, the individual ultimately gets removed from the list and gets a second chance of being searched by the algorithm.

4 arGA over GPU

As is clear from Fig. 3, the arGA algorithm involves genetic search within a genetic search. This makes the algorithm slow and therefore a good candidate for implementation over a GPU. In order to simplify the parallelization we will parallelize only the most time-consuming parts of the algorithms. Empirical results show that arGA and the arLS operators are the most time-consuming parts of the proposed algorithms. Therefore, we will concentrate only on the parallelization of these two operators. Our approach is to have two different kernels, one for handling the local search, while the other is responsible for the arGA operator. We need to incorporate minor modifications to make the algorithm suitable for the GPU implementation. We separate a number of individuals on which we want to run arLS and/or arGA. When the number of these individuals reaches a certain amount,

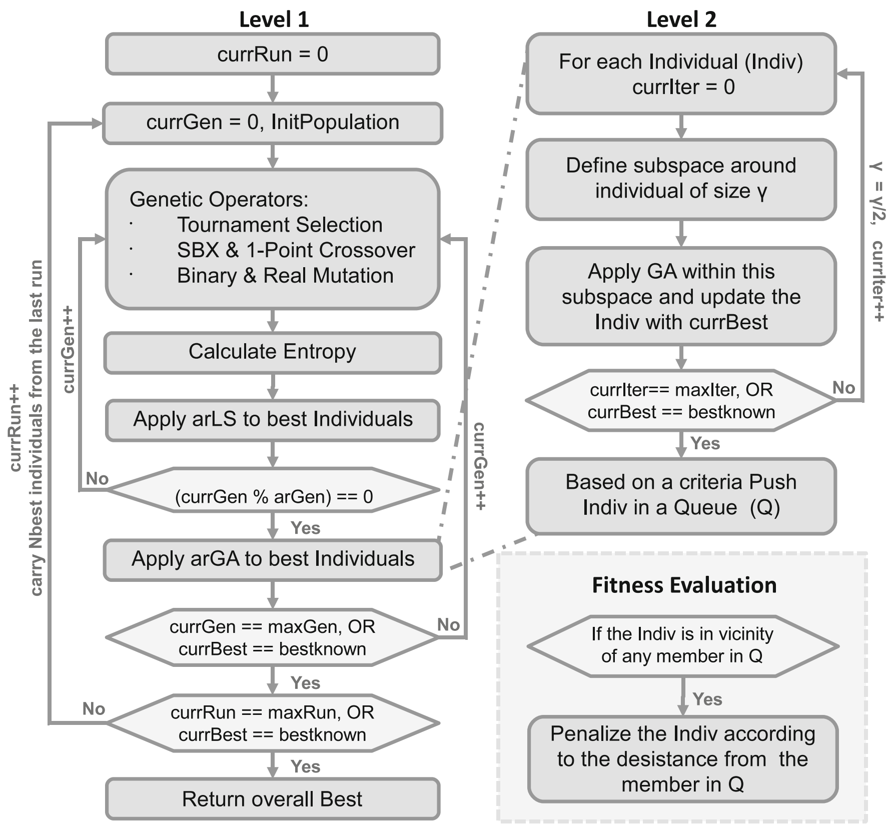
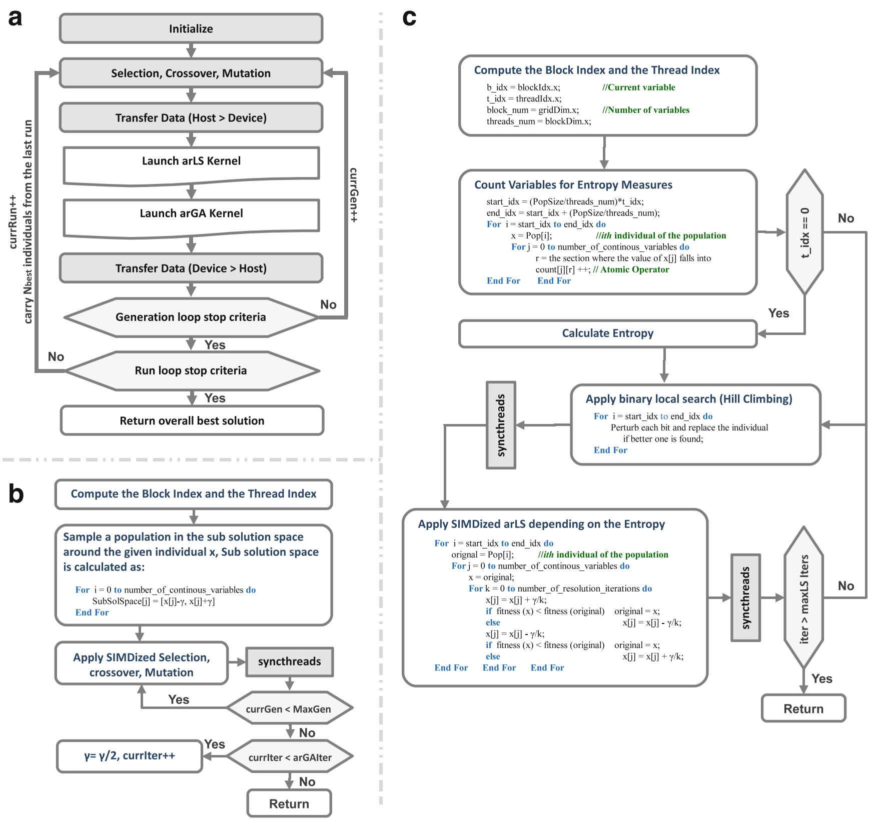

Level 1
currRun = 0
currGen = 0, InitPopulation
Genetic Operators:
· Tournament Selection
· SBX & 1-Point Crossover
· Binary & Real Mutation
Calculate Entropy
Apply arLS to best Individuals
(currGen % arGen) == 0
No
Yes
Apply arGA to best Individuals
currGen == maxGen, OR currBest == bestknown
No
Yes
currRun == maxRun, OR currBest == bestknown
No
Yes
Return overall Best
currRun++ carry Nbest individuals from the last run
currGen++
currGen++
Level 2
For each Individual (Indiv) currIter = 0
Define subspace around individual of size γ
Apply GA within this subspace and update the Indiv with currBest
currIter== maxIter, OR currBest == bestknown
No
Yes
γ = γ/2, currIter++
Based on a criteria Push Indiv in a Queue (Q)
Fitness Evaluation
If the Indiv is in vicinity of any member in Q
Yes
Penalize the Indiv according to the desistance from the member in Q

Fig. 3 Adaptive resolution Genetic Algorithm (arGA)

we call the kernels to perform the operations on the individuals. Only the required individuals are transferred between the host and device memory.

4.1 arLS and arGA Kernels

The local search is based on a feedback system developed by the authors in Munawar et al. [15] to make it more suitable for the GPU implementation. We concentrated on increasing the occupancy of both the GPU and the host processor while keeping the data transfer between host and device to a minimum level. The flow diagram of the host side controller running over the CPU and the two kernels is shown in Fig. 4. As the figures shows, a lot of consideration has been given to the SIMDization part of the implementation. Except for few conditional branching, the


a
Initialize
Selection, Crossover, Mutation
Transfer Data (Host > Device)
Launch arLS Kernel
Launch arGA Kernel
Transfer Data (Device > Host)
Generation loop stop criteria
No
currGen++
Yes
Run loop stop criteria
No
currRun++
carry Nbest individuals from the last run
Yes
Return overall best solution
b
Compute the Block Index and the Thread Index
Sample a population in the sub solution space around the given individual x, Sub solution space is calculated as:
Apply SIMDized Selection, crossover, Mutation
syncthreads
currGen < MaxGen
Yes
No
currIter < arGAIter
Yes
γ= γ/2, currIter++
No
Return
c
Compute the Block Index and the Thread Index
Count Variables for Entropy Measures
t_idx == 0
No
Yes
Calculate Entropy
Apply binary local search (Hill Climbing)
syncthreads
Apply SIMDized arLS depending on the Entropy
syncthreads
iter > maxLS Iters
No
Yes
Return

```
For  i = 0 to number_of_continous_variables do
      SubSolSpace[j] = [x[j]-γ, x[j]+γ]
End For
```

```
b_idx = blockIdx.x;               //Current variable
t_idx = threadIdx.x;
block_num = gridDim.x;            //Number of variables
threads_num = blockDim.x;
```

```
start_idx = (PopSize/threads_num)*t_idx;
end_idx = start_idx + (PopSize/threads_num);
For  i = start_idx to end_idx do
        x = Pop[i];                    //ith individual of the population
        For j = 0 to number_of_continous_variables do
                r = the section where the value of x[j] falls into
                count[j][r] ++; // Atomic Operator
End For      End For
```

```
For  i = start_idx to end_idx do
      Perturb each bit and replace the individual
          if better one is found;
End For
```

```
For  i = start_idx to end_idx do
      orignal = Pop[i];          //ith individual of the population
      For j = 0 to number_of_continous_variables do
            x = original;
            For k = 0 to number_of_resolution_iterations do
                  x[j] = x[j] + γ/k;
                  if  fitness (x) < fitness (original)    original = x;
                  else                                      x[j] = x[j] - γ/k;
                  x[j] = x[j] - γ/k;
                  if  fitness (x) < fitness (original)    original = x;
                  else                                      x[j] = x[j] + γ/k;
End For     End For     End For
```

Fig. 4 Implementation of arGA + arLS algorithm over GPU. (**a**) Host side controller. (**b**) arGA kernel performs adaptive resolution genetic search. (**c**) arLS kernel calculates entropy and performs adaptive resolution local search

threads in a warp follow same instruction pattern. This allows the algorithm to take maximum advantage of the low-lying hardware.

The arLS kernel is responsible to compute the entropy measure of each real dimension before moving on to the local search. The binary local search and the real local search are then applied to the individuals. The entropy is used by the real local search operator to search a better solution in a decreasingly smaller neighborhood. In case of arGA kernel SIMDized mutation, crossover and selection operators are used. These operators use XOR and some other SIMD friendly techniques discussed in Munawar et al. [15].

4.2 Random Number Generator Kernel

Current graphics hardware does not provide the function for generating random numbers. Fortunately, CUDA SDK comes with a sample random number generator (RNG) [16] based on Mersenne twister proposed by M. Matsumoto et al. [13]. Mersenne twister has properties like long period, efficient use of memory, good distribution properties, and high performance. With some modifications proposed by M. Matsumoto [14], the algorithm maps well onto the CUDA programming model to produce uniformly distributed pseudorandom numbers. Our strategy is to calculate the total number of random numbers required by the arLS and arGA kernel and precompute them by calling an RNG kernel. RNG kernel is not shown in Fig. 4 in order to reduce the complexity of the figure.

5 Results

Results given in this section were collected over a system with Intel Core$^{\text{TM}}$i7 920@ 2.67 GHz 4 GB memory as the host CPU. We are using nVidia Tesla C2050 GPU mounted on the host for results of GPU implementation. C2050 have 3 GB of device memory; the total number of CUDA cores is 448. The GPU has 64 KB L1 cache + shared memory/32 cores and 768 KB L2 unified cache. The clock rate is 1.15 GHz. Unlike its predecessors nVidia Tesla C2050 supports half speed IEEE 754 double precision operations. The compute capability of the device is 2.0. We are using Fedora Core 10 (×86_64) as the operating system and CUDA SDK/Toolkit ver. 2.3 with nVidia driver ver. 260.19.36. Other tools used for optimization and profiling include *CudaVisualProfiler* and *CudaOccupancyCalculator*. C2050 is dedicated to computations only. System has a separate GeForce 8400 GS GPU acting as a display card.

The algorithm is controlled and configured by various input parameters. All the parameters are preconfigured to an appropriate value. However, an advanced user can modify the parameters by accessing the parameter's input file. The nomenclature of the parameters is given below:

G, P, R	Maximum number of allowed generations, population, and restarts
nC, nD, L_c	Number of continuous, discrete variables and chromosome length $(nC + nD)$
m_{eq}, m	Number of equality constraints and total number of active constraints
L_i, U_i	Lower and upper bound for the ith continuous variable, respectively
l_j, u_j, b_j	Lower and upper bound and bits required for the jth discrete variable
P, P_c	Penalty function and penalty configuration
P_c, P_{mb}, P_{mr}	Crossover probability, binary mutation and real mutation probability
P_{ls}, N_{ls}	Local search probability and number of LS iterations
T	Tournament size for the selection operator
Pr	Proximity parameter for sampling of population in consecutive runs
η_c, η_m	Crossover and mutation probability distribution index
Ω	Initial value of the oracle for oracle penalty method

Table 1 Benchmarking problems

Prob	m_{eq}	m	nC	nD	B_f	Type	Category
1	0	2	1	1	2	Minimization	Process synthesis problem
2	1	2	2	1	2.124	Minimization	Process synthesis and design
2*	0	1	1	1	2.124	Minimization	Process synthesis problem
3	0	5	2	1	1.07654	Minimization	Process flow sheeting problem
4	6	14	3	2	99.245	Minimization	Two-reactor problem
4*	0	4	2	1	99.245	Minimization	Two-reactor problem
5	0	9	3	4	3.557473	Minimization	Process synthesis problem
6	0	3	-3	2	32,217.4	Maximization	Process design problem
7	2	23	9	8	67.998	Minimization	Process design problem
8	0	6	5	0	-30,665.54	Minimization	Process design problem
9	0	4	6	0	4.9307	Minimization	Space mission trajectory design
10	0	8	6	0	1,581,950	Maximization	Space mission trajectory design
11	0	2	12	0	18.19	Minimization	Space mission trajectory design

Note: 2* and 4* are the reformulated versions of problem 2 and 4 respectively

I_a	Number of levels for performing adaptive resolution GA
I_r	Number of iterations of adaptive resolution local search for continuous variables
St	Stopping criteria
M_f, M_t	Maximum allowed fitness evaluations and execution time
M_G, M_R	Maximum allowed generations and runs without any improvement in fitness
Q	Size of the queue for the Tabu list
N	Size of the neighborhood that needs to be penalized
R_e	Number of partitions to discretize the continuous range for entropy calculations
L_i^r, U_i^r	Lower and upper range of the rth partition of the ith variable
E	Allowed error between the calculated and the best known result
E_p	Residual accuracy

With so many parameters for the algorithm calibration, it is vital to carefully study the effect of each and every important parameter on the total execution time and solution quality. The preset values of the parameters used for the above experimentation are as follows: $G = 10$, $P = 10$, $R = 30000$, $P_c = 0.7$, $P_{mb} = 0.06$, $P_{mr} = 0.3$, $P_{ls} = 0.01$, $\Omega = 1e6$, $\eta_c = 2$, $\eta_m = 100$, $P_r = 1e2$, $M_G = 6$, $M_R = 30$, $I_a = I_r = 8$, $E_p = 0.01$, $E = 1\,\%$, $R_e = 8$, $Q = 30$, $P =$ "oracle penalty", St $=$ "best solution found or no improvement", $N_n = 1$, and $T = 3$. The values of these parameters are selected empirically by studying the effect of the different parameters on the output.

We consider MINLP/NLPs from a wide variety of problem domains. The problems vary in difficulty levels starting from simple to extremely difficult problems. The benchmarking problems used in the results section are shown in Table 1. As it is clear from the table the benchmarks include several kinds of problems from chemical batch processing and problems from space mission trajectory design problems. The table below explains the abbreviations used in Table 1.

Abbreviation	Explanation
Prob	Problem number
m_{eq}	Number of equality constraints
m	Total number of constraints
nC	Number of continuous variables
nD	Number of discrete variables
B_{f}	Best known fitness
Type	Is the problem minimization problem or maximization
Category	The kind of problem

The rest of the section is divided into two sub-sections. The first subsection discusses the efficiency of the algorithm over different optimization problems, and the next subsection talks about the gains of implementing the algorithm over a CUDA-compatible GPU.

5.1 Algorithm's Efficiency

Results obtained by applying the proposed algorithm on the MINLP/NLP benchmark problems can be found in Table 2. Note that all the results are an average of 30 independent runs under identical circumstances. The table below explains the abbreviations used in Table 2.

Abbreviation	Explanation
Problem	Problem number
Restarts$_{\mathrm{mean}}$	Average number of restarts
Eval$_{\mathrm{mean}}$	Average number of evaluations over all runs with a feasible solution
Feasible	Number of feasible solutions found out of 30 test runs
Optimal	Number of optimal solutions found out of 30 test runs
f_{best}	Best (feasible) objective function value found out of 30 test runs
f_{worst}	Worst (feasible) objective function value found out of 30 test runs
f_{mean}	Average objective function value over all runs with a feasible solution
Time$_{\mathrm{mean}}$	Average CPU time over all runs with a feasible solution

Figure 5 shows the relationship between the average number of evaluations and average number of restarts vs. the generation/population (G, P) pair. Even though the number of average restarts decreased with the increase of G and P, the total number of fitness evaluations increased by a significant amount. As the total number of fitness evaluations is directly related to the total execution time, larger values of G and P results in longer execution time. The results provide a solid ground for the use of micro GAs instead of conventional GAs.

Figure 6 shows the importance of optimizing the probability of LS P_{ls}. It is clear that $P_{\mathrm{ls}} = 0.01$ is the optimal value as it results in the maximum number of optimal solutions found. Keeping the value of $P_{\mathrm{ls}} \geq 0.1$ forces the algorithm towards local optima.

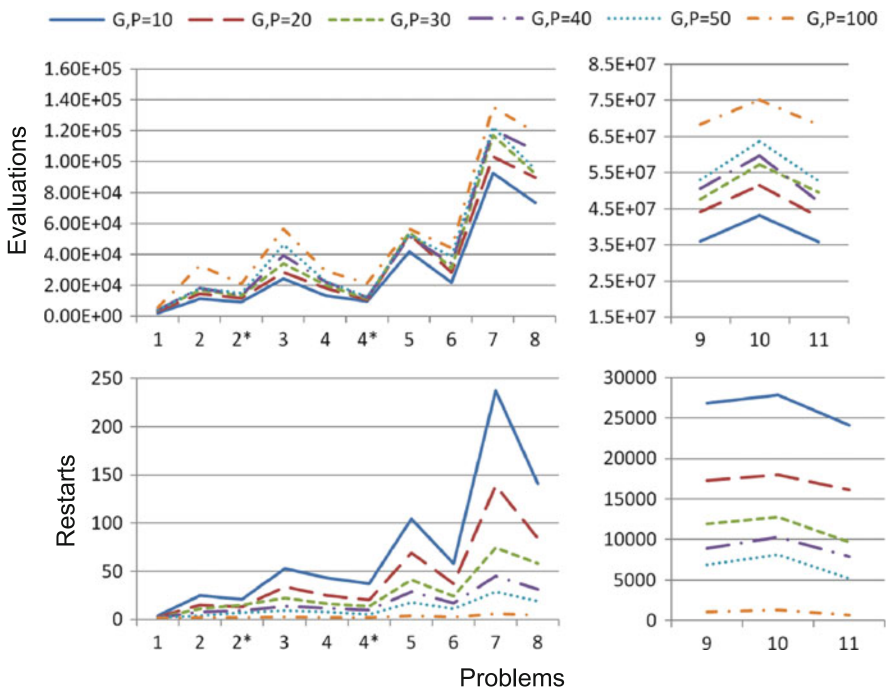


Fig. 5 Effect of generation/population (G, P) pair on the total number of evaluations and average number of restarts

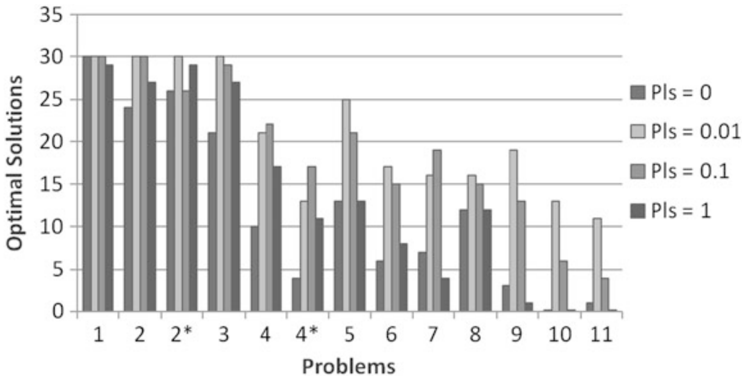


Fig. 6 Effect of local search probability (P_{ls}) on the number of optimal solutions found

Figure 7 shows the effect of arGA iterations on the total number of optimal solutions found in 30 runs. arGA is a kind of nondeterministic local search algorithm. The figure depicts the importance of the arGA step.

Figure 8 shows the efficiency of the queue-based approach to avoid redundancy in searching. The figure shows that for $Q = 0$ the algorithm is not able to find any

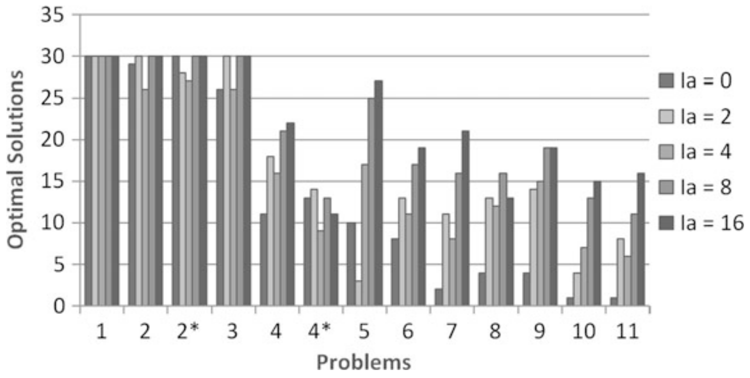


Fig. 7 Effect of arGA iterations (I_a) on the number of optimal solutions found

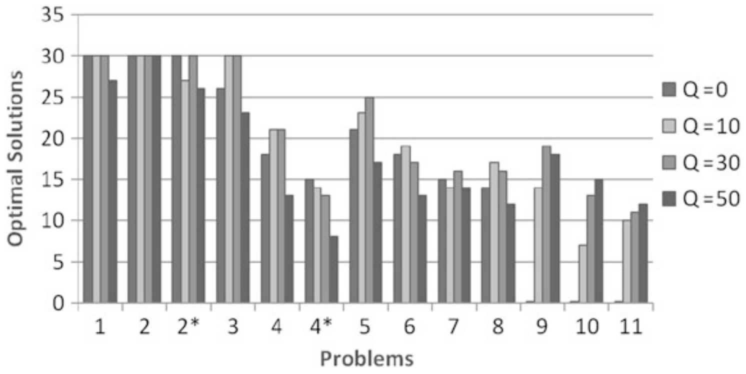


Fig. 8 Effect of queue size (Q) on the number of optimal solutions found

optimal result for some problems, while for other difficult optimization problems, the performance remains low. Hence, the quality of the results is tightly related with the value of Q.

6 GPU Implementation

We have tested the GPU implementation of the proposed algorithm using the same problems used above. As we can see from the Table 1, the problems have different difficulty levels depending on the number of variables, number of constraints, and the internal details of the problem. Some problems are MINLP as they contain both real and discrete parts, while others are NLP problems with no discrete part.

In order to compare our algorithm over GPU with other modern processors we have implemented it both in a serial and parallel fashion over general purpose

Table 2 Results obtained by applying the proposed algorithm on different optimization problems

Problem	$\text{Restarts}_{\text{mean}}$	$\text{Eval}_{\text{mean}}$	Feasible	Optimal	f_{best}	f_{worst}	f_{mean}	$\text{Time}_{\text{mean}}$
1	4	1,976	30	30	2	2	2	0.21 ± 0.034
2	25	11,301	30	30	2.124	2.124	2.124	1.38 ± 0.269
2*	21	9,210	30	30	2.124	2.124	2.124	0.85 ± 0.089
3	53	24,253	30	30	1.07654	1.07654	1.07654	2.87 ± 0.252
4	43	14,381	30	21	99.245	104.37	100.921	1.48 ± 0.258
4*	37	9,586	30	13	99.245	108.408	101.311	1.02 ± 0.092
5	104	41,626	30	25	3.557473	3.636	3.559	5.92 ± 1.002
6	58	26,092	30	17	32,217.4	31,393.1	32,194.7	3.21 ± 0.700
7	237	92,646	30	16	67.998	71.214	68.347	10.86±1.517
8	141	73,581	30	16	-30,665.54	-30,683.29	-30,667.13	8.55± 0.428
9	26,821	36.04e6	29	19	4.93	5.93	5.01	70m36s ± 40s
10	27,871	43.25e6	30	13	1,581,950	1,473,261	1,544,508	84m23s± 64s
11	24,092	35.91e6	18	11	18.19	27.57	20.00	68m32± 44s

Note: 2* and 4* are the reformulated versions of problem 2 and 4 respectively

commodity processors. Five different implementations used to obtain the results are shown in the table below:

Abbreviation	Explanation
S_{duo}	Serial implementation over Intel Core$^{\text{TM}}$2 Duo E8600@ 3.33 GHz CPU (2 cores/2 threads) with 4 GB of memory
P_{duo}	OpenMP-based parallel implementation over Intel Core $^{\text{TM}}$2 Duo E8600 @ 3.33 GHz CPU (2 cores/2 threads) with 4 GB of memory
P_{i7}	OpenMP-based parallel implementation over Intel Core$^{\text{TM}}$i7 920 @ 2.67 GHz CPU (4 cores/8 threads) with 4 GB memory
D_{gpu}	Double precision implementation on nVidia C2050 GPU
S_{gpu}	Single precision implementation on nVidia C2050 GPU

Table 3 shows the average execution time required and the standard deviation for solving the set of benchmarking problems given in Table 1 over different kinds of parallel architectures. It is clear from the table that we get a significant reduction in the execution time when we implement the algorithm over the GPU in single or double precision. This speedup as compared with the serial implementation over Intel Core$^{\text{TM}}$2 Duo E8600 processor is shown in Fig. 9. We can see that the speedups vary according to the difficulty of the problems. Problem number 9 being the most difficult requires maximum number of fitness evaluations and therefore yields over 40$\times$ speedup for single precision and around 20$\times$ speedup for double precision implementation.

However, measuring the speedups and execution time is not enough to show a successful implementation over the GPU. In Table 4 we have shown the total number of optimal solutions found out of 30 independent runs. Moreover, we also show the average of total fitness evaluations required by each implementation. This table provides us with significant results. It shows that the quality and convergence properties of the algorithm are not affected by the changes done during the parallelization process.

Table 3 Average execution time required for solving the set of benchmark problem over different architectures

Prob	Average execution time (sec) ± standard deviation				
	S_{duo}	P_{duo}	P_{i7}	D_{gpu}	S_{gpu}
1	0.21 ± 0.034	0.112 ± 0.001	0.050 ± 0.003	0.152 ± 0.004	0.072 ± 0.005
2	1.38 ± 0.269	0.676 ± 0.063	0.260 ± 0.028	0.195 ± 0.033	0.097 ± 0.008
2*	0.85 ± 0.089	0.391 ± 0.098	0.158 ± 0.023	0.146 ± 0.028	0.070 ± 0.003
3	2.87 ± 0.252	1.530 ± 0.233	0.556 ± 0.184	0.306 ± 0.019	0.143 ± 0.021
4	1.48 ± 0.258	0.768 ± 0.071	0.300 ± 0.119	0.180 ± 0.008	0.080 ± 0.002
4*	1.02 ± 0.092	0.533 ± 0.156	0.218 ± 0.024	0.170 ± 0.010	0.083 ± 0.004
5	5.92 ± 1.002	3.021 ± 0.203	1.235 ± 0.195	0.486 ± 0.068	0.226 ± 0.045
6	3.21 ± 0.700	1.648 ± 0.185	0.722 ± 0.050	0.308 ± 0.072	0.151 ± 0.081
7	10.86±1.517	5.575 ± 0.794	2.442 ± 0.282	0.781 ± 0.103	0.370 ± 0.082
8	8.55± 0.428	4.401 ± 0.819	1.910 ± 0.057	0.638 ± 0.075	0.306 ± 0.078
9	70m36s ± 40s	36m2s ± 32s	14m38s ± 25s	3m33s ± 13s	1m41s ± 6.9s

Note: 2* and 4* are the reformulated versions of problem 2 and 4 respectively

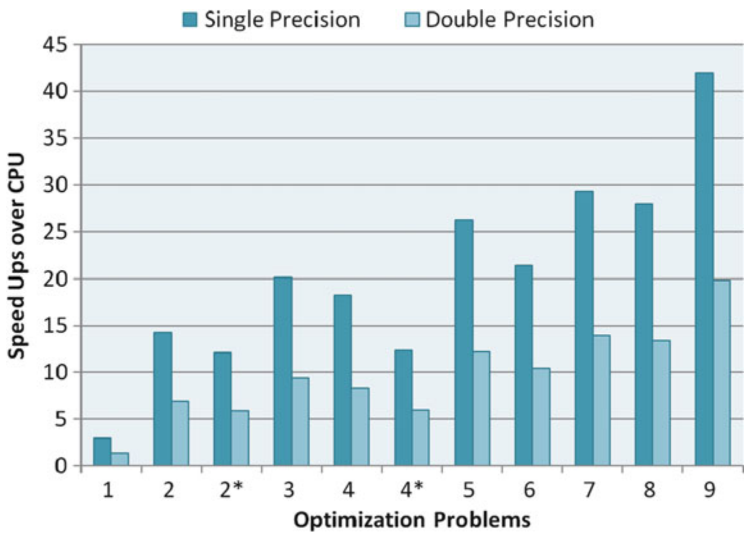


Fig. 9 Speedups achieved in single and double precision implementation over nVidia Fermi GPU as compared with a serial implementation over Intel Core2Duo processor

7 Conclusions and Future Work

The proposed method exploits some simple concepts in optimization to construct an algorithm that is well suited for the many-core processors like modern GPUs and is robust enough to solve some of the extremely difficult MINLP/NLP problems. Adaptive resolution technique combined with micro GAs is able to find good local optima in a very rough multidimensional terrain. This technique combined with an operator inspired by the Tabu list in Tabu search does the trick. Local optima are

Table 4 Total number of optimal solutions found and the average of total fitness evaluations

Prob	Optimal solutions					Average fitness evaluation				
	S_{duo}	P_{duo}	P_{i7}	D_{gpu}	S_{gpu}	S_{duo}	P_{duo}	P_{i7}	D_{gpu}	S_{gpu}
1	30	30	30	30	30	1,976	1,959	1,961	2,489	2,053
2	30	30	28	29	30	11,301	11,337	11,268	11,618	11,203
2*	30	30	30	30	30	9,210	9,254	9,297	10,380	9,893
3	30	30	29	30	30	24,253	24,331	24,209	25,592	24,848
4	21	22	23	21	22	14,381	14,301	14,348	14,991	14,401
4*	13	14	13	14	15	9,586	9,548	9,492	10,264	9,708
5	25	24	23	24	25	41,626	41,590	41,602	43,114	41,965
6	17	16	17	15	17	26,092	26,087	26,118	27,806	26,040
7	16	15	17	15	15	92,646	92,651	92,641	92,851	92,698
8	16	16	15	17	19	73,581	73,553	73,562	75,924	74,058
9	19	20	18	17	18	36.04e6	36.04e6	36.04e6	36.04e6	36.04e6

Note: 2* and 4* are the reformulated versions of problem 2 and 4 respectively

stored in a list for a specific number of generations in order to avoid redundant searching in already searched areas. Even though the algorithm depends on many different operators and input parameters, one of the most important operators is the restarting of the algorithm whenever stuck. This might sound simple but micro GA with many restarts produces much better results than a normal GA with larger population and greater number of generations.

The algorithm presented in this chapter was able to solve extremely difficult MINLP/NLP problems from various domains of research in a reasonable amount of time. GPU implementation shows a considerable improvement over serial execution. We show here a speedup of 42× for single precision and over 20× speedup for double precision accuracy.

As future work it would be interesting to calibrate (optimize) the input parameters using the same algorithm. The parameter tuning in this case will become a very complicated MINLP. The fitness function in this case would be the cumulative performance of the algorithm over a set of benchmarking MINLP/NLP problems.

Acknowledgement This work is supported by Grant-in-Aid for Scientific Research (C) No. 22500196 by MEXT, Japan.

References

1. Agrawal, R.B., Deb, K.: Simulated binary crossover for continuous search space. Department of Mechanical Engineering, Indian Institute of Technology, Kanpur, UP, India (1994)
2. Danish, M., Kumar, S., Qamareen, A., Kumar, S.: Optimal solution of MINLP problems using modified genetic algorithm. Chem. Prod. Process Model. **1**(1) (2006)
3. El-mihoub, T.A., Hopgood, A.A., Nolle, L., Battersby, A.: Hybrid genetic algorithms: A review. Eng. Lett. **13**(12), 124–137 (2006)
4. French, A.P., Robinson, A.C., Wilson, J.M.: Using a hybrid genetic-algorithm/branch and bound approach to solve feasibility and optimization integer programming problems. J. Heuristics **7**(6), 551–564 (2001)

5. Gantovnik, V.B., Gurdal, Z., Watson, L.T., Anderson-Cook, C.M.: A genetic algorithm for mixed integer nonlinear programming problems using separate constraint approximations. Departmental Technical Report TR-03-22, Computer Science, Virginia Polytechnic Institute and State University (2005)
6. GENO: General evolutionary numerical optimizer. http://tomopt.com/tomlab/products/geno/
7. Goldberg, D.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional, Upper Saddle River (1989)
8. Goldberg, D., Deb, K., Clark, J.: Genetic algorithms, noise, and the sizing of populations. Complex Syst. **6**, 333–362 (1991)
9. Grossmann, I.E.: Review of nonlinear mixed-integer and disjunctive programming techniques. Optim. Eng. **3**, 227–252 (2002)
10. Holland, J.: Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor (1975)
11. Krishnakumar, K.: Micro-genetic algorithms for stationary and non-stationary function optimization. SPIE Intell. Control Adapt. Syst. **1196**, 289–296 (1989)
12. Lemonge, A.C., Barbosa, H.J.: An adaptive penalty scheme for genetic algorithms in structural optimization. Int. J. Numer. Methods Eng. **59**(5), 703–736 (2004)
13. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. **8**(1), 3–30 (1998)
14. Matsumoto, M., Nishimura, T.: Dynamic creation of pseudorandom number generators. In: Monte Carlo and Quasi-Monte Carlo Methods 1998. Springer, Berlin (2000)
15. Munawar, A., Wahib, M., Munetomo, M., Akama, K.: Hybrid of genetic algorithm and local search to solve MAX-SAT problem using NVIDIA CUDA framework. Genet. Program. Evol. Mach. **10**(4), 391–415 (2009)
16. Podlozhnyuk, V.: Parallel mersenne twister, CUDA 2.1 SDK Documentation (June 2007)
17. Ponsich, A., Azzaro-Pantel, C., Domenech, S., Pibouleau, L.: Some guidelines for genetic algorithm implementation in MINLP batch plant design problems. In: Advances in Metaheuristics for Hard Optimization. Natural Computing Series [ISSN 1619-7127]. Springer, Berlin (2008)
18. Schlueter, M.: Midaco: Global optimization software for mixed integer nonlinear programming. http://www.midaco-solver.com (2009)
19. Schlueter, M., Gerdts, M.: The oracle penalty method. J. Glob. Optim. **47**(2), 293–325 (2010)
20. Shannon, C.: A mathematical theory of communication. Bell Syst. Tech. J. **27**, 3–55 (2001)
21. Young, C., Zheng, Y., Yeh, C., Jang, S.: Information-guided genetic algorithm approach to the solution of MINLP problems. Ind. Eng. Chem. Res. **46**(5), 1527–1537 (2007)

An Analytical Study of Parallel GA with Independent Runs on GPUs

Shigeyoshi Tsutsui and Noriyuki Fujimoto

Abstract This chapter proposes a genetic algorithm for solving QAPs with parallel independent run using GPU computation and gives a statistical analysis on how speedup can be attained with this model. With the proposed model, we achieve a GPU computation performance that is nearly proportional to the number of equipped multiprocessors (MPs) in the GPUs. We explain these computational results by performing statistical analysis. Regarding performance comparison to CPU computations, GPU computation shows a speedup of 7.2× and 13.1× on average using a single GTX 285 GPU and two GTX 285 GPUs, respectively. The parallel independent run model is the simplest of the various parallel evolutionary computation models, and among the models it demonstrates the lower limit performance.

1 Introduction

In evolutionary computation, many applications consume their run time for evaluation of individuals. Thus, in evolutionary computation, the approach in which populations are managed by CPUs and evaluation tasks are assigned to GPUs is a good approach though some overhead time for the transfer of individuals between the CPU and GPU is needed. A typical example of this approach is reported by Maitre et al. [7], in which they showed a successful speedup of 60× in solving a real-world material-science problem.

S. Tsutsui (✉)
Hannan University, 5-4-33, Amamihigashi, Matsubara, Osaka 580-8502, Japan
e-mail: tsutsui@hannan-u.ac.jp

N. Fujimoto
Osaka Prefecture University, 1-1, Gakuen-Cho, Naka-ku, Sakai, Osaka 599-8531, Japan
e-mail: fujimoto@mi.s.osakafu-u.ac.jp

On the other hand, there are applications where the evolutionary algorithm itself must be performed fast. Typical examples of such applications are combinatorial optimization problems such as routing problems and assignment problems. In this class of problems, computation time for the evaluation of individuals is not dominant. In this case, it is useful that evolutionary operations are performed on GPUs using parallel evolutionary models, such as fine-grained or coarse-grained models [3, 5]. However with this approach the speedup is at most 10×.

In a previous study [15], we applied GPU computation to solve quadratic assignment problems (QAPs) with parallel evolutionary computation using a coarse-grained model on a single GPU. The results in that study showed that parallel evolutionary computation with the NVIDIA GeForce GTX285 GPU produces a speedup of 3–12× compared to the Intel Core i7 965 (3.2 GHz). However, the analysis of the results was postponed for future work.

In this chapter, we propose a simplified parallel evolutionary model with independent runs and analyze how the speedup is obtained using a statistical model of parallel runs of the algorithm. The basic idea of the parallel independent runs is as follows. A set of small-size subpopulations is run in parallel in each block in CUDA [10]. The computation time to get an acceptable solution is different in each block because the computations are performed probabilistically using pseudorandom numbers with different seeds. If one subpopulation finds an acceptable solution first, then we stop executions of all subpopulations. This time is shorter than the average time where we run all subpopulation until the acceptable solution is obtained. The parallel independent run model is the simplest of the various parallel evolutionary computation models, and among the models it demonstrates the lower limit performance.

In the remainder of this chapter, Sect. 2 describes a brief review of GPU computation and its application to evolutionary computation. Then, the parallel GA model with independent runs to solve QAPs on GPUs is described in Sect. 3. Section 4 describes computational results and gives analysis. Finally, Sect. 5 concludes the chapter.

2 A Brief Review of GPU Computation

In terms of hardware, CUDA GPUs are regarded as two-level shared-memory machines as shown in Fig. 1. Processors in a CUDA GPU are grouped into multiprocessors (MPs). Each MP consists of eight thread processors (TPs). TPs in a MP exchange data via fast 16 KB shared memory. On the other hand, data exchange between MPs is performed via VRAM. VRAM is also like main memory for processors. So, code and data in a CUDA program are basically stored in VRAM. Although processors have no data cache for VRAM (except for the constant and texture memory areas), the shared memory can be used as manually controlled data cache.

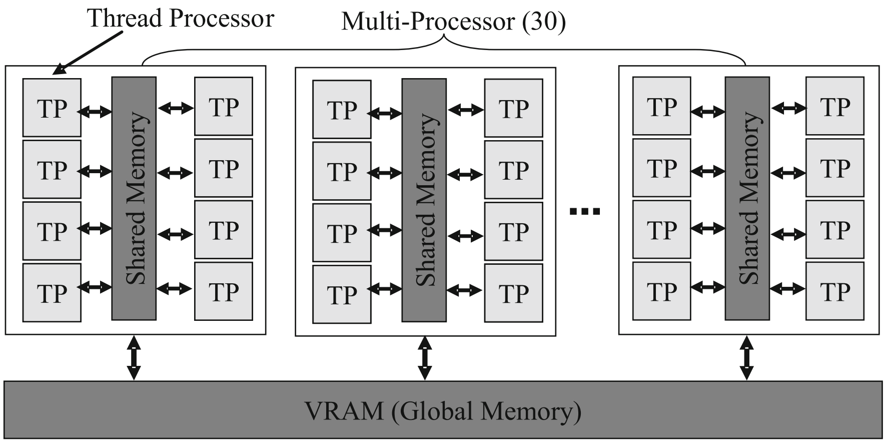


Fig. 1 CUDA architecture as a two-level shared-memory machine (GTX 285)

The CUDA programming model is a multi-threaded programming model. In a CUDA program, threads form two hierarchies: a *grid* and *thread blocks*. A thread block is a set of threads. A thread block has a dimensionality of 1, 2, or 3. A grid is a set of blocks with the same size and dimensionality. A grid has dimensionality of 1 or 2. Each thread executes the same code specified by the *kernel function*. A kernel function call generates threads as a grid with given dimensionality and size.

In GTX 285, VRAM bandwidth is very high at 159 GB/s. However, when accessing VRAM, there is memory latency as large as 100–150 arithmetic operations [10]. This VRAM latency can be hidden if there are a sufficient number of threads by overlapping memory access of a thread with computation of other threads. The number of threads that run concurrently on a multiprocessor is restricted by the fact that shared memory and registers in a MP are divided among concurrent thread blocks allocated for the MP. VRAM has a read-only region of size 64 KB [10]. The region is called the *constant memory*. The constant memory space is cached. The cache working set for constant memory is 8 KB per MP [10].

3 Parallel GA Model with Independent Runs to Solve QAPs on GPU

Here, we propose a parallel independent evolutionary model to solve QAPs.

3.1 Quadratic Assignment Problem

In QAP, we are given L locations and L facilities, and the task is to obtain an assignment ϕ which minimizes cost as defined in the following equation.

$$\mathrm{cost}(\phi) = \sum_{i=0}^{L-1} \sum_{j=0}^{L-1} f_{ij}\, d_{\phi(i)\phi(j)}, \tag{1}$$

where d_{ij} is the distance matrix which represents distance between locations i and j and f_{ij} is the flow matrix which represents flow between facilities i and j. The QAP is an NP-hard optimization problem [12], and it is considered one of the hardest optimization problems.

3.2 Evolutionary Model for GPU Computation for QAP

3.2.1 The Base GA Model for QAP

The base evolutionary model for QAP in GPU computation is the same as was used in our previous study [15]. Figure 2 shows the base evolutionary model for QAP in this research. Let N be the population size. We use two pools P and W of size N. P is the population pool to store individuals of the current population, and W is a working pool to store newly generated offspring individuals until they are selected to update P for the next generation. The algorithm is performed as follows:

Step 1 Set generation counter $t \leftarrow 0$ and initialize P.
Step 2 Evaluate each individual in P.
Step 3 For each individual I_i in P, select its partner I_j $(j \neq i)$ randomly. Then apply a crossover to the pair $(I_i\,, I_j)$ and generate one child, I_i', in position i in W.
Step 4 For each I_i', apply a mutation with probability p_{m}.
Step 5 Evaluate each individual in W.
Step 6 For each i, compare the costs of I_i and I_i'. If I_i' is the winner, then replace I_i with I_i'.
Step 7 Increment generation counter $t \leftarrow t + 1$.
Step 8 If the termination criteria are met, terminate the algorithm. Otherwise, go to Step 3.

Usually, a crossover operator generates two offspring from two parents. However, in this model we generate only one child from two parents. In Step 6, the comparison of costs is performed like a tournament selection with size 2. However, each comparison is performed between individuals I_i and I_i' which have the same index i. Please remember here that I_i' is generated from I_i as one of its parents (the other parent I_j was chosen randomly). Since a parent and a child have partly similar substrings, this comparison scheme can be expected to maintain population diversity like the deterministic crowding proposed by Mahfoud [6].

From the viewpoint of parallelization, this model also has advantage because a process for one individual in a generation cycle can be implemented in a parallel thread easily. Using one child from two parents was already proposed

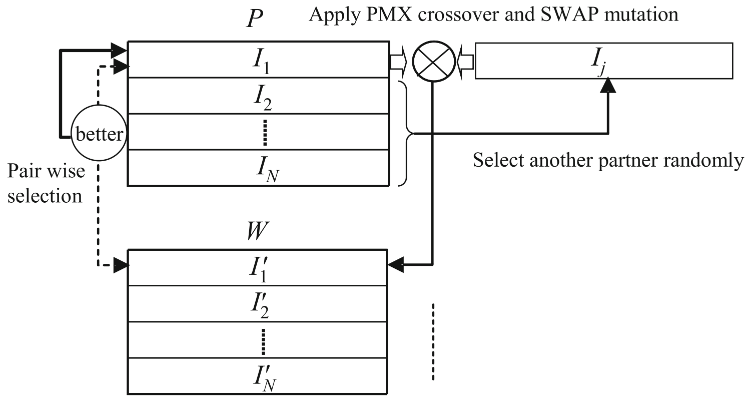

P
Apply PMX crossover and SWAP mutation
I_1
I_2
I_N
I_j
better
Pair wise selection
Select another partner randomly
W
I'_1
I'_2
I'_N

Fig. 2 The base evolutionary mode for QAP

for designing the well-known GENITOR algorithm by Whitley et al. [17]. For crossover operators, we performed preliminary tests using two well-known operators, i.e., the order crossover (OX) [11] and the partially matched crossover (PMX) [4], by implementing the GA model both on CPU and GPU. The results showed the PMX operator worked much better than the OX operator on QAPs. Thus, in the experiment in IV, we will use PMX operator.

For mutation operator, we used a *swap mutation* where values of two randomly chosen positions in a string are exchanged. Strings to which the mutation are applied are probabilistically determined with a mutation rate of p_{m}. Applying local search in solving QAP is very common in evolutionary algorithms [8, 13, 14]. One of the main purpose of this research is to analyze the effect of the parallel evolutionary computation on GPUs, we will not apply any local search in this study.

3.2.2 Parallel Evolutionary Model for GPU Computation with Independent Runs

The NVIDIA GeForce GTX285 GPU which we use in this study has 30 MPs, and each MP has 8 TPs sharing 16 KB high-speed shared memory among them. For each subpopulation, we use the base evolutionary model described in Sect. 3.2.1. So, we allocate the population pools P and W described in Fig. 2 to the shared memory of each MP.

We represent an individual as an array of type *unsigned char*, rather than type *int*. This restricts the problem size we can solve to at most 255. However, this does not immediately interfere with solving QAP because QAP is fairly difficult even if the problem size is relatively small, and maximum problem sizes of QAP are at most 150. Let L be the problem size of a given QAP instance. Then, the size of each individual is L bytes, and the size of P and W is $2L \times N$ where N is the number of individuals allocated to each MP at the same time, i.e., subpopulation size. We have only 16 KB shared memory per MP. To maximize both L and N, we chose N as 128 under the assumption that L is at most 56. Consequently, for W and P, our implementation consumes $2L \times N = 2 \times 56 \times 128 = 14336$ bytes of the shared memory at most. From the CUDA programming scheme, we generate p blocks, and each block consists of 128 threads.

We stored distance matrix d_{ij} and flow matrix f_{ij} in the constant memory space so that they can be accessed via cache. To save the memory space size for these matrices, *unsigned short* was used for the elements of these matrices. We implemented a simple random number generator for each thread each with a different seed number.

In our previous study [15], individuals in each block are exchanged among blocks every 500-generation interval as follows: (1) in the host machine, all individuals are shuffled. Then, they are sent to the VRAM of the GPU. (2) Each MP selects a block not yet processed and copies the corresponding individuals from VRAM to its shared memory, performs the generational process up to 500 generations, and finally copies the evolved individuals from its shared memory to VRAM. (3) The above process is repeated until all blocks are processed. (4) Then, all individuals are copied back to the memory of its host machine and merged. (5) These processes are repeated until termination criteria are satisfied.

In this study, we evolve subpopulations of each block independently without exchanging individuals among blocks. If a subpopulation in a block has found an acceptable solution, then it sets "FoundFlag" to 1 (the initial value being set to 0) in VRAM which can be shared by all blocks, sends the solution to the VRAM, and then terminates the execution. All subpopulations in other blocks check the flag at every generation whether the flag is set or not. If they find the flag is set by another block, then they terminate their executions; otherwise, they continue. In this way, with this parallel independent run model, execution of the algorithm terminates if one of the subpopulations in the blocks finds an acceptable solution.

The subpopulation size of 128 is relatively small for solving QAP, and independent runs with this population size often cause stagnation of evolution in the subpopulations. To prevent the stagnation in the subpopulation, we introduce a *restart strategy* as described in the following subsection.

3.2.3 Implementing Restart Strategy

Restart strategies have been discussed elsewhere. The delta coding method by Mathias and Whitley [9] is an iterative genetic search strategy that sustains search by periodically reinitializing the population. It also remaps the search hyperspace with each iteration, and it is reported that it shows good performance especially when used with Gray coding. The CHC method by Eshelman is a safe search

Table 1 Effect of restart strategy

QAP instances	No restart				Restart			
	#OPT	T_{avg}	Min	Max	#OPT	T_{avg}	Min	Max
tai25b	6	0.07	0.05	0.10	30	0.07	0.05	3.77
kra30a	1	0.27	0.27	0.27	30	10.03	0.27	61.13
kra30b	0	–	–	–	30	25.20	0.45	92.24
tai30b	0	–	–	–	30	3.96	0.39	12.53
kra32	0	–	–	–	30	10.70	0.33	59.52
tai35b	0	–	–	–	30	29.69	2.53	137.72
ste36b	0	–	–	–	30	17.00	0.73	45.10
tai40b	1	0.30	0.30	0.30	30	6.96	0.30	20.96
tai50b	0	–	–	–	30	48.07	1.12	147.37

#OPT number of success run in 30 runs, T_{avg} : average time to find acceptable solutions in success runs in second.
Results of this table were obtained by emulating a single block run using a CPU

strategy where restart of the search process is done if it gets stuck at local optima by reinitializing the population with individuals generated by mutating the best solution obtained so far (also keeping the best one) [2].

In another study, Tsutsui et al. proposed a search space division and restart mechanism to avoid getting stuck in local traps in the framework referred to as *the bi-population GA scheme (bGA)* [16]. Although the use of a restart strategy is a feature of the bGA, its main purpose is to maintain a suitable balance between exploration and exploitation during the search process by means of two populations.

The approach taken in this study is similar to CHC. Let the best functional value in a subpopulation be represented by $f_{\text{c-best}}$. In each generation, we count the number of individuals whose functional values are equal to $f_{\text{c-best}}$ and represent the number by B_{count}. If B_{count} is larger than or equal to $N \times B_{\text{rate}}$, then we assume the subpopulation is trapped at local solutions and the subpopulation is reinitialized. According to a preliminary study, we found that the approach of keeping the best individual during reinitialization causes ill effect. So, in the reinitialization, all member of the subpopulation are generated anew. We used $B_{\text{rate}} = 0.6$.

In a block, each individual is processed as an independent thread, so to get B_{count}, we need to use instructions which control exclusive and synchronous executions. In this study, we used *atomic instructions* which work on the shared memory. These instructions are available in a GPU with compute capability 1.2 or later. Since this counting takes some additional overhead time, the checking current B_{count} is performed only every 50 generations. Figure 3 shows the pseudo code for each block.

Table 1 shows the effect of the restart strategy. Results of this table were obtained by emulating a single block run using a CPU (Intel Core i7965 3.2 GHz). We used a mutation rate of $p_{\text{m}} = 0.1$. From this table, we can see a clear effect as a result of using the restart strategy.

```
//Code for GPU
__global__ void kernel()
{
        int threadID = POOL_SIZE * blockIdx.x + threadIdx.x;
        get seed of random number generator for threadID from VRAM;
        t=0;
        initialize string with the seed;
        evaluate the string;
        for(int t=1; t<MaxGeneration; t++){
                __syncthreads();
                reset memory contents related restart;
                Evolutionary Code for the thread threadID;
                ...

                if (this thread (individual) found an acceptable solution){
                      set FoundFlag in VRAM to 1 using "atomicCAS" instruction;
                      if(this thread has set)
                              write the solution (string) to VRAM;
                }
                __syncthreads();
                if(FoundFlag in VRAM ==1)
                      break; //an acceptable solution has found
                if(t%50==0){
                        atomicMin(fc_best, perf);//get minimum value at s_minValue in SM
                          __syncthreads();
                          if(perf == *fc_best)//if this thread have minimum functional value
                                atomicAdd(B_count, 1);//add 1 to B_count in SM
                                  __syncthreads();
                                  if(threadIdx.x==0){//check restart condition by thread with id 0
                                        if(*B_count > (int)(POOL_SIZE*B_rate+0.5))//satisfy?
                                                *restart_flag = 1;
                                                //inform other thread that the restart condition has satisfied
                                  }
                                  __syncthreads();
                                  if(*restart_flag == 1){//restart?
                                        initialize string of this thread;
                                        evaluate;
                                        __syncthreads();
                                  }
                          }
                          else
                                __syncthreads();
                }
        }
}
//main function for CPU
int main()
{
          ....
        data copy from main memory to VRAM
        //kernel call
        dim3 grid(128,1);
        dim3 block(P, 1, 1);
        kernel<<<grid, block, SMsize >>>();
        copy from main memory to VRAM
           ....
`
```

Fig. 3 Pseudo code for parallel independent run in CUDA

4 Computational Results and Analysis

4.1 Experimental Conditions

In this study, we used a PC which has one Intel Core i7 965 (3.2 GHz) processor and two NVIDIA GeForce GTX285 GPUs. The OS was Windows XP Professional with NVIDIA graphics driver Version 195.62. For CUDA program compilation, Microsoft Visual Studio 2008 Professional Edition with optimization option (O2) and CUDA 2.3 SDK were used. The two GPUs were controlled using thread programming provided in the Win32 API.

The instances on which we tested our algorithm were taken from the QAPLIB benchmark library at [1]. QAP instances in the QAPLIB can be classified into four classes: (i) randomly generated instances, (ii) grid-based distance matrix, (iii) real-life instances, and (iv) real-life-like instances [13]. In this experiment, we used the following nine instances which were classified as either (iii) or (iv) with the problem size ranging from 25 to 50; tai25b, kra30a, kra30b, tai30b, kra32, tai35b, ste36b, tai40, and tai50b.

Thirty runs were performed for each instance. We measured the performance by the average time to find acceptable solutions as measured by CPU (wall clock). Acceptable solutions for all instances except tai50b were set to known optimal values. For tai50b, we set the acceptable solution be within 0.06 % of the known optimal solution. We represent the average time over 30 runs as $T_{p,\text{avg}}$ where p is the number of MPs. We used a mutation rate of $p_\text{m} = 0.1$ in all experiments.

4.2 Distribution of Computation Time on a Single Block

Here we performed 100 runs on one GPU using a single block until acceptable solutions were obtained. Figure 4 shows the distribution of the run time until an acceptable solution was obtained for each instance. Each asterisk indicates the time in each run, and black boxes indicate their average time ($T_{1,\text{avg}}$). Please refer to Table 2 for each value of $T_{1,\text{avg}}$ and its standard deviation. From Fig. 4, if we run the algorithm using $p(p > 1)$ blocks simultaneously, it is clear that the average runtime is reduced, i.e., $T_{p,\text{avg}} < T_{1,\text{avg}}$ for $p > 1$.

4.3 Numerical Results

Since our system has two GPUs, we performed two experiments: one was to use a single GPU with 30 blocks ($p = 30$), and the other was to use two GPUs with 60 blocks ($p = 60$). In the CUDA programming model, it is possible to run our algorithm using many blocks such that the number of blocks is more

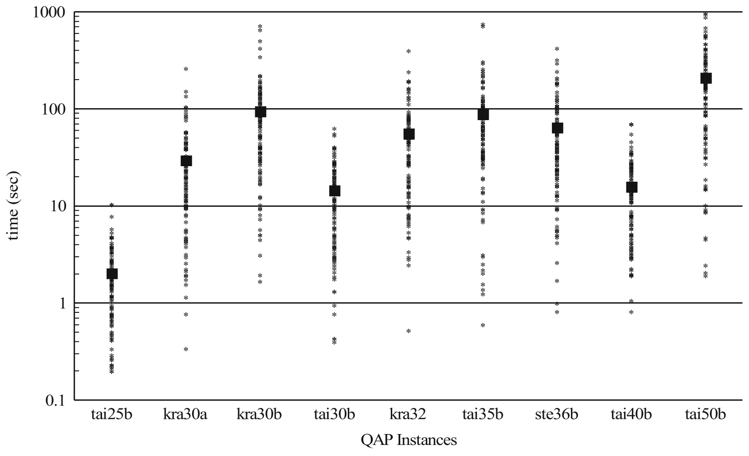


Fig. 4 Distribution of time to find acceptable solution on a single block runs. Squares in *black color* indicate mean times

Table 2 Results of parallel independent runs and statistical estimation

GPU	Instances	tai25b					kra30a					kra30b				
	No of blocks p	GPU			Γ		GPU			Γ		GPU			Γ	
		$T_{p,\mathrm{avg}}$	σ	gain$_p$	$M(P)$	Δ_p	$T_{p,\mathrm{avg}}$	σ	gain$_p$	$M(P)$	Δ_p	$T_{p,\mathrm{avg}}$	σ	gain$_p$	$M(P)$	Δ_p
GPU×1	1	2.02	1.89	1.00	–	–	34.25	36.79	1.00	–	–	113.69	113.36	1.00	–	–
	30	0.21	0.06	9.56	0.03	0.18	1.35	1.23	25.43	0.79	0.56	3.17	2.52	35.85	2.92	0.25
GPU×2	60	0.19	0.02	10.85	0.00	0.18	0.70	0.47	49.10	0.34	0.36	1.63	1.48	69.59	1.21	0.42

GPU	Instances	tai30b					kra32					tai35b				
	No of blocks p	GPU			Γ		GPU			Γ		GPU			Γ	
		$T_{p,\mathrm{avg}}$	σ	gain$_p$	$M(P)$	Δ_p	$T_{p,\mathrm{avg}}$	σ	gain$_p$	$M(P)$	Δ_p	$T_{p,\mathrm{avg}}$	σ	gain$_p$	$M(P)$	Δ_p
GPU×1	1	14.31	13.64	1.00	–	–	56.18	61.11	1.00	–	–	92.08	75.90	1.00	–	–
	30	0.71	0.46	20.24	0.45	0.25	2.13	1.80	26.35	1.78	0.35	3.67	3.56	25.12	3.25	0.41
GPU×2	60	0.46	0.16	30.88	0.16	0.30	1.12	0.93	50.11	0.83	0.29	1.65	1.38	55.68	1.66	−0.01

GPU	Instances	ste36b					tai40b					tai50b				
	No of blocks p	GPU			Γ		GPU			Γ		GPU			Γ	
		$T_{p,\mathrm{avg}}$	σ	gain$_p$	$M(P)$	Δ_p	$T_{p,\mathrm{avg}}$	σ	gain$_p$	$M(P)$	Δ_p	$T_{p,\mathrm{avg}}$	σ	gain$_p$	$M(P)$	Δ_p
GPU×1	1	70.82	73.23	1.00	–	–	19.07	16.43	1.00	–	–	212.55	203.64	1.00	–	–
	30	2.57	1.89	27.54	1.63	0.94	1.15	0.62	16.56	0.46	0.69	8.75	8.65	24.29	6.19	2.56
GPU×2	60	1.35	0.77	52.40	0.66	0.70	0.90	0.16	21.27	0.12	0.78	4.28	3.84	49.62	2.72	1.56

σ : standard deviation, Γ : results with Gamma distribution estimation
Values of $T_{p,\mathrm{avg}}$ and $M(P)$ are in second

than the number of equipped MPs. However, doing this could cause side effects in performance in our parallel evolutionary model due to hardware resource limits. The results are summarized in Table 2.

In the table, gain_p indicates $T_{1,\text{avg}}/T_{p,\text{avg}}$, the run time gain obtained by p-block parallel runs to single block runs (please note that the single block runs are taken from the experiment in Sect. 4.2 and are an average over 100 runs). For example, on tai25b, $\text{gain}_{30} = 9.56$ and $\text{gain}_{60} = 10.85$, respectively, showing not so high a gain. However, on kra30a, $\text{gain}_{30} = 25.43$ and $\text{gain}_{60} = 49.10$, respectively. The values of gain_p are different from instance to instance; they are in the range [10, 35] for $p = 30$ and [10, 70] for $p = 60$ and are nearly proportional to p, except for instances tai25b and tai40b. In these two instances, the gains are smaller. Thus, with some exception, we can clearly confirm the effectiveness of the parallel independent run model.

4.4 Analytical Estimation of Speed with Parallel Independent Runs

In this subsection, we analyze the results in Sect. 4.3 from a statistical perspective. Let the probability density function of the run time on a single block be represented by $f(t)$ and probability distribution function of $f(t)$ by $F(t)$, where

$$F(t) = \int_0^t f(t)dt. \tag{2}$$

Here, consider a situation where we run p blocks in parallel independently, and if a block finds an acceptable solution while other blocks are still running, we stop the GPU computation immediately as described in Sect. 3.2.2. Let the probability density function of the runtime on p blocks be represented by $g(p, t)$, probability distribution function of $g(p, t)$ by $G(p, t)$, and the mean time by $M(p)$. Then, $g(p, t)$ can be simply obtained as follows. Since the probability that all p blocks cannot find acceptable solutions at t is $(1 - F(t))^p$, the probability distribution function $G(p, t)$ can be obtained as

$$G(p,t) = 1 - (1 - F(t))^p, \tag{3}$$

and its probability density function $g(t)$ is obtained as

$$g(p,t) = \frac{dG(p,t)}{dt}, s \tag{4}$$

and $M(p)$ is

$$M(p) = \int_0^\infty t \cdot g(p,t)dt. \tag{5}$$

We estimated the distributions in Fig. 4 by the normal distribution and the gamma distribution using the least-square method. Here, The probability density function $f(t)$ of the gamma distribution can be expressed as follows:

$$f(t) = \frac{1}{\theta^k}\frac{1}{\Gamma(k)}t^{k-1}e^{-\frac{t}{\theta}}, \tag{6}$$

where k and θ are parameters of gamma distribution and $\Gamma(k)$ is the gamma function. Results showed that the gamma distribution of (6) reflected the distributions well as shown in Fig. 5. As we can see in the distributions in Fig. 4, the distributions are not symmetrical against mean values, i.e., they distribute tighter in smaller t (please note the vertical axis is log scale). This is the reason that the normal distribution does not reflect the distributions.

By applying (5) to the estimated distributions in Fig. 5, we calculated $M(p)$ for $p = 30$ and 60. These results are shown in Table 2. Seeing these results, we can see that they reflect the results obtained by GPU computation ($T_{p,\mathrm{avg}}$). However, the values of $M(p)$ are slightly smaller than the corresponding $T_{p,\mathrm{avg}}$ with the exception of tai35b ($p = 60$). We showed the differences of both values by Δ_p. Although the Δ_p values are different among instances, we can see larger size instances have larger Δ_p values than smaller size instances.

Now consider here why these differences arise. In our estimation in (3), we assumed that there is no additional overhead time even if we run p ($p > 1$) blocks in parallel. Detailed hardware performance information from the manufacturer is not open to us. But we can recognize that some additional overhead time must arise if we run multiple blocks simultaneously. One possible scenario we can consider is overhead caused by constant memory access conflict. In our implementation, QAP data (distance matrix and flow matrix, please see Sect. 3.1) are stored in the constant memory. The constant memory has 8 KB cache memory for each MP, but some access conflict occurs when data are transferred from constant memory to cache memory.

Although there exist some differences (Δ_p) between real GPU computation and our statistical analysis values, this analysis does represent the theoretical elements of speedup by the parallel independent run.

4.5 Comparison with CPU Computation

To compare the results of GPU computation and CPU computation, we measured the CPU computation time of the same tasks. The CPU was an Intel Core i7 965 (3.2 GHz) processor running Windows XP Professional. For CPU computation, we used the same evolutionary model as described in Fig. 2. Other conditions, such as the population size, crossover operator, mutation operator, and restart strategy, are also the same. A single population was used on the CPU.

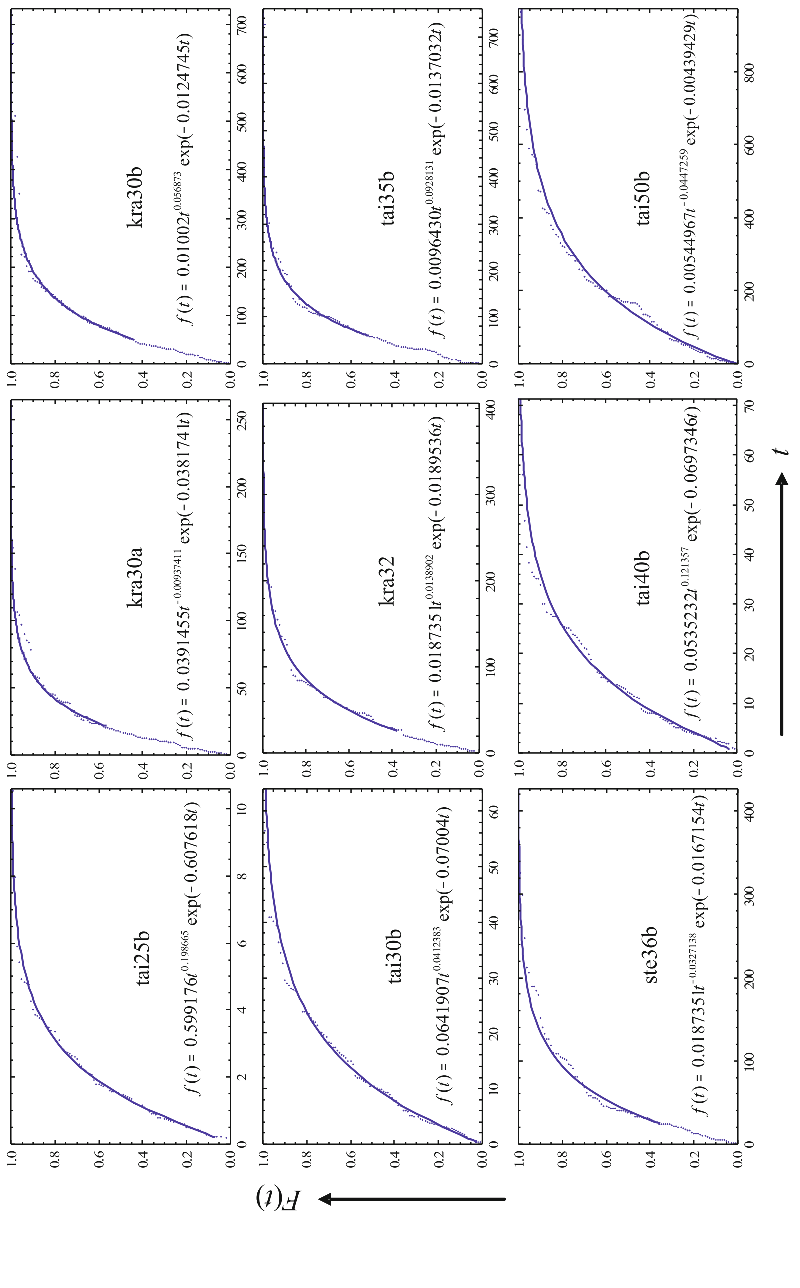


Fig. 5 Estimation of distribution of run time to find acceptable solution on a single block run by gamma distribution

Table 3 Comparison between GPU computation and CPU computation with a single thread

QAP instances	GPU (s)		CPU (s)	speedup	
	$T_{30,\text{avg}}$ (GPU×1)	$T_{60,\text{avg}}$ (GPU×2)	T_{avg}	GPU×1	GPU×2
tai25b	0.21	0.19	0.82	3.9	4.4
kra30a	1.35	0.70	21.45	15.9	30.7
kra30b	3.17	1.63	25.20	7.9	15.4
tai30b	0.71	0.46	3.96	5.6	8.5
kra32	2.13	1.12	10.70	5.0	9.5
tai35b	3.67	1.65	29.64	8.1	17.9
ste36b	2.57	1.35	16.99	6.6	12.6
tai40b	1.15	0.90	6.98	6.1	7.8
tai50b	8.75	4.28	48.07	5.5	11.2
Average	–	–	–	7.2	13.1

```
//cut1 are cut2 are random number in [0,L-1]. cut1 < cut2 is assumed.
//unsigned char *parent1, *parent2 are strings of the parents.
//unsinged char *child is a new string to be generated.
for (int j = 0; j <L; j++)
   child[j] = parent1[j];
for(int i = cut1; i < cut2 ; i++){
   for(int j = 0 ; j < L; j++){
      if (parent2[i] == child[j]){
         unsigned char tmp = child[i]; child[i] = child[j]; child[j] =tmp;
         break;
      }
   }
}
```

Fig. 6 Code of the PMX operator for each thread

Results are summarized in Table 3. On tai25b, the speedup values of the GPU are 3.9× and 4.4× compared against CPU computation, using a single GPU and two GPUs, respectively. These results are not so promising. On kra30 however, the speedup values of the GPU are 15.9× and 30.7× compared against CPU computation, using a single GPU and two GPUs, respectively, showing promising results. On average, we got the speedup values of 7.2× and 13.1× compared against CPU computation, using a single GPU and two GPUs, respectively.

To obtain higher speedup values, we need to improve the implementation of genetic operators used in each thread in the blocks. Figure 6 shows the code of the PMX operator used in this study. The flow is the same as is used in CPU implementation. Here, cut1 and cut2 are cut points and have different values among threads. Thus, the loop numbers in the "for" statements in each thread are different from each other. Further, whether the branch condition holds or not in each thread is also different from thread to thread. These differences among threads increase the computation time in a block because each warp of 32 threads is essentially run in a SIMD fashion in a MP; high performance can only be achieved if all of a warp's threads execute the same instruction.

To use the same cut points among threads in one generation may result in some speedup of the execution of the operator, but we need further consideration of operators suitable for more efficient GPU computation.

5 Conclusions

In this chapter, we proposed an evolutionary algorithm for solving QAPs with parallel independent runs using GPU computation and gave an analysis of the results. In this parallel model, a set of small-size subpopulations was run in parallel in each block in CUDA independently. With this scheme, we got a performance of GPU computation that is almost proportional to the number of equipped multiprocessors (MPs) in the GPUs.

We explained these computational results by performing statistical analysis. Regarding performance comparison to CPU computations, GPU computation showed speedup values of 7.2× and 13.1× on average using a single GPU and two GPUs, respectively. The parallel independent run model is the simplest of the various parallel evolutionary computation models, and, among the models, it demonstrates the lower limit performance. We can consider many parallel evolutionary models for GPU computation. To implement these models and analyze them remain for future work.

Acknowledgements This research is partially supported by the Ministry of Education, Culture, Sports, Science and Technology of Japan under Grant-in-Aid for Scientific Research No. 22500215.

References

1. Burkard, R., Çela, E., Karisch, S., Rendl, F.: QAPLIB - A quadratic assignment problem library. www.seas.upenn.edu/qaplib (2009)
2. Eshelman, L.J.: The CHC adaptive search algorithm: how to have safe search when engaging in nontraditional genetic recombination. In: Foundations of Genetic Algorithms, pp. 265–283. Morgan Kaufmann, Los Altos (1991)
3. Fok, K., Wong, T., Wong, M.: Evolutionary computing on consumer-level graphics hardware. IEEE Intell. Syst. **22**(2), 69–78 (2007)
4. Goldberg, D.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading (1989)
5. Li, J.M., Wang, X.J., He, R.S., Chi, Z.X.: An efficient fine-grained parallel genetic algorithm based on GPU-accelerated. In: Network and Parallel Computing Workshops, pp. 855–862 (2007)

6. Mahfoud, S.: A comparison of parallel and sequential niching methods. In: International Conference on Genetic Algorithms, pp. 136–145. Morgan Kaufmann, Los Altos (1995)
7. Maitre, O., Baumes, L.A., Lachiche, N., Corma, A., Collet, P.: Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA. In: Genetic and Evolutionary Computation Conference, pp. 1403–1410. ACM, New York (2009)
8. Maniezzo, V., Colorni, A.: The ant system applied to the quadratic assignment problem. IEEE Trans. Knowl. Data Eng. **11**(5), 769–778 (1999)
9. Mathias, K.E., Whitley, L.D.: Changing representation during search: A comparative study of delta coding. Evolut. Comput. **2**(3), 249–278 (1997)
10. NVIDIA. www.nvidia.com/page/home.html (2009)
11. Oliver, I., Smith, D., Holland, J.: A study of permutation crossover operators on the traveling salesman problem. In: International Conference on Genetic Algorithms, pp. 224–230. L. Erlbaum Associates Inc., London (1987)
12. Sahni, S., Gonzalez, T.: P-complete approximation problems. J. ACM **23**, 555–565 (1976)
13. Stützle, T., Hoos, H.: Max-min ant system. Future Gener. Comput. Syst. **16**(9), 889–914 (2000)
14. Tsutsui, S.: Parallel ant colony optimization for the quadratic assignment problems with symmetric multi processing. In: Ant Colony Optimization and Swarm Intelligence, pp. 363–370s. Springer, Berlin (2008)
15. Tsutsui, S., Fujimoto, N.: Solving quadratic assignment problems by genetic algorithms with GPU computation: a case study. In: Genetic and Evolutionary Computation Conference (Companion), pp. 2523–2530. ACM, New York (2009)
16. Tsutsui, S., Ghosh, A., Corne, D., Fujimoto, Y.: A real coded genetic algorithm with an explorer and an exploiter populations. In: International Conference on Genetic Algorithm (ICGA97), pp. 238–245. Morgan Kaufmann, Los Altos (1997)
17. Whitley, D., Starkweather, T., Fuquay, D.: Scheduling problems and traveling salesman problem: The genetic edge recombination operator. In: International Conference on Genetic Algorithms, pp. 133–140. Morgan Kaufmann, Los Altos (1989)

Many-Threaded Differential Evolution on the GPU

Pavel Krömer, Jan Platoš, Václav Snášel, and Ajith Abraham

Abstract Differential evolution (DE) is an efficient populational meta-heuristic optimization algorithm that has been applied to many difficult real-world problems. Due to the relative simplicity of its operations and real-encoded data structures, it is very suitable for a parallel implementation on multicore systems and on the GPUs that nowadays reach peak performance of hundreds and thousands of giga FLOPS (floating-point operations per second). In this chapter, we present a simple yet highly parallel implementation of differential evolution on the GPU using the CUDA (Compute Unified Device Architecture) architecture and demonstrate its performance on selected test problems.

1 Introduction

Differential evolution (DE) is a popular meta-heuristic optimization algorithm belonging to the wide family of evolutionary algorithms (EAs). As with many other evolutionary algorithms, it aims to solve the optimization problems by a simulated evolution of a population of candidate solutions. The population of candidates evolved by the algorithm performs a massively parallel search through the problem domain towards globally optimal solutions. The candidate solutions can be seen as individual points on the fitness landscape of the solved problem that are iteratively and in many directions at once moved towards promising regions on the fitness landscape. The implicit parallelism of the algorithm makes it even more interesting for an implementation on a truly parallel platform.

P. Krömer (✉) · J. Platoš · V. Snášel · A. Abraham
IT4Innovations, VŠB - Technical University of Ostrava, 17. listopadu 12, Ostrava, Czech Republic
e-mail: pavel.kromer@vsb.cz; jan.platos@vsb.cz; vaclav.snasel@vsb.cz; ajith.abraham@ieee.org

In this chapter, we present a fine-grained implementation of DE designed specifically for super parallel SIMD (single-instruction multiple-data) devices such as the GPUs. The SIMD hardware found in the modern day GPUs supports parallel execution of hundreds of threads at the same time and the software drivers and runtime libraries allow efficient scheduling of tens of thousands of threads. General-purpose computing on graphics processing units (GPGPU) can use either commodity hardware such as the GPUs used primarily for computer graphics and entertainment or GPU coprocessors designed to perform massively parallel computations in the first place.

This chapter is organized in the following way: in Sect. 2, the basic principles of differential evolution and some of its applications are presented. Section 3 gives a brief overview of GPU computing, the CUDA platform, and recent implementations of differential evolution on GPUs. Many-threaded differential evolution is presented in Sect. 4, and its performance on selected test problems, first reported in [16] and [17], is described in Sect. 5.

2 Differential Evolution

Differential evolution (DE) is a versatile and easy to use stochastic evolutionary optimization algorithm [29]. It is a population-based optimizer that evolves a population of real-encoded vectors representing the solutions to given problems. DE was introduced by Storn and Price in 1995 [38, 39], and it quickly became a popular alternative to the more traditional types of evolutionary algorithms. It evolves a population of candidate solutions by iterative modification of candidate solutions by the application of differential mutation and crossover [29]. In each iteration, the so-called trial vectors are created from the current population by differential mutation and further modified by various types of crossover operator. At the end, the trial vectors compete with existing candidate solutions for survival in the population.

2.1 The DE Algorithm

DE starts with an initial population of N real-valued vectors. The vectors are initialized with real values either randomly or so that they are evenly spread over the problem space. The latter initialization leads to better results of the optimization [29].

During the optimization, DE generates new vectors that are scaled perturbations of existing population vectors. The algorithm perturbs selected base vectors with the scaled difference of two (or more) other population vectors in order to produce the trial vectors. The trial vectors compete with members of the current population with the same index called the target vectors. If a trial vector represents a better solution than the corresponding target vector, it takes its place in the population [29].

There are two most significant parameters of DE [29]. The scaling factor $F \in [0, \infty]$ controls the rate at which the population evolves and the crossover probability $C \in [0, 1]$ determines the ratio of bits that are transferred to the trial vector from its opponent. The size of the population and the choice of operators are other important parameters of the optimization process.

The basic operations of classic DE can be summarized using the following formulas [29]: the random initialization of the ith vector with N parameters is defined by

$$x_i[j] = rand(b_j^L, b_j^U), \quad j \in \{0, \ldots, N-1\} \tag{1}$$

where b_j^L is the lower bound of the jth parameter, b_j^U is the upper bound of the jth parameter and $rand(a, b)$ is a function generating a random number from the range $[a, b]$. A simple form of differential mutation is given by

$$v_i^t = v_{r1} + F(v_{r2} - v_{r3}) \tag{2}$$

where F is the scaling factor and v_r^1, v_r^2, and v_r^3 are three random vectors from the population. The vector v_{r1} is the base vector, v_{r2} and v_{r3} are the difference vectors, and the ith vector in the population is the target vector. It is required that $i \neq r1 \neq r2 \neq r3$. The differential mutation in 2D (i.e., for $N = 2$) is illustrated in Fig. 1. The uniform crossover that combines the target vector with the trial vector is given by

$$l = rand(0, N-1) \tag{3}$$

$$v_i^t[m] = \begin{cases} v_i^t[m] & \text{if } (rand(0,1) < C) \text{ or } m = l \\ x_i[m] & \end{cases} \tag{4}$$

for each $m \in \{1, \ldots, N\}$. The uniform crossover replaces with probability $1 - C$ the parameters in v_i^t by the parameters from the target vector x_i. The outline of classic DE according to [10] is summarized in Algorithm 1. However, the monograph on DE by Price, Storn, and Lampinen [29] lists a different version of the basic DE. They first form a whole new population of trial vectors P^t and subsequently merge P and P^t. It means that the newly created trial vectors do not enter the population of candidate solutions P immediately and therefore cannot participate in the creation of next trial vectors until the whole population was processed.

There are also many other modifications to the classic DE. Mostly, they differ in the implementation of particular DE steps such as the initialization strategy, the vector selection, the type of differential mutation, the recombination operator, and control parameter selection and usage [10, 29].

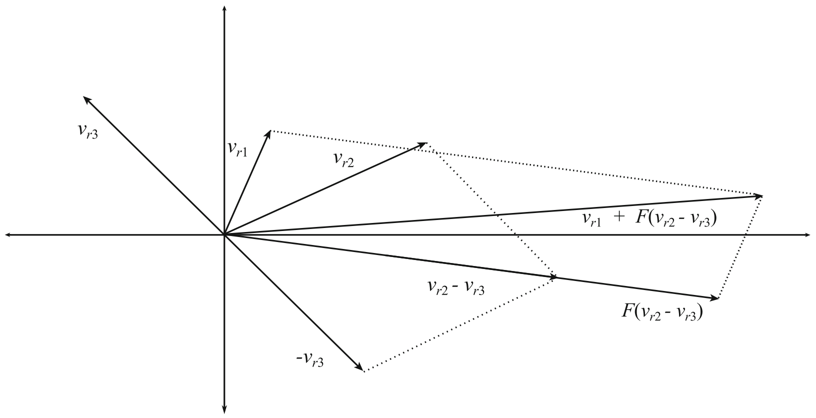


Fig. 1 Differential mutation

```
1  Initialize the population $P$ consisting of $M$ vectors using (1);
2  Evaluate an objective function ranking the vectors in the population;
3  while Termination criteria not satisfied do
4      for $i \in \{1, \dots, M\}$ do
5          Differential mutation: Create trial vector $v_i^t$ according to (2);
6          Validate the range of coordinates of $v_i^t$. Optionally adjust coordinates of $v_i^t$ so,
           that $v_i^t$ is valid solution to given problem;
7          Perform uniform crossover. Select randomly one parameter $l$ in $v_i^t$ and modify
           the trial vector using (3);
8          Evaluate the trial vector. If the trial vector $v_i^t$ represent a better solution than
           population vector $v^i$, replace $v^i$ in $P$ by $v_i^t$;
9      end
10 end
```

Algorithm 1: A summary of classic differential evolution

The initialization strategy affects the way vectors in the initial population are placed in the problem space. In general, a better initial coverage of the problem space represents a better starting point for the optimization process because the vectors can explore various regions of the fitness landscape from the very beginning [29].

The selection strategy defines how are the target vector, the base vector, and the difference vectors selected. Moreover, it has an effect on the time each vector survives in the population, which can be given either by the age of the vector or by the fitness of the vector. Popular base vector selection strategies are the random selection and methods based on stochastic universal sampling [3, 29]. The random selection without restrictions allows the same vector to be used as a base vector more than once in each generation. The methods based on stochastic universal sampling ensure that all vectors are used as base vectors exactly once in each generation. The selection methods based on the stochastic universal sampling generate a permutation of vector indexes that defines which base vector will be coupled with which target vector [29]. An alternative base vector selection strategy is called the biased base vector selection. The biased base vector selection uses the information about the fitness value of each vector when selecting base vectors. The biased base vector selection strategies include the best-so-far base vector selection (the best vector in the population is always selected as base vector) and target-to-best base vector selection (the base vector is an arithmetic recombination of the target vector and the best-so-far vector) [29]. The biased base vector selection schemes introduce a more intensive selection pressure which can, as in other evolutionary techniques, result in faster convergence but it can also lead to a loss of diversity in the population.

The differential mutation is the key driver of DE. It creates a trial vector as a recombination of base vector and scaled difference of selected difference vectors. The scaling factor F, which modifies vector differences, can be a firmly set constant, a random variable selected according to some probability distribution, or defined by some other function as, e.g., in self-adaptive DE variants. A variable scaling factor increases the number of vector differentials that can be generated given the population P [29]. In general, a smaller scaling factor causes smaller steps in the fitness landscape traversal while a greater scaling factor causes larger steps. The former leads to longer time for the algorithm to converge, and the latter can cause the algorithm to miss the optima [29].

The recombination (crossover) operator plays a special role in DE. It achieves a similar goal as the mutation operator in other evolutionary algorithms, i.e., it controls the introduction of new material to the population using a mechanism similar to the n-point crossover in traditional EAs. The crossover probability $C \in [0, 1]$ defines the probability that a parameter will be inherited from the trial vector. Similarly, $C - 1$ is the probability that the parameter will be taken from the target vector. Crossover probability has also a direct influence on the diversity of the population [10]. An increased diversity initiated by larger C means a more intensive exploration and faster convergence of the algorithm at the cost of the robustness of the algorithm [29]. Common DE crossover operators include exponential crossover and the uniform crossover. Some other crossover operators are, e.g., arithmetic crossover and either-or-crossover [10, 29].

2.2 DE Variants and Applications

Particular DE variants are often referred to using a simple naming scheme [10, 29] that uses the pattern DE/x/y/z. In the pattern, x describes the base vector selection type, y represents the number of differentials, and z is the type of crossover operator. For example, classic DE is often called DE/rand/1/bin, which means that it uses random base vector selection, a single vector differential, and uniform (binominal) crossover. Some other variants of DE described in the literature are [10]:

1. *DE/best/1/**, which always select the so far best vector as the base vector and its differential mutation can be described by

$$v_i^t = v_{\text{best}} + F(v_{r2} - v_{r3}) \tag{5}$$

   This DE variant can use any type of the crossover operator.
2. *DE/*/n_v/** that uses arbitrary base vector selection strategy, n_v differential vectors, and type of the crossover operator. Its differential mutation can be described by

$$v_i^t = v_{r1} + F \sum_{k=1}^{n_v} (v_{r2}^k - v_{r3}^k) \tag{6}$$

   where $v_{r2}^k - v_{r3}^k$ is the kth vector differential.
3. *DE/rand-to-best/n_v/** combines random base vector selection with the $best$ base vector selection strategy:

$$v_i^t = \gamma v_{\text{best}} + (1 - \gamma) v_{r1} + F \sum_{k=1}^{n_v} (v_{r2}^k - v_{r3}^k) \tag{7}$$

   It uses the parameter $\gamma \in [0, 1]$ to control the exploitation of the mutation. The larger γ the larger the exploitation. The parameter γ can be also adaptive.
4. *DE/current-to-best/1+n_v/** uses at least two difference vectors. The first differential participating in the mutation is computed between the best vector and the base vector and all the other differentials are computed using randomly selected vectors:

$$v_i^t = v_{r1} + F(v_{\text{best}} - v_{r1}) + F \sum_{k=1}^{n_v} (v_{r2}^k - v_{r3}^k) \tag{8}$$

DE is a very successful algorithm with a number of applications. DE was used, among others, for merit analysis, for non-imaging optical design, for the optimization of industrial compressor supply systems, for multi-sensor fusion, for the determination of earthquake hypocenters, for 3D medical image registration, for the design of erasure codes, for digital filters, for the analysis of X-ray reflectivity data, to solve the inverse fractal problem, and for the compensation of RF-driven plasmas [29]. It was also used for evolutionary clustering [8] and to optimize the deployment of sensor nodes [34]. Despite its continuous nature, the DE was used also to solve combinatorial optimization problems. The applications of DE in this domain included turbo code interleaver optimization [19], scheduling of independent tasks in heterogeneous computing environments [15, 20], search for optimal solutions to the linear ordering problem [37], and many others.

DE is a successful evolutionary algorithm designed for continuous parameter optimization driven by the idea of scaled vector differentials. That makes it an interesting alternative to the widespread genetic algorithms that are designed to work primarily with discrete encoding of the candidate solutions. As well as genetic algorithms, it represents a highly parallel population-based stochastic search meta-heuristic. In contrast to GA, differential evolution uses the real encoding of candidate solutions and different operations to evolve the population. It results in different search strategies and different directions found by DE when crawling a fitness landscape of the problem domain.

3 GPU Computing

Modern graphics hardware has gained an important role in the area of parallel computing. GPUs have been used to power gaming and 3D graphics applications, but recently they have been used to accelerate general computations as well. The new area of general-purpose computing on graphics processing units (GPGPU) has been flourishing since then. The data parallel architecture of GPUs is suitable for vector and matrix algebra operations, which leads to the wide use of GPUs in the area of scientific computing with applications in information retrieval, data mining, image processing, data compression, etc.

To simplify the development of GPGPU programs, various vendors have introduced languages, libraries, and tools to create parallel code rapidly. The GPU platform and API developed by nVidia is called CUDA (Compute Unified Device Architecture). It is based on the CUDA-C language, which is an extension to C that allows development of GPU routines called kernels. Each kernel defines instructions that are executed on the GPU by many threads at the same time following the SIMD model. The threads can be organized into so-called thread groups that can benefit from GPU features such as fast shared memory, atomic data manipulation, and synchronization. The CUDA runtime takes care of the scheduling and execution of the thread groups on available hardware. The set of thread groups requested to execute a kernel is called in CUDA terminology a grid. A kernel program can use several types of memory: fast local and shared memory, large but slow global memory, and fast read-only constant memory and texture memory. The structure of CUDA program execution and the relation of threads and thread groups to device memory is illustrated in Fig. 2. GPU programming has established a new platform for evolutionary computation [9]. The majority of evolutionary algorithms, including genetic algorithms (GA) [28], genetic programming (GP) [21, 33], and DE [40, 42, 43], have been implemented on GPUs. Most of the contemporary implementations of evolutionary algorithms on GPUs map each candidate solution in the population to a single GPU thread. However, recent work in the area of evolutionary computation on GPUs has introduced further parallelization of GA by, e.g., many-threaded implementation of the crossover operator and local search [12].

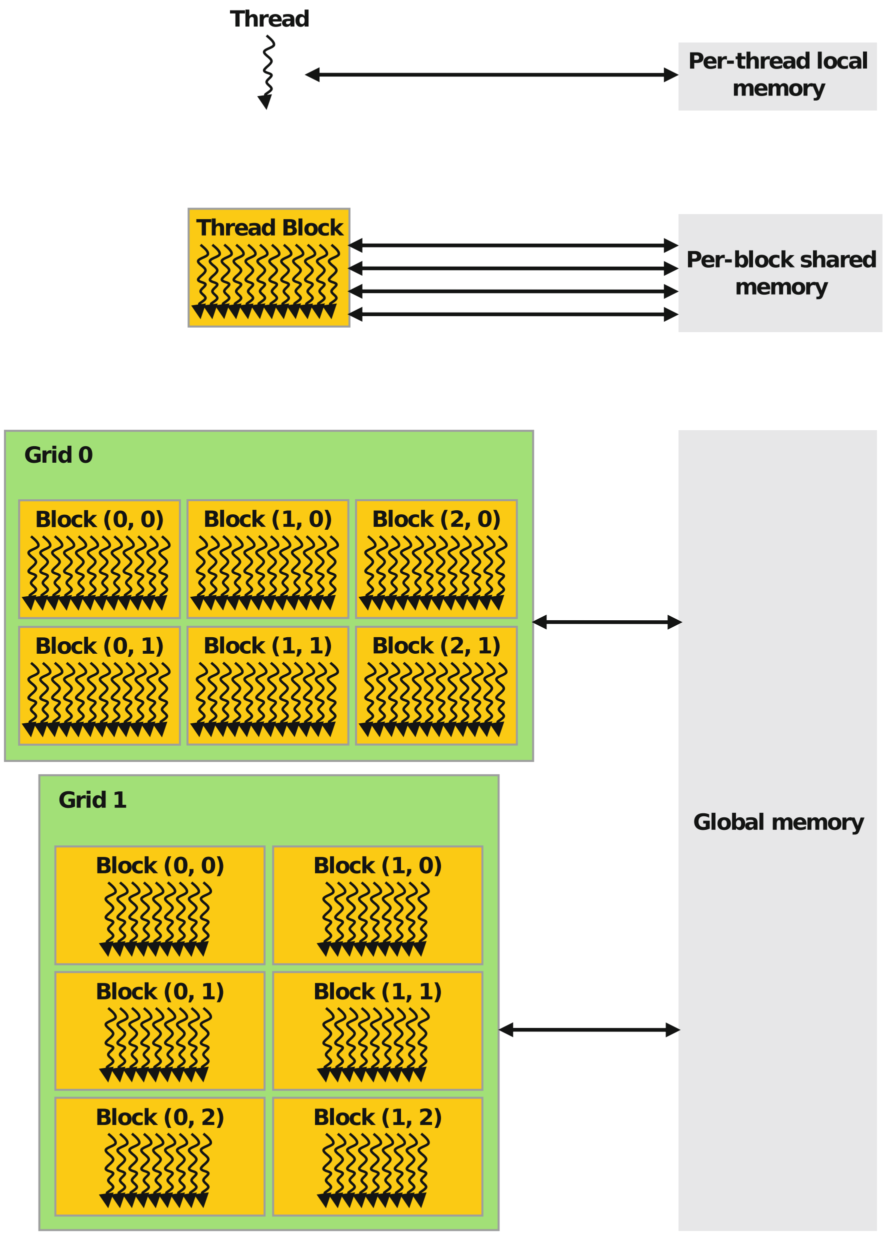


Fig. 2 CUDA-C program structure and memory hierarchy [26]

3.1 Differential Evolution on the GPU

Due to the simplicity of its operations and real encoding of the candidate solutions, DE is suitable for parallel implementation on the GPUs. In DE, each candidate solution is represented by a vector of real numbers (parameters), and the population as a whole can be seen as a real-valued matrix. Moreover, both the mutation operator and the crossover operator can be implemented easily as straightforward vector operations.

The first implementation of DE on the CUDA platform was introduced in the early 2010 by de Veronese and Krohling [40]. Their DE was implemented using the CUDA-C language, and it achieved speedup between 19 and 34 times comparing to the CPU implementation on a set of benchmarking functions. The generation of random numbers was implemented using the Mersenne Twister from the CUDA SDK, and the selection of random trial vectors for mutation was done on the CPU.

Zhu [42], and Zhu and Li [43] implemented DE on CUDA as part of a differential evolution-pattern search algorithm for bound-constrained optimization problems and as part of a differential evolutionary Markov chain Monte Carlo method (DE-MCMC), respectively. In both cases, the performance of the algorithms was demonstrated on a set of continuous benchmarking functions.

The common property of the above DE implementations is the mapping of a single GPU thread to one candidate solution and the usage of the Mersenne Twister from the CUDA SDK for random number generation on the GPU. Moreover, some parts of the random number generation process were [42] offloaded to the CPU. In this work, we use a new implementation of DE on the CUDA platform using many threads to process each candidate solution and utilizing the GPU to generate random numbers needed for the optimization.

4 Many-Threaded Differential Evolution on the GPU

The goal of the implementation of DE on the CUDA platform was achieving high parallelism while retaining the simplicity of the algorithm. The implementation consists of a set of CUDA-C kernels for generation of initial population, generation of batches of random numbers for the decision making, DE processing including generation of trial vectors, mutation and crossover, verification of the generated vectors, and the merger of parent and offspring populations. Besides these kernels implementing DE, an implementation of the fitness function evaluation was done in a separate kernel. The overview of the presented DE implementation is shown in Fig. 3. The kernels were implemented using the following principles:

1. Each candidate solution is processed by a thread block (thread group). The number of thread groups is in nVidia CUDA 4.0 limited to $(2^{16} - 1)^3$ and in earlier versions to $(2^{16} - 1)^2$. Hence, the maximum population size is in this case the same.

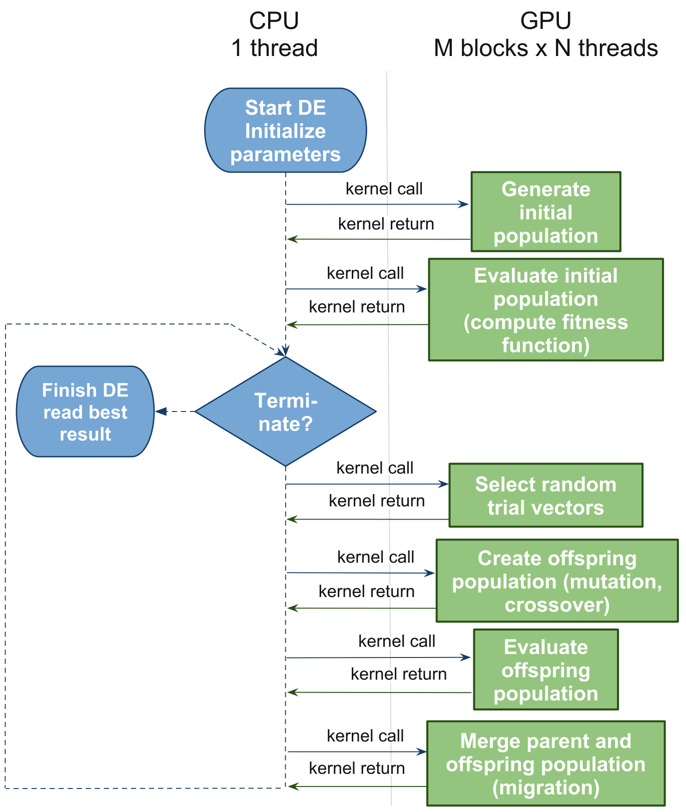

CPU
1 thread
GPU
M blocks x N threads
Start DE Initialize parameters
kernel call
kernel return
Generate initial population
kernel call
kernel return
Evaluate initial population (compute fitness function)
Finish DE read best result
Termi-nate?
kernel call
kernel return
Select random trial vectors
kernel call
kernel return
Create offspring population (mutation, crossover)
kernel call
kernel return
Evaluate offspring population
kernel call
kernel return
Merge parent and offspring population (migration)

Fig. 3 The flowchart of the DE implementation on CUDA

2. Each vector parameter is processed by a thread. The limit of threads per block depends in CUDA on the hardware compute capability, and it is 512 for compute capability 1.x and 1024 for compute capability 2.x [26]. This limit enforces the maximum vector length. This is enough for the application area considered in this paper. The mapping of CUDA threads and thread blocks to the DE vectors is illustrated in Fig. 4.
3. Each kernel call aims to process the whole population in a single step, e.g., it asks the CUDA runtime to launch M blocks with N threads in parallel. The CUDA runtime executes the kernel with respect to available resources.

Such an implementation brings several advantages. First, all the generic DE operations can be considered done in parallel, and, thus, their complexity reduces from $M \times N$ (population size multiplied by vector length) to c (constant, duration of the operation plus CUDA overhead). Second, this DE operates in a highly parallel way also on the logical level. A population of offspring chromosomes of the same size as the parent population is created in a single step and later merged with the parent population. Third, the evaluation of vectors is accelerated by the implementation of the fitness function on GPU.

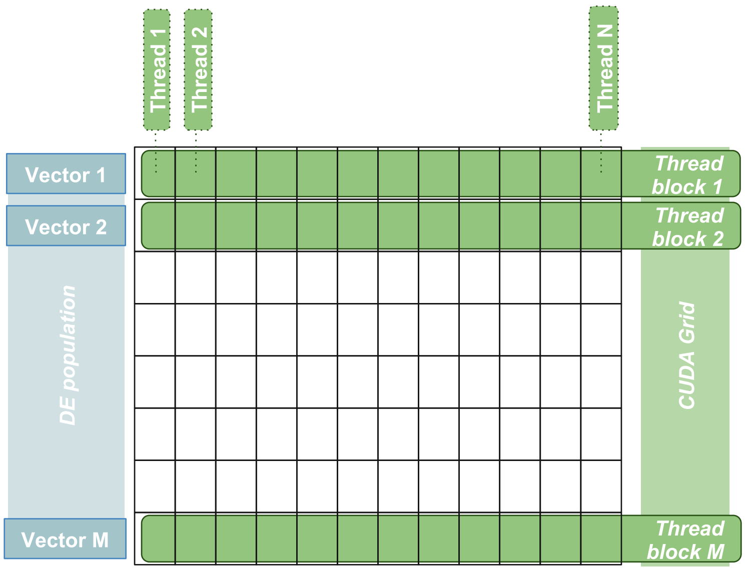


Fig. 4 The mapping of CUDA threads and thread blocks on DE population

5 The Performance of the Many-Threaded DE on the GPU

The performance of many-threaded DE was evaluated on several test problems. To perform the experiments, the *DE/rand/1/bin* type of DE was implemented for both, the CUDA platform and sequential execution on the CPU. When not stated otherwise, the presented experiments were performed on a server with two dual core AMD Opteron processors at 2.6 GHz and an nVidia Tesla C2050 with 448 cores at 1.15 GHz.

In contrast to the majority of DE applications, the many-threaded DE on the GPU was used to solve combinatorial optimization problems. However, to provide at least indirect comparison with previous DE implementations on CUDA, the optimization of the test function f_2 was performed.

5.1 Function Optimization

The previous GPU based DE implementations were most often tested using a set of continuous benchmarking functions. To provide at least a rough comparison of our approach to another DE variant, we have implemented a DE searching for the minimum of the test function f_2 from [40]:

$$f_2(x) = \sum_{i-1}^{n} \left(x_i^2 - 10\cos(2\pi x_i) + 10\right) \tag{9}$$

Table 1 Comparison of the many-threaded DE with CUDA-C implementation from [40]

	DE from [40]		Proposed DE	
f_2 Variables	Value	Time	Value	Time
100/128	278.18	0.64	232.8668	0.6604656
100/128	98.53	27.47	32.98331	27.00045
256	N/A	N/A	91.10914	27.00054
512	N/A	N/A	295.6335	27.00055

We note that the comparison is indirect and rather illustrative since the two algorithms were executed for the same test function but with different dimensions, on different hardware, and with different settings.

The purpose of this comparison is to show whether the proposed many-threaded DE can find similar, better, or worse solutions of a previously used benchmark function. From Table 1, it can be seen that many-threaded DE has found in a very similar time frame a solution to f_2 with better (i.e., lower) fitness value. Many-threaded DE was executed on a more powerful GPU than DE in [40], but it had to solve a function with 1.28 times larger dimension. In 0.64 s, it delivered approximately 1.19 times better (in terms of fitness value) solution, and in 27 s it had found an approximately three times better solution than the previous implementation.

This leads us to the conclusion that the proposed DE is able to find a good minimum of a continuous functions and it appears to be competitive compared with previous CUDA-C implementations.

5.2 *Linear Ordering Problem*

The linear ordering problem (LOP) is a well-known NP-hard combinatorial optimization problem. It has been intensively studied and there are plenty of exact, heuristic, and meta-heuristic algorithms for LOP. With its large collection of well-described testing data sets, the LOP represents an interesting testbed for meta-heuristic algorithms for combinatorial optimization [22, 23].

The LOP can be formulated as a graph problem [22]. For a complete directed graph $D_n = (V_n, A_n)$ with weighted arcs c_{ij}, compute a spanning acyclic tournament T in A_n such that $\sum_{(i,j)\in T} c_{ij}$ is as large as possible. The LOP can also be defined as a search for an optimal column and row reordering of a weight matrix C [6, 22, 35, 36]. Consider a matrix $C^{n\times n}$, permutation Π, and a cost function f:

$$f(\Pi) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} c_{\Pi(i)\Pi(j)} \tag{10}$$

The LOP is a search for permutation Π so that $f(\Pi)$ is maximized, i.e., the permutation restructures the matrix C so that the sum of its elements above the main diagonal is maximized. The LOP is an NP-hard problem with a number of applications in scheduling (scheduling with constraints), graph theory, economy, sociology (paired comparison ranking), tournaments, and archaeology among others.

In economics, LOP algorithms are deployed to triangularize input-output matrices. The resulting permutation provides useful information on the stability of the investigated economy. In archaeology, LOP algorithms are used to process the Harris Matrix, a matrix describing most probable chronological ordering of samples found in different archaeological sites [36]. Other applications of LOP include the equivalent graph problem, the related graph problem, the aggregation of individual preferences, ranking in sports tournaments, and the minimization of crossing [22].

A variant of the LOP is the linear ordering problem with cumulative costs (LOPCC) that has applications in the optimization of the universal mobile telecommunication standard (UMTS) in mobile phone telecommunication systems [4, 22, 30].

5.2.1 LOP Data Sets

There are several test libraries used for benchmarking LOP algorithms. They are well preprocessed and thoroughly described, and the optimal (or so-far best) solutions are available. The majority of the investigated algorithms were tested against the LOLIB library. The original LOLIB library contains 49 instances of input-output matrices describing European economies in the 1970s. Optimal solutions of the LOLIB matrices are available. Although the LOLIB contains real-world data, it is considered rather simple and easy to solve [31]. Mitchell and Bochers [24] have published an artificial LOP data library and a LOP instance generator. The data (MBLB) and code are available from Rensselaer Polytechnic Institute.[1]

Schiavinotto and Stützle [35, 36] have shown that the LOLIB and MBLB instances are significantly different, having diverse high-level characteristics of the matrix entries such as sparsity or skewness. The search space analysis revealed that MBLB instances typically have higher correlation length and also a generally larger fitness-distance correlation than LOLIB instances. It suggests that MBLB instances should be easier to solve than LOLIB instances of the same dimension. Moreover, a new set of large artificial LOP instances (based on LOLIB) called XLOLIB was created and published. Another set of LOP instances is known as the Stanford GraphBase (SGB). The SGB is composed of larger input-output matrices describing the US economies.

[1] http://www.rpi.edu/~mitchj/generators/linord/.

Many LOP libraries are hosted by the Optsicom project.[2] The Optsicom archive contains LOLIB, SGB, MBLB, XLOLIB, and other LOP instances. One important feature of the Optsicom LOP archive is that its LOP matrices are normalized [22], i.e., they were preprocessed so that:

- All matrix entries are integral.
- $c_{ii} = 0$ for all $i = \{1, 2, \ldots, n\}$.
- $min\{c_{ij}, c_{ji}\} = 0$ for all $1 \leq i < j \leq n$.

5.2.2 LOP Algorithms

There are several exact and heuristic algorithms for the linear ordering problem. The exact algorithms are strongly limited by the fact that LOP is a NP-hard problem (i.e., there are no exact algorithms that could solve LOP in polynomial time). Among the exact algorithms, branch & bound approach based on LP relaxation of the LOP for the lower bound, a branch & cut algorithm, and interior point/cutting plane algorithm attracted attention [36]. Exact algorithms are able to solve rather small general instances of the LOP and bigger instances (with the dimension of few hundred rows and columns) of certain classes of LOP [36].

A number of heuristic algorithms, including the greedy algorithms, local search, elite tabu search, scattered search, and iterated local search, were used to solve the LOP instances [13, 22, 36]. In this work, we use the LOP as a testbed for the performance evaluation of a many-threaded DE implementation powered by the GPU.

The LOP candidate solutions were for the purpose of the DE represented using the random keys encoding [2]. With this encoding, the candidate solution consists of an array of real numbers. The change in the order of elements of the array after sorting corresponds to a permutation. Due to its simplicity, the random keys encoding is a natural choice for differential evolution of permutations.

Computational experiments were performed on a server described in the beginning of this section. For comparison, the LOP was also computed on a laptop with Intel Core i5 at 2.3 GHz. Although the CPUs used in the experiments were multicore, the LOP evaluation was single-threaded. The comparison of fitness computation times for different population sizes is illustrated in Fig. 5 (note the log scale). In the benchmark, a population of candidate solutions of a LOP instance with the dimension 50 was evaluated on the GPU and on two CPUs.

We can see that the Core i5 is always 3.3–4.2 times faster than the Opteron. The Tesla C2050 is equally fast as the Opteron when evaluating 16 LOP candidates and 1.4–9.7 times faster for larger LOP candidate populations. The Core i5 is faster than GPU for population sizes 16, 32, and 64. The GPU performs similarly as Core i5 when evaluating 128 LOP candidates and 1.4–2.4 times faster for larger populations.

[2] http://heur.uv.es/optsicom/LOLIB/.

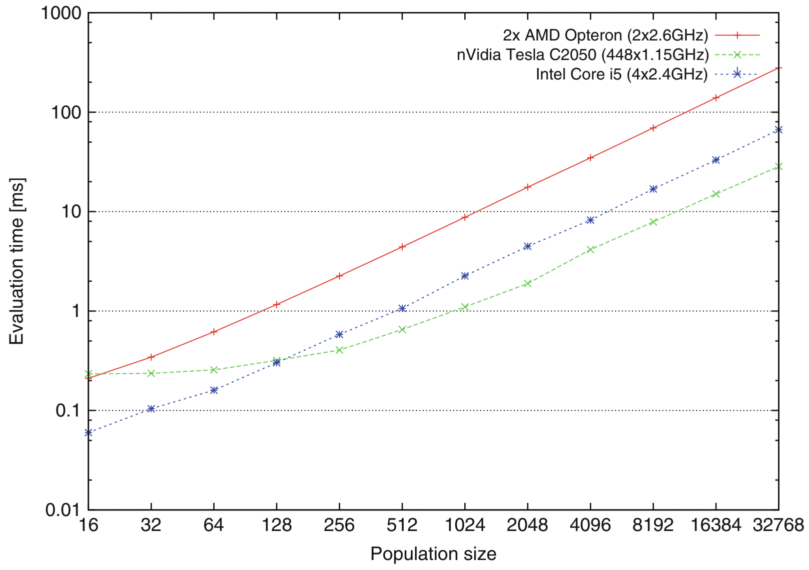


Fig. 5 The speed of LOP evaluation on the CPUs and on the GPU

5.2.3 Search for LOLIB Solutions on the GPU

The LOLIB solutions obtained by the DE on CUDA and by a sequential DE implementation are shown in Table 2. The table contains best and average error after 5 s and 10 s computed from 10 independent optimization runs for each LOP matrix. We can see that the largest average error after 5 s is 0.232 for matrix t69r11xx, which is better than the best LOP solution found by the DE in [37]. The average error for all LOLIB matrices was 0.043 after 5 s and 0.031 after 10 s.

We have compared the progress of DE for LOP on the GPU and on the CPU. A typical example of the optimization is shown in Fig. 6. Apparently, DE on the GPU delivers optimal or nearly optimal results very quickly compared to the CPU.

Interestingly, the results of the optimization after 10 s were sometimes worse than the results of optimization after 5 s (the results were obtained in separate program runs). It suggests that DE on CUDA quickly converges to a suboptimal solution but sometimes fails to find global optimum. On the other hand, the algorithm has found global optimum in all test runs for 15 out of 49 LOP matrices, which is a good result for a pure meta-heuristic.

5.2.4 Search for N-LOLIB Solutions

DE on CUDA was also used to find solutions to the normalized LOLIB (N-LOLIB) library. The results of DE for LOP implemented on the GPU are shown in Table 3.

Table 2 The average error of LOLIB solutions (in percent)

	After 5 s		After 10 s			After 5 s		After 10 s	
LOP	Best	Avg	Best	Avg	LOP	Best	Avg	Best	Avg
be75eec	1.13E−03	2.00E−03	0	8.83E−03	t70k11xx	4.30E−04	2.16E−03	1.07E−02	1.19E−02
be75np	2.53E−04	5.18E−04	1.01E−03	2.72E−03	t70l11xx	0	0	0	0
be75oi	0	0	0	0	t70n11xx	0	0	0	0
be75tot	1.11E−01	1.12E−01	1.11E−01	1.15E−01	t70u11xx	1.12E−01	1.12E−01	1.35E−01	1.42E−01
stabu1	5.69E−03	7.79E−03	2.30E−02	2.31E−02	t70w11xx	2.21E−02	1.23E−01	3.52E−02	4.50E−02
stabu2	6.47E−02	6.73E−02	6.16E−02	6.21E−02	t70x11xx	8.51E−02	8.57E−02	6.94E−02	8.85E−02
stabu3	1.38E−01	1.44E−01	1.50E−01	1.84E−01	t74d11xx	0	0	1.44E−02	1.44E−02
t59b11xx	0	0	0	0	t75d11xx	7.26E−03	9.38E−03	1.45E−04	8.42E−04
t59d11xx	0	0	0	0	t75e11xx	2.16E−02	8.47E−02	2.10E−02	2.72E−02
t59f11xx	0	0	0	0	t75i11xx	2.28E−04	1.96E−03	2.38E−02	6.43E−02
t59i11xx	2.19E−03	4.97E−02	3.39E−03	4.07E−03	t75k11xx	0	0	0	0
t59n11xx	0	0	0	0	t75n11xx	0	0	0	0
t65b11xx	2.60E−02	2.21E−01	7.46E−02	8.55E−02	t75u11xx	6.77E−02	6.83E−02	7.62E−02	7.79E−02
t65d11xx	2.89E−02	2.95E−02	2.96E−02	3.53E−02	tiw56n54	7.09E−03	1.03E−02	7.09E−03	7.89E−03
t65f11xx	8.52E−02	9.04E−02	1.37E−01	1.37E−01	tiw56n58	1.94E−03	2.01E−03	1.94E−03	2.53E−03
t65i11xx	2.23E−02	2.28E−02	1.01E−02	1.54E−02	tiw56n62	0	0	0	0
t65l11xx	0	0	0	0	tiw56n66	0	1.46E−02	0	0
t65n11xx	0	0	0	0	tiw56n67	0	3.24E−04	0	0
t65w11xx	1.00E−01	1.05E−01	1.12E−01	1.43E−01	tiw56n72	3.02E−03	3.05E−03	2.16E−04	4.97E−04
t69r11xx	1.77E−01	2.32E−01	4.60E−02	4.65E−02	tiw56r54	5.49E−03	1.19E−02	7.06E−03	7.14E−03
t70b11xx	0	0	0	0	tiw56r58	1.87E−03	9.70E−03	0	1.24E−04
t70d11xn	6.43E−02	6.51E−02	2.51E−02	2.90E−02	tiw56r66	0	3.55E−03	0	0
t70d11xx	0	3.89E−02	0	2.00E−04	tiw56r67	0	0	0	0
t70f11xx	2.08E−01	2.15E−01	1.11E−02	3.08E−02	tiw56r72	0	0	0	0
t70i11xx	1.11E−01	1.63E−01	7.97E−02	8.32E−02					

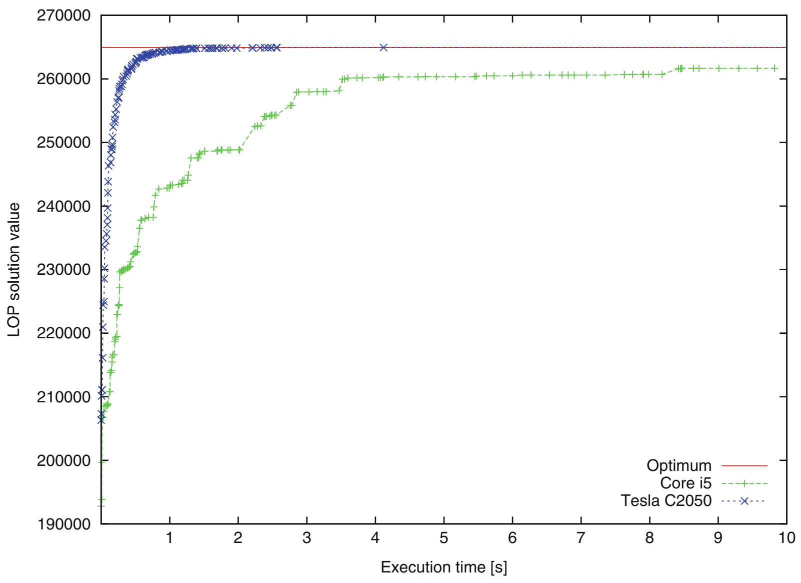


Fig. 6 Example of DE for LOLIB matrix be75eec on CPU and GPU

The average error for all N-LOLIB matrices was 0.034 after 5 s and 0.037 after 10 s. Moreover, the DE has found the optimal solution for 18 out of 50 LOP instances.

When we compare these results to N-LOLIB results obtained recently by several meta-heuristic methods in [22], we can see that the differential evolution performs better than pure genetic algorithms with average error 0.38 and optimal results found for 9 matrices. On the other hand, other meta-heuristics, including tabu search, memetic algorithms, and simulated annealing, performed better. However, we have to note that the DE used in this work is a pure meta-heuristic and uses no domain knowledge or local search to improve the solutions.

5.3 Independent Task Scheduling

In grid and distributed computing, the mixed-machine heterogeneous computing (HC) environments utilize a distributed suite of different machines to perform different computationally intensive applications that have diverse requirements [1, 5]. Task scheduling, i.e., mapping of a set of tasks to a set of resources, is required to exploit the different capabilities of a set of heterogeneous resources. It is known that an optimal mapping of computational tasks to available machines in a HC suite is an NP-complete problem [11], and, as such, it is a subject to various heuristic [5, 14, 25] and meta-heuristic [7, 27, 32, 41] algorithms, including differential evolution [18].

Table 3 The average error of N-LOLIB solutions (in percent)

	After 5 s		After 10 s			After 5 s		After 10 s	
LOP	Best	Avg	Best	Avg	LOP	Best	Avg	Best	Avg
N-be75eec	0	1.10E−03	0	0	N-t70k11xx	3.79E−03	3.97E−03	6.26E−03	6.81E−03
N-be75np	3.49E−03	2.42E−02	2.79E−04	4.04E−04	N-t70l11xx	0	0	0	0
N-be75oi	0	1.80E−04	0	0	N-t70n11xx	0	0	0	0
N-be75tot	8.30E−02	1.33E−01	1.21E−01	1.22E−01	N-t70u11xx	1.39E−01	1.65E−01	1.40E−01	1.40E−01
N-stabu70	8.28E−04	1.60E−02	8.28E−04	3.56E−03	N-t70w11xx	1.96E−01	1.99E−01	1.59E−01	1.88E−01
N-stabu74	1.05E−01	1.06E−01	6.54E−02	1.02E−01	N-t70x11xx	6.76E−02	6.80E−02	3.13E−02	1.36E−01
N-stabu75	1.72E−01	2.27E−01	1.61E−01	1.61E−01	N-t74d11xx	0	0	0	0
N-t59b11xx	0	0	0	0	N-t75d11xx	0	1.73E−05	1.73E−04	3.54E−03
N-t59d11xx	0	0	0	0	N-t75e11xx	1.97E−02	2.30E−02	5.07E−03	1.36E−02
N-t59f11xx	0	0	0	0	N-t75i11xx	9.97E−03	1.09E−02	5.51E−05	3.86E−02
N-t59i11xx	6.05E−05	6.05E−05	6.05E−05	1.27E−03	N-t75k11xx	0	0	0	0
N-t59n11xx	0	0	0	0	N-t75n11xx	0	0	0	0
N-t65b11xx	3.08E−02	3.92E−02	1.34E−01	1.86E−01	N-t75u11xx	8.14E−02	8.16E−02	8.14E−02	8.78E−02
N-t65d11xx	7.99E−03	1.06E−02	3.45E−02	3.45E−02	N-tiw56n54	1.20E−02	1.69E−02	2.18E−03	7.10E−03
N-t65f11xx	1.61E−01	1.61E−01	1.61E−01	1.61E−01	N-tiw56n58	2.40E−03	2.48E−03	2.40E−03	2.40E−03
N-t65i11xx	2.51E−03	5.48E−03	1.82E−03	1.82E−03	N-tiw56n62	0	0	0	0
N-t65l11xx	0	0	0	0	N-tiw56n66	4.86E−03	6.31E−03	0	1.55E−02
N-t65n11xx	0	0	0	0	N-tiw56n67	0	3.58E−03	0	0
N-t65w11xx	3.04E−02	4.17E−02	1.60E−01	1.62E−01	N-tiw56n72	2.74E−04	6.30E−04	3.83E−03	3.83E−03
N-t69r11xx	6.90E−02	6.90E−02	5.16E−02	5.22E−02	N-tiw56r54	1.17E−02	1.69E−02	0	5.83E−04
N-t70b11xx	0	0	0	0	N-tiw56r58	2.32E−03	2.62E−03	0	2.08E−03
N-t70d11xxb	2.67E−02	3.17E−02	2.67E−02	6.97E−02	N-tiw56r66	1.24E−02	1.48E−02	0	8.59E−04
N-t70d11xx	0	1.55E−02	0	0	N-tiw56r67	0	0	0	0
N-t70f11xx	1.28E−02	1.28E−02	1.94E−03	1.17E−02	N-tiw56r72	0	0	0	0
N-t70i11xx	2.13E−02	8.01E−02	8.34E−02	8.66E−02	N-usa79	0.2.66E−01	2.86E−01	2.95E−02	3.42E−02

A HC environment is a composite of computing resources (PCs, clusters, or supercomputers). Let $T = \{T_1, T_2, \dots, T_n\}$ denote the set of tasks that is in a specific time interval submitted to a resource management system (RMS). Assume the tasks are independent of each other with no inter-task data dependencies and preemption is not allowed (the tasks cannot change the resource they have been assigned to). Also assume at the time of receiving these tasks by RMS, m machines $M = \{M_1, M_2, \dots, M_m\}$ are within the HC environment. For our purpose, scheduling is done on the machine level, and it is assumed that each machine uses first-come, first-served (FCFS) method for performing the received tasks. We assume that each machine in the HC environment can estimate how much time is required to perform each task. In [5], the expected time to compute (ETC) the matrix was used to estimate the required time for executing a task in a machine. An ETC matrix is an $n \times m$ matrix in which n is the number of tasks and m is the number of machines. One row of the ETC matrix contains the estimated execution time for a given task on each machine. Similarly, one column of the ETC matrix consists of the estimated execution time of a given machine for each task. Thus, for an arbitrary task T_j and an arbitrary machine M_i , $[ETC]_{j,i}$ is the estimated execution time of T_j on M_i. In the ETC model we take the usual assumption that we know the computing capacity of each resource, an estimation or prediction of the computational needs of each job, and the load of prior work of each resource.

The two objectives to optimize during the task mapping are makespan and flowtime. Optimum makespan (meta-task execution time) and flowtime of a set of jobs can be defined as:

$$makespan = \min_{S \in Sched} \{ \max_{j \in Jobs} F_j \} \tag{11}$$

$$flowtime = \min_{S \in Sched} \{ \sum_{j \in Jobs} F_j \} \tag{12}$$

where $Sched$ is the set of all possible schedules, $Jobs$ stands for the set of all jobs to be scheduled, and F_j represents the time in which job j finalizes. Assume that C_{ij} $(j = 1, 2, \dots, n, i = 1, 2, \dots, m)$ is the completion time for performing the j-th task in the i-th machine and W_i $(i = 1, 2, \dots, m)$ is the previous workload of M_i, then $\sum_{j \in S(i)} C_{ij} + W_i$ is the time required for M_i to complete the tasks included in it ($S(i)$ is the set of jobs scheduled for execution on M_i in schedule S). According to the aforementioned definition, makespan and flowtime can be evaluated using:

$$makespan = \max_{i \in \{1,2,\dots,m\}} \{ \sum_{j \in S(i)} C_{ij} + W_i \} \tag{13}$$

$$flowtime = \sum_{i=1}^{m} \sum_{j \in S(i)} C_{ij} \tag{14}$$

Minimizing makespan aims to execute the whole meta-task as fast as possible while minimizing flowtime aims to utilize the computing environment efficiently.

A schedule of n independent tasks executed on m machines can be naturally expressed as a string of n integers $S = (s_1, s_2, \dots, s_n)$ that are subject to $s_i \in 1, \dots, m$. The value at the i position in S represents the machine on which the i-th job is scheduled in schedule S. Since differential evolution uses for problem encoding real vectors, real coordinates must be used instead of discrete machine numbers. The real-encoded DE vector is translated to schedule representation by simple truncation of its coordinates (e.g., 3.6 $\rightarrow$ 3, 1.2 $\rightarrow$ 1). Assume schedule S from the set of all possible schedules *Sched*. For the purpose of differential evolution, we define a fitness function $\mathit{fit}(S) : \mathit{Sched} \rightarrow \mathbb{R}$ that evaluates each schedule:

$$\mathit{fit}(S) = \lambda \cdot \mathit{makespan}(S) + (1 - \lambda) \cdot \frac{\mathit{flowtime}(S)}{m} \tag{15}$$

The function $\mathit{fit}(S)$ is a sum of two objectives, the makespan of schedule S and flowtime of schedule S divided by the number of machines m to keep both objectives in approximately the same magnitude. The influence of makespan and flowtime in $\mathit{fit}(S)$ is parametrized by the variable λ. The same schedule evaluation was used also in [7].

5.3.1 Search for Optimal Independent Task Schedules

We have implemented DE for scheduling of independent tasks on the CUDA platform to evaluate the performance and quality of the proposed solution. The GPU implementation was compared to a simple CPU implementation (high-level object-oriented C++ code) and optimized CPU implementation (low-level C code to achieve maximum performance). The optimized CPU implementation was created to provide a fair comparison of performance oriented implementations on the GPU and on the CPU. Optimized CPU and GPU implementations of the DE for scheduling optimization were identical with the exception of the CUDA-C language constructions.

First, the time needed to compute the fitness for the population of DE vectors was measured for all three DE implementations. The comparison of the fitness computation times ON for different population sizes is illustrated in Fig. 7 (note the log scale of both axes).

The GPU implementation was 25.2–216.5 times faster than the CPU implementation and 2.2–12.5 times faster than the optimized CPU implementation of the same algorithm. This, along with the speedup achieved by the parallel implementation of the DE operations, contributes to the overall performance of the algorithm. To compare the GPU based and the CPU based DE implementations for the independent task scheduling, we have used the benchmark proposed in [5]. The benchmark contains several ETC matrices for 512 jobs and 16 machines. The matrices are labeled according to the following properties [5]:

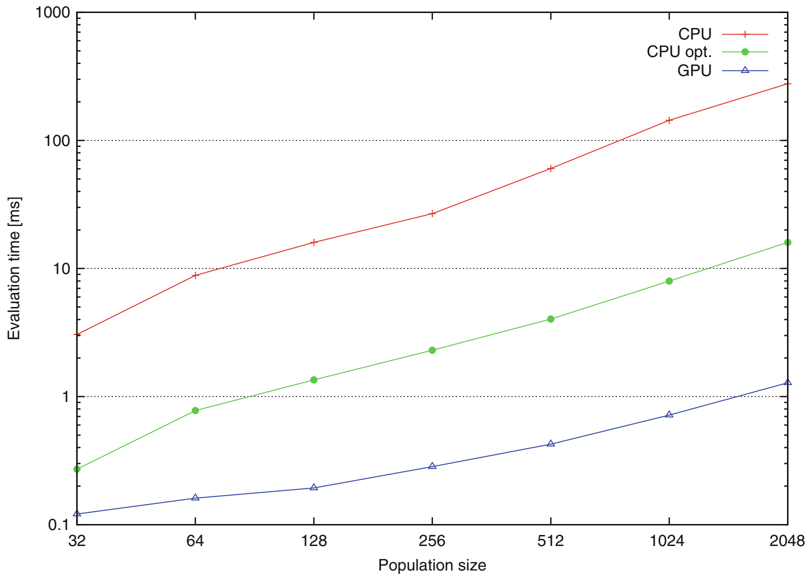


Fig. 7 Comparison of schedule evaluation time on CPU and GPU

- *Task heterogeneity*—V_{task} represents the amount of variance among the execution times of tasks for a given machine.
- *Machine heterogeneity*—V_{machine} represents the variation among the execution times for a given task across all the machines.
- *Consistency*—an ETC matrix is said to be consistent whenever a machine M_j executes any task T_i faster than machine M_k; in this case, machine M_j executes all tasks faster than machine M_k.
- *Inconsistency*—machine M_j may be faster than machine M_k for some tasks and slower for others.

Each ETC matrix is named using the pattern TxMyCz, where x describes task heterogeneity ($h$igh or $l$ow), y describes machine heterogeneity ($h$igh or $l$ow), and z describes the type of consistency ($i$nconsistent, $c$onsistent, or $s$emi-consistent).

We have investigated the speed and quality of the results obtained by the proposed DE implementation and compared it to the results obtained by CPU implementations. Average fitness values of the best schedules found by different DE variants after 30 s are listed in Table 4. The best results for each ETC matrix are shown in bold. We can see that the GPU implementation delivered the best results for population sizes 1024 and 512. However, the most successful population size was 64. Apparently, such a population size seems to be suitable for the investigated scheduling problem with given dimensions (i.e., number of jobs and number of machines). When executing differential evolution with population size 64, the optimized CPU implementation delivered the best results for the consistent ETC

Table 4 The fitness of best schedule found in 30 s using different population sizes (lower is better)

	Population size = 64			Population size = 512			Population size = 1024		
ETC matrix	CPU	CPU opt.	GPU	CPU	CPU opt.	GPU	CPU	CPU opt.	GPU
ThMhCc	1.07E + 7	**9.03E + 6**	9.57E + 6	2.08E + 7	1.69E + 7	**9.46E + 6**	2.42E + 7	1.92E + 7	**9.35E + 6**
ThMhCi	6.60E + 6	3.72E + 6	**3.18E + 6**	1.92E + 7	1.77E + 7	**3.19E + 6**	2.18E + 7	1.96E + 7	**3.29E + 6**
ThMhCs	7.48E + 6	4.89E + 6	**4.27E + 6**	2.02E + 7	1.73E + 7	**4.24E + 6**	2.37E + 7	1.99E + 7	**4.43E + 6**
ThMlCc	194841	**180070**	186913	240260	206585	**188508**	269340	233054	**187939**
ThMlCi	118491	88383.6	**78645.4**	233159	213770	**78905.1**	251670	235113	**80649.6**
ThMlCs	141940	111729	**104012**	233885	205696	**104898**	257279	228244	**108694**
TlMhCc	361021	**322400**	334667	693637	564866	**328479**	787541	666462	**325734**
TlMhCi	219874	123442	**104475**	683699	579198	**104597**	728971	670957	**107532**
TlMhCs	243946	158307	**142704**	644251	567544	**143857**	769772	638295	**149150**
TlMlCc	6387.09	**5908.12**	6185.68	7647.84	6896.78	**6148.65**	9035.75	7804.94	**6155.41**
TlMlCi	3883.62	2813.56	**2540.56**	7882.03	7070.03	**2549.5**	8349	7685.84	**2619.71**
TlMlCs	4640.97	3697.94	**3388.26**	7657.22	6726.06	**3418.79**	8471.38	7669.97	**3581.62**

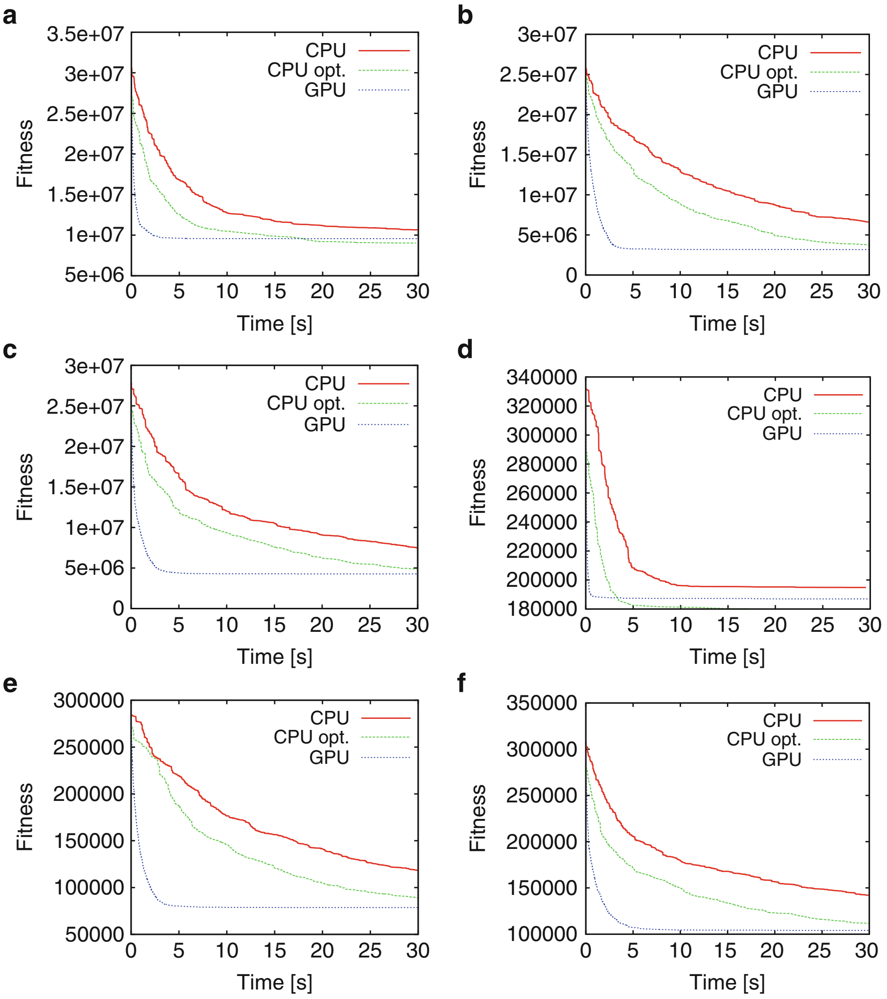


Fig. 8 Fitness improvement of different DE implementations for ThM*C* matrices. (**a**) ThMhCc, (**b**) ThMhCi, (**c**) ThMhCs, (**d**) ThMlCc, (**e**) ThMlCi, (**f**) ThMlCs

matrices, i.e., ThMhCc, ThMlCc, TlMhCc, and TlMlCc. In all other cases, the best result was found by the GPU powered differential evolution.

The progress of DE with the most successful population size 64 for different ETC matrices is shown in Figs. 8 and 9. The figures clearly illustrate the big difference between DE on the CPU and the GPU. DE executed on the GPU achieves the most significant fitness improvement during the first few seconds (roughly 5 s), while the CPU implementations require much more time to deliver solutions with similar quality, if they manage to do it at all. Needless to say, the optimized CPU implementation always found better solutions than the simple CPU optimization because it managed to process more candidate vectors in the same time frame.

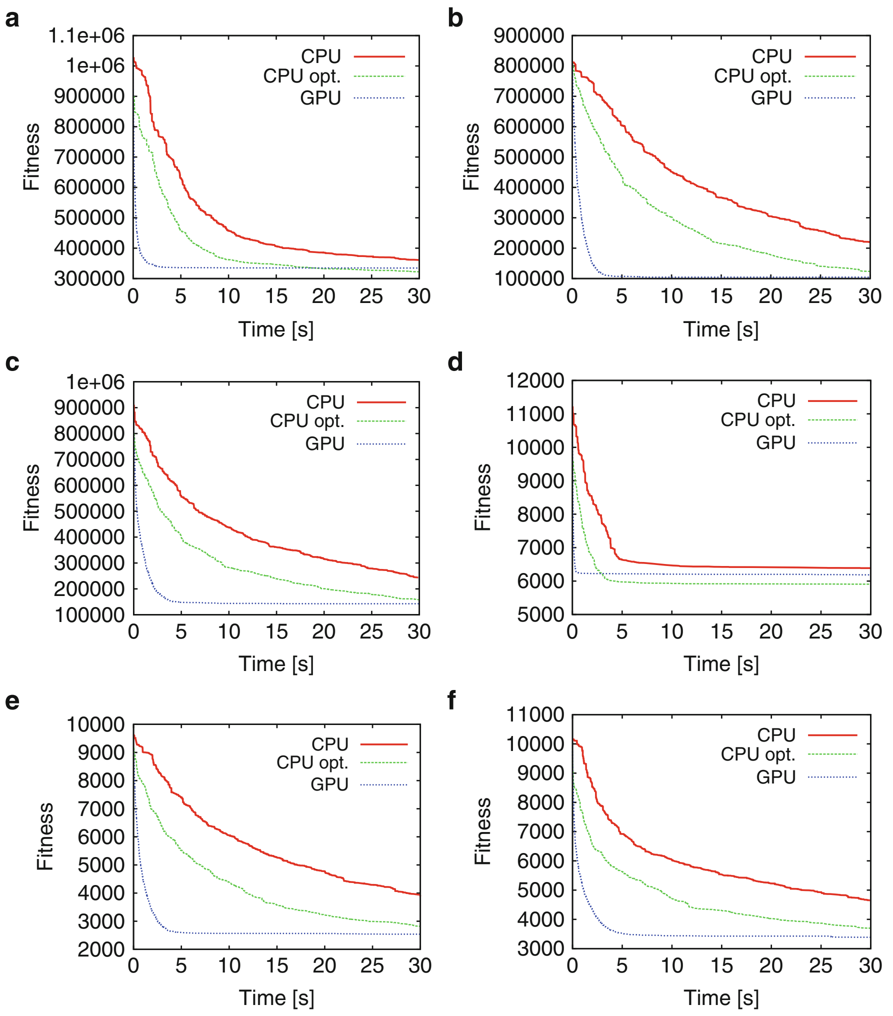


Fig. 9 Fitness improvement of different DE implementations for TlM*C* matrices. (**a**) TlMhCc, (**b**) TlMhCi, (**c**) TlMhCs, (**d**) TlMlCc, (**e**) TlMlCi, (**f**) TlMlCs

6 Conclusions

This chapter described the design and implementation of a fine-grained DE on the GPUs. The basic steps of the algorithm were implemented with respect to the super parallel SIMD architecture of the GPUs allowing efficient parallel execution of hundreds of threads. The fine-grained many-threaded DE design was chosen in order to maximize the utilization of the resources provided by the GPUs.

The performance of the solution was demonstrated on a series of computational experiments. The experimental evaluation involved continuous function optimization to provide a rough comparison with a previous DE design for the GPU and two popular combinatorial optimization problems with real-world applications. The many-threaded DE implemented on the CUDA platform has shown good performance and good results in all test cases.

Acknowledgements This work was supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070) and by the Bio-inspired Methods: Research, Development, and Knowledge Transfer project, reg. no. CZ.1.07/2.3.00/20.0073 funded by the Operational Programme Education for Competitiveness, co-financed by ESF and the state budget of the Czech Republic.

References

1. Ali, S., Braun, T., Siegel, H., Maciejewski, A.: Heterogeneous computing. In: Urbana, J., Dasgupta, P. (eds.) Encyclopedia of Distributed Computing. Kluwer Academic Publishers, Norwell, MA (2002)
2. Ashlock, D.: Evolutionary Computation for Modeling and Optimization. Springer, New York (2006)
3. Baker, J.E.: Reducing bias and inefficiency in the selection algorithm. In: Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms, pp. 14–21. L. Erlbaum Associates Inc., Hillsdale, NJ, USA (1987)
4. Bertacco, L., Brunetta, L., Fischetti, M.: The linear ordering problem with cumulative costs. Eur. J. Oper. Res. **127**(3), 1345–1357 (2008)
5. Braun, T.D., Siegel, H.J., Beck, N., Bölöni, L.L., Maheswaran, M., Reuther, A.I., Robertson, J.P., Theys, M.D., Yao, B., Hensgen, D., Freund, R.F.: A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. J. Parallel Distr. Comput. **61**, 810–837 (2001)
6. Campos, V., Glover, F., Laguna, M., Martí, R.: An experimental evaluation of a scatter search for the linear ordering problem. J. Global Optim. **21**(4), 397–414 (2001)
7. Carretero, J., Xhafa, F., Abraham, A.: Genetic algorithm based schedulers for grid computing systems. Int. J. Innovat. Comput. Inform. Contr. **3**(7) (2007)
8. Das, S., Abraham, A., Konar, A.: Automatic hard clustering using improved differential evolution algorithm. In: Metaheuristic Clustering. Studies in Computational Intelligence, vol. 178, pp. 137–174. Springer, Berlin/Heidelberg (2009)
9. Desell, T.J., Anderson, D.P., Magdon-Ismail, M., Newberg, H.J., Szymanski, B.K., Varela, C.A.: An analysis of massively distributed evolutionary algorithms. In: IEEE Congress on Evolutionary Computation, Barcelona, Spain, pp. 1–8, 18–23 July 2010
10. Engelbrecht, A.: Computational Intelligence: An Introduction, 2nd edn. Wiley, New York, NY, USA (2007)
11. Fernandez-Baca, D.: Allocating modules to processors in a distributed system. IEEE Trans. Software Eng. **15**(11), 1427–1436 (1989)
12. Fujimoto, N., Tsutsui, S.: A highly-parallel TSP solver for a GPU computing platform. In: Dimov, I., Dimova, S., Kolkovska, N. (eds.) Numerical Methods and Applications. Lecture Notes in Computer Science, vol. 6046, pp. 264–271. Springer, Berlin/Heidelberg (2011)
13. Huang, G., Lim, A.: Designing a hybrid genetic algorithm for the linear ordering problem. In: GECCO 2003, Springer, Heidelberg, pp. 1053–1064 (2003)

14. Izakian, H., Abraham, A., Snásel, V.: Comparison of heuristics for scheduling independent tasks on heterogeneous distributed environments. In: Computational Sciences and Optimization, 2009. International Joint Conference on, vol. 1, pp. 8–12 (2009)
15. Krömer, P., Abraham, A., Snášel, V., Platoš, J., Izakian, H.: Differential evolution for scheduling independent tasks on heterogeneous distributed environments. In: Advances in Intelligent Web Mastering - 2. Advances in Soft Computing, vol. 67, pp. 127–134. Springer, Berlin/Heidelberg (2010)
16. Krömer, P., Platoš, J., Snásel, V.: Differential evolution for the linear ordering problem implemented on CUDA. In: Smith, A.E. (ed.) Proceedings of the 2011 IEEE Congress on Evolutionary Computation. IEEE Computational Intelligence Society, pp. 790–796. IEEE Press, New Orleans, USA (2011)
17. Krömer, P., Snásel, V., Platoš, J., Abraham, A.: Many-threaded implementation of differential evolution for the CUDA platform. In: Krasnogor, N., Lanzi, P.L. (eds.) GECCO, ACM, pp. 1595–1602 (2011)
18. Krömer, P., Snásel, V., Platoš, J., Abraham, A., Ezakian, H.: Evolving schedules of independent tasks by differential evolution. In: Caballé, S., Xhafa, F., Abraham, A. (eds.) Intelligent Networking, Collaborative Systems and Applications. Studies in Computational Intelligence, vol. 329, pp. 79–94. Springer, Berlin/Heidelberg (2011)
19. Krömer, P., Snášel, V., Platoš, J., Abraham, A.: Optimization of turbo codes by differential evolution and genetic algorithms. In: HIS '09: Proceedings of the 2009 Ninth International Conference on Hybrid Intelligent Systems. IEEE Computer Society, Washington, DC, USA, pp. 376–381, 2009
20. Krömer, P., Snášel, V., Platoš, J., Abraham, A., Izakian, H.: Scheduling independent tasks on heterogeneous distributed environments by differential evolution. In: Proceedings of the International Conference on Intelligent Networking and Collaborative Systems, INCOS '09. IEEE Computer Society, Barcelona, Spain, pp. 170–174, 2009
21. Langdon, W.B., Banzhaf, W.: A SIMD interpreter for genetic programming on GPU graphics cards. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) Genetic Programming. Lecture Notes in Computer Science, vol. 4971, pp. 73–85. Springer, Berlin/Heidelberg (2008)
22. Martí, R., Reinelt, G.: The Linear Ordering Problem: Exact and Heuristic Methods in Combinatorial Optimization. Applied Mathematical Sciences, vol. 175. Springer, Heidelberg; Dordrecht; London; New York (2011)
23. Martí, R., Reinelt, G., Duarte, A.: A benchmark library and a comparison of heuristic methods for the linear ordering problem. Comput. Optim. Appl. **51**(3), 1–21 (2011)
24. Mitchell, J.E., Borchers, B.: Solving linear ordering problems with a combined interior point/simplex cutting plane algorithm. Technical report, Mathematical Sciences, Rensselaer Polytechnic Institute, Troy, NY 12180–3590 (1997)
25. Munir, E., Li, J.Z., Shi, S.F., Rasool, Q.: Performance analysis of task scheduling heuristics in grid. In: Machine Learning and Cybernetics, 2007 International Conference on, vol. 6, pp. 3093–3098 (2007)
26. NVIDIA. NVIDIA CUDA Programming Guide and link. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Accessed 27 July 2013
27. Page, A.J., Naughton, T.J.: Framework for task scheduling in heterogeneous distributed computing using genetic algorithms. Artif. Intell. Rev. **24**, 137–146 (2004)
28. Pospíchal, P., Jaroš, J., Schwarz, J.: Parallel genetic algorithm on the CUDA architecture. In: Applications of Evolutionary Computation, LNCS 6024, pp. 442–451. Springer, Heidelberg (2010)
29. Price, K.V., Storn, R.M., Lampinen, J.A.: Differential Evolution: A Practical Approach to Global Optimization. Natural Computing Series. Springer, Berlin, Germany (2005)
30. Proakis, J.G.: Digital Communications, 4th edn. McGraw-Hill, New York (2001)
31. Reinelt, G.: The Linear Ordering Problem: Algorithms and Applications. Research and Exposition in Mathematics, vol. 8. Heldermann Verlag, Berlin (1985)

32. Ritchie, G., Levine, J.: A hybrid ant algorithm for scheduling independent jobs in heterogeneous computing environments. In: Proceedings of the 23rd Workshop of the UK Planning and Scheduling Special Interest Group (2004)
33. Robilliard, D., Marion, V., Fonlupt, C.: High performance genetic programming on GPU. In: Proceedings of the 2009 workshop on Bio-inspired algorithms for distributed systems, BADS '09, pp. 85–94. ACM, New York, NY, USA (2009)
34. Roy, S., Izlam, S.M., Ghosh, S., Das, S., Abraham, A., Krömer, P.: A modified differential evolution for autonomous deployment and localization of sensor nodes. In: N. Krasnogor, P.L. Lanzi (eds.) GECCO (Companion), pp. 235–236. ACM, New York, NY, USA (2011)
35. Schiavinotto, T., Stützle, T.: Search space analysis for the linear ordering problem. In: Raidl, G.R., Meyer, J.A., Middendorf, M., Cagnoni, S., Cardalda, J.J.R., Corne, D., Gottlieb, J., Guillot, A., Hart, E., Johnson, C.G., Marchiori, E. (eds.) Applications of Evolutionary Computing. Lecture Notes in Computer Science, vol. 2611, pp. 322–333. Springer, Berlin, Germany (2003)
36. Schiavinotto, T., Stützle, T.: The linear ordering problem: Instances, search space analysis and algorithms. J. Math. Model. Algorithm. **3**(4), 367–402 (2004)
37. Snášel, V., Krömer, P., Platoš, J.: Differential evolution and genetic algorithms for the linear ordering problem. In: Velásquez, J.D., Ríos, S.A., Howlett, R.J., Jain, L.C. (eds.) KES (1). Lecture Notes in Computer Science, vol. 5711, pp. 139–146. Springer, Berlin, Heidelberg (2009)
38. Storn, R.: Differential evolution design of an IIR-filter. In: Proceeding of the IEEE Conference on Evolutionary Computation ICEC, pp. 268–273. IEEE Press, Nagoya, Japan (1996)
39. Storn, R., Price, K.: Differential Evolution: A Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces. Berkeley, CA, Tech. Rep. (1995)
40. de Veronese, L., Krohling, R.: Differential evolution algorithm on the GPU with C-CUDA. In: Evolutionary Computation (CEC), 2010 IEEE Congress on, pp. 1–7 (2010)
41. YarKhan, A., Dongarra, J.: Experiments with scheduling using simulated annealing in a grid environment. In: GRID '02: Proceedings of the Third International Workshop on Grid Computing, pp. 232–242. Springer, London, UK (2002)
42. Zhu, W.: Massively parallel differential evolution—pattern search optimization with graphics hardware acceleration: an investigation on bound constrained optimization problems. J. Global Optim. **50**(3), 417–437 (2011)
43. Zhu, W., Li, Y.: GPU-accelerated differential evolutionary Markov chain Monte Carlo method for multi-objective optimization over continuous space. In: Proceeding of the 2nd Workshop on Bio-inspired Algorithms for Distributed Systems, BADS '10, pp. 1–8. ACM, New York, NY, USA (2010)

Scheduling Using Multiple Swarm Particle Optimization with Memetic Features on Graphics Processing Units

Steven Solomon, Parimala Thulasiraman, and Ruppa K. Thulasiram

Abstract We investigate the performance of a highly parallel Particle Swarm Optimization (PSO) algorithm implemented on the graphics processing unit (GPU). In order to achieve this high degree of parallelism we implement a collaborative multi-swarm PSO algorithm on the GPU which relies on the use of many swarms rather than just one. We choose to apply our PSO algorithm against a real-world application: the task matching problem in a heterogeneous distributed computing environment. Due to the potential for large problem sizes with high dimensionality, the task matching problem proves to be very thorough in testing the GPU's capabilities for handling PSO. Our results show that the GPU offers a high degree of performance and achieves a maximum of 37 times speedup over a sequential implementation when the problem size in terms of tasks is large and many swarms are used.

1 Introduction

A significant problem in a heterogeneous distributed computing environment, such as grid computing, is the optimal matching of tasks to machines such that the overall execution time is minimized. That is, given a set of heterogeneous resources (machines) and tasks, we want to find the optimal assignment of tasks to machines such that the makespan, or time until all machines have completed their assigned tasks, is minimized.

Task matching, when treated as an optimization problem, quickly becomes computationally difficult as the number of tasks and machines increases. In response

S. Solomon (✉) · P. Thulasiraman · R.K. Thulasiram
University of Manitoba, Winnipeg, MB, Canada
e-mail: umsolom9@cs.umanitoba.ca; tulsi@cs.umanitoba.ca; thulasir@cs.umanitoba.ca

to this problem, researchers and developers have made use of many heuristic algorithms for the task mapping problem. Such algorithms include first-come-first-serve (FCFS), min–max and min–min [4], and suffrage [8]. More recently, bio-inspired heuristic algorithms such as Particle Swarm Optimization (PSO) [6] have been used and studied for this problem [21, 22]. The nature of algorithms such as PSO potentially allows for the generation of improved solutions without significantly increasing the costs associated with the matching process.

The basic PSO algorithm, as described by Kennedy and Eberhart [6], works by introducing a number of particles into the solution space (a continuous space where each point represents one possible solution) and moving these particles throughout the space, searching for an optimal solution. While single-swarm PSO has already been applied to the task matching problem, there does not exist, to the best of our knowledge, an implementation that makes use of multiple swarms collaborating with one another.

We target the graphics processing unit (GPU) for our implementation. In recent years, GPUs have provided significant performance improvements for many parallel algorithms. One example comes from Mussi et al. [10]'s work on PSO, which shows a high degree of speedup over a sequential CPU implementation. As the GPU offers a tremendous level of parallelism, we believe that multi-swarm PSO provides a good fit for the architecture. With a greater number of swarms, and, thus, a greater number of particles, we can make better use of the threading capabilities of the GPU.

The rest of this chapter is organized as follows. The next section discusses the CUDA programming model and GPU architecture, followed by a description of the task matching problem in Sect. 3. In Sect. 4 we provide an introduction to single- and multi-swarm PSO, and in Sect. 5, we discuss related work in PSO for task matching, parallel PSO on GPUs, and multi-swarm PSO. We provide the description of our GPU implementation of multi-swarm PSO for task matching in Sect. 6 and follow this up with our performance and solution quality results in Sect. 7. Finally, we detail our conclusions and ideas for future work in Sect. 8.

2 Parallel Computing and the GPU

We start with a discussion on the relevant details of the GPU architecture and CUDA framework. As we implemented and tested all of our work on an Nvidia GTX 260 GPU (based on the GT200 architecture), all information and hard numbers in this section pertains to this particular model. Because the GPU is a relatively new architecture in the general purpose parallel algorithms arena, we start with a brief discussion on the two general categories of parallel systems and where the GPU fits in. We follow this with a discussion on the GPU architecture and CUDA itself and conclude this section with a brief description of a basic parallel algorithm we use in this work: parallel reduction.

2.1 Parallel Systems

We divide the parallel systems used in parallel computing into two camps, homogeneous and heterogeneous. Which camp a parallel system belongs to is based on the processing elements contained within the system. Homogeneous systems are the most common systems and include hardware like the traditional multicore processor which contains a number of equivalent, or symmetrical, cores. Such a system is *homogeneous* in that its processing elements are all the same. Heterogeneous systems, on the other hand, contain processing elements that differ from others within the same system.

The GPU falls under the heterogeneous systems category. While we will explain that the GPU itself is composed of a number of identical processing elements, it, alone, does not compose the entirety of the system. The GPU requires a traditional CPU in order to drive the computational processes we want to execute on it. The GPU, in effect, becomes an accelerator. When present in a system, we design algorithms that the main CPU will schedule for execution on the GPU in order to accelerate tasks on an architecture that may be able to offer improved performance. We need both a CPU and a GPU within a system in order to execute algorithms/code on the GPU, creating our heterogeneous system.

Flynn's taxonomy [3] splits up parallel computing systems into three categories based on their execution capabilities (Flynn actually describes four total categories of computing systems; however, the Single Instruction Single Data category is not a parallel system but a uniprocessing system). The two we are concerned about here are Multiple Instruction Multiple Data (MIMD) and Single Instruction Multiple Data (SIMD). With MIMD, each processing element executes independently of one another. In essence, MIMD allows for each processing element to execute different instructions on different data from one another.

SIMD, on the other hand, involves each processing element executing in lockstep with one another. That is, every processing element executes the same instruction at the same time as one another but executes this instruction on different data. MIMD exploits task-level parallelism (achieving parallelism by executing multiple tasks at one time), while SIMD exploits data-level parallelism (achieving parallelism by taking advantage of repetitive tasks applied to different pieces of data). As we discuss next, the MIMD style of system is very different from that of the GPU, which follows the SIMD paradigm.

2.2 CUDA Framework

Prior to discussing the GPU architecture itself, we will cover some details of the CUDA framework. Nvidia developed CUDA [13], or the Compute Unified Device Architecture, in order to provide a more developer-friendly environment for GPU application development. CUDA acts as an extension to the C language, providing

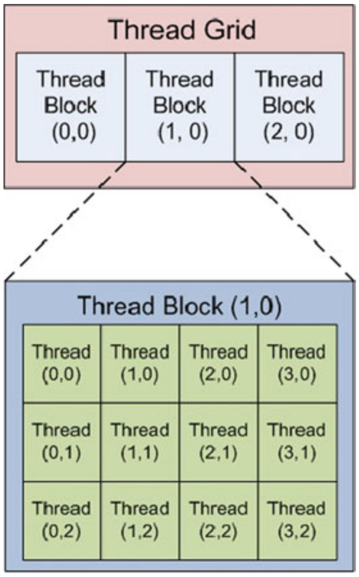


Fig. 1 Two-dimensional organization of thread grid and thread blocks [11]

access to all of the threading, memory, and helper functions that a developer requires when working with the GPU for general purpose applications.

The GPU hardware provides us with a tremendous level of exploitable parallelism on a single chip. Not only does a standard high-end GPU contain hundreds of processing cores, but the hardware is designed to support thousands, hundreds of thousands, even *millions* of threads being scheduled for execution. CUDA provides a number of levels of thread organization in order to make the management of all these threads simpler. At the top level of the thread organization we have the thread grid. The *thread grid* encompasses all threads that will execute our GPU application, otherwise referred to as a kernel. To get to the next level down, the *thread block*, we split up the threads in the thread grid into multiple, equal-sized blocks. The user specifies the organization of threads within a thread block and thread blocks within a grid. What this means for a thread block is that we may organize and address threads in a one-, two-, or three-dimensional fashion. The same holds true for thread blocks within a grid; the user specifies one-, two-, or three-dimensional organization of the blocks composing the thread grid. Figure 1 provides an example of two-dimensional organization of a thread grid and thread blocks.

At the lowest level of the thread organization we have the *thread warp*. Equal-sized chunks of threads from a thread block form the thread warps for that block. Unlike the size or dimensions of a block/grid, the hardware specifications determine the size of a warp, and the threads are ordered in a one-dimensional fashion. For the GT200 (and earlier) architecture, 32 threads form a warp. The hardware issues each thread within a warp the same instruction to execute, regardless of whether or not all 32 threads have to execute it (we discuss this concept further in Sect. 2.3). When branching occurs, threads which have diverged are marked as inactive and do not execute until instructions from their path of the branch are issued to the warp. Algorithms for GPUs should therefore reduce branching or ensure that all threads in a warp will take the same path in a branch in order to maximize performance.

Typically, a parallel application will involve some degree of synchronization. Synchronization is the act of setting a barrier in place until some (or perhaps all) threads reach the barrier. This ensures that the threads in question will all be at the same step in the algorithm immediately after the synchronization point. CUDA provides a few mechanisms for synchronization based around the thread warp, block, and grid. First, each thread in a warp is always synchronized with all the other threads in that same warp as they all receive the same instruction to execute. Secondly, CUDA provides block-level synchronization in the form of an instruction. By using the `__syncthreads()` instruction, threads reaching the instruction will wait until all threads in the thread block have also hit that point.

Unfortunately, CUDA does not provide any mechanisms within a kernel to synchronize all threads in a grid. As a result, we must complete execution of the kernel and rely on the CPU to perform the synchronization. CUDA provides two methods for accomplishing this:

1. Launching another kernel—after invoking one kernel, attempting to launch another will result in the CPU application halting until the previous kernel has completed execution (effectively "synchronizing" all threads in the thread grid, as they must all have completed execution of the first kernel).
2. Using the `cudaThreadSynchronize()` instruction in the CPU application—essentially the same as the above but explicitly controlled by the user. Again, the CPU application will halt here until the previous kernel has completed execution.

2.3 GPU Architecture

With the introductory CUDA material covered, we move on to a description of the GPU architecture itself. We begin with the GPU as a whole, which is composed of two separate units: the core and the off-chip memory. The GPU core itself contains a number of Streaming Multiprocessors or SMs. As pictured in Fig. 2, each SM contains eight CUDA cores or CCs. These CCs are the computational cores of the GPU and handle the execution of instructions for the threads executing within the SM. SMs also contain a multi-threaded instruction dispatcher and two Special Function Units (SFUs) that provide extra transcendental mathematic capabilities.

Execution of instructions on each SM follows a model similar to SIMD, which Nvidia [11] refers to as SIMT or Single Instruction Multiple Threads. In SIMT, the hardware scheduler first schedules a warp for execution on the CCs of an SM. The hardware then assigns the same instruction for execution across all threads in the chosen warp—only the data each instruction acts on is changed. This threading model implies that all threads in a warp are issued the same instruction, regardless

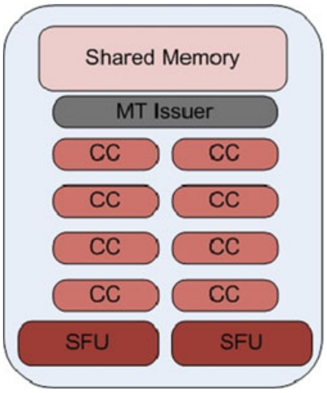


Fig. 2 General layout of a streaming multiprocessor

of whether or not every thread needs to execute that instruction. Consider the case where the threads encounter a branch: half of the threads in a warp take path A in the branch, the other take path B. With SIMT, the hardware will issue instructions for path A to all threads in the warp, even those that took path B. This represents an important concept as threads in a warp *diverging* across different paths in a branch results in a loss of parallelism—each branch is essentially executed serially, rather than in parallel. That is to say, rather than having 32 threads performing useful work, only a subset of the threads do work for path i, while the remaining threads idle, waiting for instructions from their own path.

Reaching back to our knowledge of the CUDA framework, we see a connection between thread blocks and SMs. All of the threads within a thread block must execute entirely within a single SM. This means that we (or the hardware) cannot split up the threads in a thread block between multiple SMs. Multiple thread blocks, however, may execute on a single SM if that SM has enough resources to support the requirements of more than one thread block.

Aside from the computational units, each SM also contains 16 kilobytes of shared memory. This shared memory essentially acts as a developer-controlled cache for data during kernel execution. As a result, the responsibility is on the developer to place data into this memory space—there does not exist any automatic hardware caching of data (Nvidia changed this in their Fermi [11] architecture, which introduced a hardware-controlled cache at each SM). Nvidia [12] claims that accesses to shared memory are up to 100 times faster than global memory, given no bank conflicts. As shared memory is split into 16 32-bit wide banks, multiple requests for data from the same bank arriving at the same time cause bank conflicts and, as a result, are serialized. In effect, bank conflicts reduce the overall throughput of shared memory, as some threads must wait for their requested data until the shared memory has serviced the requests from previous threads. Shared memory is exclusive to each thread block executing on a given SM. That is to say, a thread block cannot access the shared memory data from another thread block, even if it is executing on the same SM.

Moving on to the other memory systems present within the GPU, we have the global memory. Global memory is the largest memory space available on the GPU and is read/write accessible to all threads. Unfortunately, a significant latency, measured by Nvidia [11] at approximately 400–800 cycles, occurs for each access to global memory. Global memory accesses are not cached at any level, and as a result, every access to global memory incurs this hefty latency hit. The GPU contains, however, some auxiliary memory systems that *are* cached at the SM level. Each SM has access to caches for the constant and texture memory of the GPU. While these two memories are still technically part of global memory (that is, data stored in these memories are stored in the global memory space), their caches help to reduce the latency penalty by exploiting data locality.

Based on what we have learned about the memory systems within the GPU, we clearly want to place an emphasis on exploiting shared memory as much as possible. With fast access speed and no significant dependencies on data locality to mitigate high latencies, shared memory represents the most optimal location for storage. Unfortunately, we run into many situations where the small size of shared memory results in insufficient storage space for the data required at a given moment in time.

While global memory clearly represents a major area of performance loss due to latency, there is one important technique we can use to mitigate the damage: *global memory coalescing*. In order to understand how coalescing works, we must first revisit the idea of a warp. As we described earlier, a thread warp is composed of 32 threads, all of which are given the same instruction to execute. In the worst case, we would expect there to be 32 individual requests to global memory if the instruction in question requires data from global memory. With coalescing, however, we have the ability to reduce the total number of requests down to only two requests in the best case. The reason for this lies with how memory requests are handled at the warp level: they are performed in a *half*-warp fashion. That is, 16 threads request data from memory first, followed by the remaining 16 shortly thereafter. As a result, the best scenario for coalescing combines all memory requests from each half warp.

In the GT200 architecture, coalescing occurs when at least two threads in one half of a warp are accessing from the same segment in global memory. This technique is very powerful and leads to tremendous improvements in the performance of global memory. Unfortunately, data access patterns must be very structured in order to ensure threads will access data from the same memory segment as one another, something that is not guaranteed when working with irregular algorithms. The easiest way to achieve this result is ensuring that each thread accesses data from global memory that is one element over from the previous thread's access location. As we will show, we make use of coalescing as much as possible in our algorithms in order to achieve greater memory performance.

We close this part off with a brief discussion on the isolation between thread blocks enforced by CUDA. Recall that shared memory is exclusive to a thread block—other threads in other blocks cannot access the shared memory allocated by the block. Furthermore, there are no built-in mechanisms for communication or synchronization between thread blocks. Of course, the availability of global

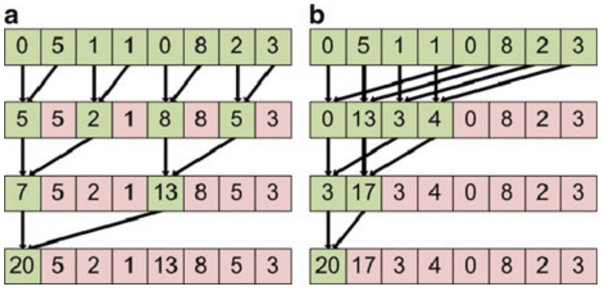


Fig. 3 Two styles of parallel add reduction on an array of elements

memory means that there will always exist a method for communication if desired. The latency of global memory coupled with the overhead of potentially thousands (if not more) of threads accessing a single data element (say, as a synchronization flag) results in an entirely unacceptable solution with a tremendous degree of performance degradation. All of these items combine to show one of the main tenets of CUDA: thread blocks are isolated units of computation. Threads within a thread block communicate with one another, but they cannot readily communicate with other thread blocks.

2.4 Parallel Reduction

As we discuss the use of parallel reductions in our work, we provide a brief description here. A parallel reduction involves reducing a set of values into a single result, in parallel. More formally, given a set of values, $v_1, v_2, \ldots, v_n$, we apply some associative operator, $\oplus$, to the elements, $v_1 \oplus v_2 \oplus \ldots \oplus v_n$, resulting in a single value, w. We consider the structure of a parallel reduction to be that of a binary tree. At the leaf nodes, we have the original set of values. We apply $\oplus$ to each pair of leaf nodes stemming from a parent node one layer up the tree (closer to the root node), giving us the value for that parent node. We repeat this process until we reach the root node, providing us with the final result, w.

When we want to parallelize this reduction technique, we first note that layer i of the tree (where the root node is layer 0 and the leaves layer $\log n$) requires the accumulated partial solution values from layer $i + 1$. This requirement results in synchronization; we can compute one layer of the tree in parallel, but we must wait for all threads to complete their processing of nodes in that layer before moving on to the next. Figure 3a provides an example of performing an addition reduction (that is, $\oplus = +$) in parallel. Each subsequent array represents the next layer of computations. In the first layer, we have the initial array. In parallel, we add up each pair of elements (four parallel operations in total) and place the results back into the array. In the next layer, we have two remaining parallel operations which we use to add up the partial sums from the previous layer. Finally, we add the remaining two partial sums together and end up with the final result in element zero of the array which contains the total sum of all initial values. On the GPU, we typically perform a parallel reduction within a single thread block if possible. This allows us to use thread block-level synchronization rather than the more costly thread grid-level synchronization. As parallelism is plentiful on the GPU, we assign one thread per node in the current layer of the tree. To further optimize this algorithm for the GPU, we do not use the interleaving method shown in Fig. 3a but, rather, split the layer into halves and work on one side (pulling data from the other). We show this technique in Fig. 3b. By working on contiguous areas/halves we ensure coalescing takes place as we read data from global memory, and bank conflicts do not occur as we read data from shared memory.

3 The Task Matching Problem

The task matching problem represents a significant problem in heterogeneous, distributed computing environments, such as grid or cloud computing. The problem involves determining the optimal assignment, or *matching*, of tasks to machines such that the total execution time is minimized. More specifically, if we are given a set of heterogeneous computing resources and a set of tasks, we want to match (assign) tasks to machines such that we optimize the time taken until all machines have completed processing of their assigned tasks. We refer to this measure of time that we look to optimize as the makespan, which is determined by the length of time taken by the last machine to complete its assigned tasks.

We provide an example of one (suboptimal) solution to a task matching problem instance in Fig. 4. In this case, we have three resources (machines) and six tasks. Each task in the figure is vertically sized based on the amount of time required to execute the task (we assume all three machines are equal in capabilities for this example). In the solution provided, machine three defines the makespan, as it will take the longest amount of time to complete. Of course, this solution is suboptimal, as we could move task six to machine two in order to generate an improved solution. In this improved solution, machine three still defines the makespan, but the actual value will be smaller, as only task four needs processing, rather than four and six.

While a toy problem such as the one in Fig. 4 may not seem particularly intensive, the task matching problem becomes very computationally intensive as the problem size scales upwards. With many machines, and even more tasks, the possible combinations of task to machine matchings become extraordinarily high. Rather than a brute-force approach, we need more intelligent algorithms to solve this problem within a reasonable amount of time.

While we will discuss the PSO-related solutions to this problem in Sect. 5.3, we will conclude this section with a brief look at some of the simpler heuristic algorithms developed for solving the task matching problem. The simplest of these is the FCFS algorithm that simply matches each task to the currently most optimal

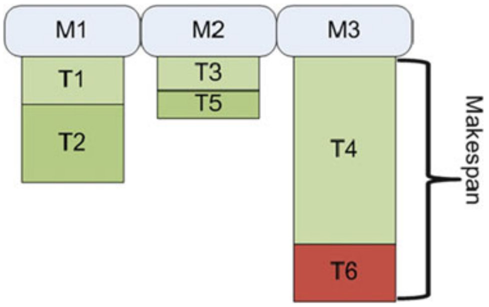


Fig. 4 Example of task matching and makespan determination

machine. The algorithm will choose an optimal machine based on the current machine available time (MAT) of each machine. The MAT of a machine is the amount of time required to complete all tasks currently matched to that machine. The algorithm assigns the task to the machine with the lowest MAT.

Two more heuristics for solving the task matching problem are the min–max and min–min [4] algorithms. The min–min algorithm first determines the minimum completion time for each task that we want to consider across each machine. Within these minimum completion time values, it searches for the *minimum* and assigns that task to the corresponding machine. The min–max algorithm handles this problem in a slightly opposite manner. The algorithm still computes the minimum completion times, but rather than assigning the task with the minimum value to the corresponding machine, it assigns the task with the maximum value.

The issue with these two algorithms is that they are suited for very particular instances of the problem. Min–min, for example, works well with many small tasks, as it will assign them to their optimal machines first, leaving the few longer tasks for last. This algorithm may improve the makespan for problems with many small tasks, but as the number of larger tasks increases, the results worsen. Min–max, on the other hand, sees improved performance when the problem instance contains many longer tasks. Neither of these problems provides optimal solutions across all potential cases. PSO, on the other hand, searches for optimal solutions through the solution space and may work effectively regardless of the task composition.

4 Particle Swarm Optimization

The PSO algorithm, first described by Kennedy and Eberhart [6], is a bio-inspired or meta-heuristic algorithm that uses a *swarm* of *particles* which move throughout the solution space, searching for an optimal solution. A point in the solution space (defined by a real number for each dimension) represents a solution to the optimization problem. As the particles move, they determine the optimality (fitness) of these positions.

The PSO algorithm uses a fitness function in order to determine the optimality of a position in the solution space. Each particle stores the location of the best (most optimal) position it has found thus far in the solution space (local best). Particles collaborate with one another by maintaining a global, or swarm, best position representing the best position in the solution space found by all particles thus far.

In order to have these particles move throughout the solution space they must be provided with some velocity value. In this chapter, we follow the modified PSO algorithm as established by Shi and Eberhart [15]. These authors update the velocity of a particle, i, with the following equation:

$$V_{i+1} = w * V_i + c_1 * \text{rnd}() * (X_{\text{Pbest}} - X_i) + c_2 * \text{rnd}() * (X_{\text{Gbest}} - X_i) \qquad (1)$$

where X_i is the particle's current location, X_{Pbest} is the particle's local best position, X_{Gbest} is the global best position (for one swarm), and rnd() generates a uniformly distributed random number between 0 and 1. w, the inertial weight factor along with c_1 and c_2, provides some tuning of the impact V_i, X_{Pbest}, and X_{Gbest} will have on the particle's updated velocity. Once the velocity has been updated, the particle changes its position, and we begin the next iteration.

Algorithm 1 provides the basic high-level form of PSO.

Algorithm 1 Basic PSO algorithm

```
Randomly disperse particles into solution space
for $i = 0 \rightarrow$ numIterations do
    for all particles in swarm do
        Compute fitness of current location
        Update $X_{\text{Pbest}}$ if necessary
        Update $X_{\text{Gbest}}$ if necessary
        Update velocity
        Update position
    end for
end for
```

We note that the PSO algorithm is an iterative, synchronous algorithm: each iteration has the particles moving to a new location and testing the suitability of this new position, and each phase (or line in Algorithm 1) carries implicit synchronization.

As we wanted to investigate the suitability of the GPU for PSO, we needed to think in terms of high degrees of parallelism. We consider a PSO variant that collaborates amongst multiple swarms in order to increase the overall parallelism. Furthermore, we hypothesized that such a variant of PSO may provide higher quality solutions than we would otherwise generate with a single swarm.

The method we choose, described by Vanneschi et al. [19], collaborates amongst swarms by swapping some of a swarm's "worst" particles with its neighboring swarm's "best" particles. In this case, best and worst refer to the fitness of the particle relative to all other particles in the same swarm. This swap occurs every given number of iterations and forces communication among the swarms, ensuring that particles are mixed around between swarms. Further, Vanneschi et al. [19] use a repulsion factor for every second swarm. This repulsive factor repulses particles away from another swarm's global best position (X_{FGBest}) by further augmenting the velocity using the equation:

$$V_{i+1} = V_{i+1} + c_3 * \mathrm{rnd}() * f(X_{\mathrm{FGbest}}, X_{\mathrm{Gbest}}, X_i) \tag{2}$$

where function f, as described by Vanneschi et al. [19], provides the actual repulsion force. We believe that this algorithm represents a good fit for the GPU, as it combines the potential for high degrees of parallelism with the iterative, synchronous nature of the PSO algorithm.

5 Related Work

As this section deals with a few areas that can be considered independently, we split the related work into a few subsections: multi-swarm PSO, PSO for the GPU, and bio-inspired algorithms targeted at the task matching problem. We investigate the existing work for each of these areas independently.

5.1 Multi-swarm PSO

The literature contains a number of works based around multi-swarm PSO. One such work by Liang and Suganthan [7] acts as a modification to a dynamic multi-swarm algorithm. The dynamic multi-swarm algorithm initializes a small number of particles in each swarm and then randomly moves particles between swarms after a given number of iterations. The authors augment this algorithm by including a local refining step. This step occurs every given number of iterations and updates the local best of a particle only if it is within some threshold value relative to the other particles in the swarm.

A different work by van den Bergh and Engelbrecht [18] considers having each swarm optimize only one of the problem's dimensions, and the authors provide a number of collaborative PSO variants. The authors showed that their algorithm provides better solutions as the number of dimensions increases. Compared to a genetic algorithm, van den Bergh and Engelbrecht experimentally show that their collaborative PSO algorithms consistently perform better in terms of solution quality. They further compare their algorithms against a standard PSO algorithm and show that the collaborative PSO algorithms beat this standard algorithm four times out of five. The authors mention, however, that if multiple dimensions are correlated, they should be packed within a single swarm.

As was previously mentioned, we follow the work described by Vanneschi et al. [19] for our implementation on the GPU. Their "MPSO" algorithm solves an optimization problem via multiple swarms that communicate by moving particles amongst the swarms. Every given number of iterations swarms will move some of their best particles to a neighboring swarm, replacing some of the worst particles in that swarm. They describe a further addition to this algorithm, "MRPSO," that further uses a repulsive factor on each particle. Their results show that both MPSO and MRPSO typically outperform the standard PSO algorithm, with MRPSO providing improved performance over MPSO.

5.2 PSO on the GPU

To the best of our knowledge, there does not exist any collaborative, multi-swarm PSO implementations on the GPU in the literature. Veronese and Krohling [2] describe a simple implementation of PSO on the GPU. Their implementations use a single swarm and split the major portions of PSO into separate kernels, with one thread managing each particle. In order to generate random numbers Veronese and Krohling [2] use a GPU implementation of the Mersenne Twister pseudorandom number generator. When executed against benchmark problems the authors show up to an approximately 23 times speedup compared to a sequential C implementation when using a large number of particles (1,000).

Similar to the work of Veronese and Krohling, Zhou and Tan [23] also describe a single-swarm PSO algorithm for the GPU. The authors, too, assign one thread to manage one particle and split up the major phases of PSO into individual GPU kernels. For random number generation, however, Zhou and Tan [23] differ from Veronese and Krohling [2] in that they use the CPU to generate pseudorandom numbers and transfer these to the GPU. The authors achieve up to an 11 times speedup compared to a sequential CPU implementation. They take care to note, however, that they used a midrange GPU for their tests, and they expect the results to be improved further on more powerful GPU hardware.

Mussi et al. [9] investigate the use of PSO on the GPU for solving a real-world problem: road-sign detection. When updating the position and velocity of the particle, the authors map threads to individual elements/dimension values and not a particle as a whole. Similarly, multiple threads within a block collaborate to compute the fitness value of each particle during the fitness update phase. Mussi et al. show that their GPU implementation achieves around a 20 times speedup compared to a sequential CPU implementation.

Mussi et al. [10] provide another, more recent GPU implementation of PSO. As with their earlier work in [9], the authors assign a single thread to a single dimension for each particle. Mussi et al. [10] test their algorithm against benchmarking problems with up to 120 dimensions and show that the parallel GPU algorithm outperforms a sequential application. Finally, the authors mention in passing the ability to run multiple swarms but do not elaborate or test such situations.

The general theme across the works we have covered has been parallelizing single-swarm PSO (with, perhaps, a brief mention of multi-swarm PSO but no actual descriptions of the work). In the most recent case, Mussi et al. [10] provided a fine-grained implementation of PSO that attempts to take advantage of the massive threading capabilities of the GPU. The authors, however, only run test sizes up to 120 dimensions and 32 particles. For our work, we wished to test across not only a larger number of dimensions but a large number of particles as well. As a result, we use a mixed strategy that does not lock a static responsibility to a thread and, further, provides support for multiple swarms that collaborate with one another.

5.3 Evolutionary Computing for Task Matching

Applying evolutionary or bio-inspired algorithms to the task matching/mapping problem has been studied in the past by various groups. For example, Wang et al. [20] investigate the use of genetic algorithms for solving the task matching and scheduling problem with heterogeneous resources. They show that their algorithm is able to find optimal solutions for small problem sets and outperform simpler (non-evolutionary) heuristics when faced with larger problem instances. Chiang et al. [1] later discuss using ant colony optimization to solve the task matching and scheduling problem and show that their algorithm provides higher quality solutions than genetic algorithm from Wang et al. [20].

A number of previous works have investigated both continuous and discrete PSO for solving the task matching problem. All of the works we discuss here share an idea in common: they work in an n dimension solution space, where n is equal to the number of tasks. One dimension maps to one task, and a location along a dimension (typically, but not always) represents the machine that the task is matched to.

To start, Zhang et al. [22] apply the continuous PSO algorithm to the task mapping problem. In their implementation, the authors use the Smallest Position Value (SPV) technique (described by Tasgetiren et al. [17]) in order to generate a position permutation from the location of the particles. Hence, the solution by Zhang et al. does not directly map a location in a dimension to a matching but rather uses the locations to generate some permutation of matchings. Zhang et al. benchmark their algorithm against a genetic algorithm and show that PSO provides superior performance.

A recent work by Sadasivam and Rajendran [14] also considers the continuous PSO algorithm coupled with the SPV technique. The authors focus their efforts on providing load balancing between grid resources (machines), thus adding another layer of complexity into the problem. Unfortunately, the authors only compare their PSO algorithm to a randomized algorithm and show that PSO provides superior solution quality.

Moving away from continuous PSO, Kang et al. [5] experimented with the use of discrete PSO for matching tasks to machines in a grid computing environment. They compared the results of their discrete PSO implementation to continuous PSO, the min–min algorithm, as well as a genetic algorithm. The authors show that discrete PSO outperforms all of the alternatives in all test cases. Shortly thereafter, Yan-Ping et al. [21] described a similar discrete PSO solution with favorable results compared to the max–min algorithm. Both sets of authors, however, test with very small problem sizes—equal to or below 100 tasks.

Our work described here follows our previous work from Solomon et al. [16].

6 Collaborative Multi-swarm PSO on the GPU

To lead into the description of our GPU implementation, we will first describe the mapping of the task matching problem to multi-swarm PSO without considering the GPU architecture. From this groundwork we can then move on to discuss the specifics of the GPU version itself.

To begin with, we define an instance of the task mapping problem as being composed of two distinct components:

1. The set of tasks, T, to be mapped
2. The set of machines, M, which tasks can be mapped to

A task is defined simply by its length or number of instructions. A machine is similarly defined by nothing more than its MIPS (millions of instructions per second) rating. The problem size is therefore defined across two components:

1. The total number of tasks, $|T|$
2. The total number of machines, $|M|$

A solution for the task matching problem consists of a vector, $V = (t_0, t_1, \ldots, t_{|T|-1})$, where the value of t_i defines the machine that task i is assigned to. From V, we compute the makespan of this solution. The makespan represents the maximum MAT of the solution. Ideally, we want to find some V that minimizes the makespan of the mapping.

We use an estimated time to complete (ETC) matrix to store lookup data on the execution time of tasks for each machine. An entry in the ETC matrix at row i, column j, defines the amount of time machine i requires to execute task j, given no load on the machine. While the ETC matrix is not a necessity, the reduction in redundant computations during the execution of the PSO algorithm makes up for the (relatively small) additional memory footprint. We will, however, have to take into consideration the issues with memory latency when we investigate how to store and access the ETC matrix from the GPU.

Similar to the work described in Sect. 5.3, each task in the problem instance represents a dimension in the solution space. As a result, the solution space for a given instance contains exactly $|T|$ dimensions. As any task may be assigned to any machine in a given solution, each dimension must have coordinates from 0 to $|M| - 1$.

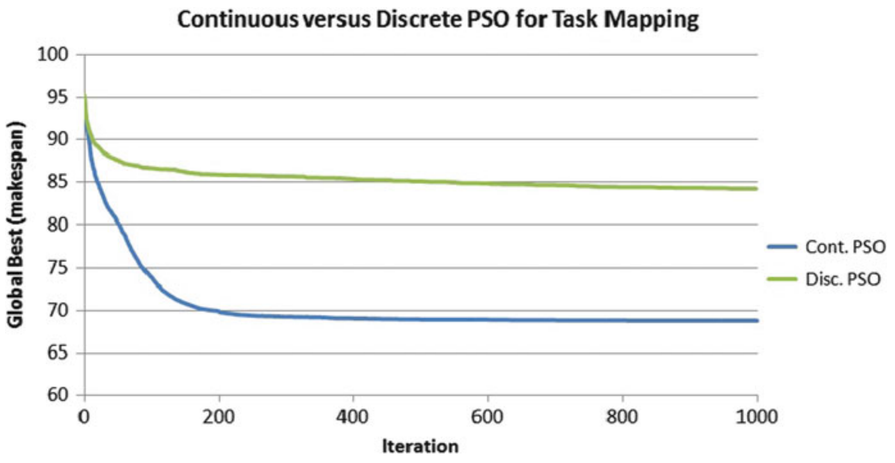


Fig. 5 Global best results for continuous and discrete PSO by iteration

At this point, we deviate from the standard PSO representation of the solution space. Typically, a particle moving along dimension x moves along a continuous domain: any possible point along that dimension represents a solution along that dimension. Clearly, this is not the case for task mapping as a task cannot be mapped to machine 3.673 but, rather, must be mapped to machine 3 or 4. Unlike Kang et al. [5] or Yan-Ping et al. [21], we do not move to a modified discrete PSO algorithm but maintain the use of the continuous domain in the solution space. We compared simple, single-swarm implementations of continuous versus discrete PSO and found that continuous provides improved results, as shown in Fig. 5. However, unlike Zhang et al. [22] or Sadasivam and Rajendran [14], we do not introduce an added layer of permutation to the position value by using the SPV technique. Rather, we use the much simpler technique of rounding the continuous value to a discrete integer.

6.1 Organization of Data on the GPU

We begin the description of our GPU implementation with a discussion on data organization. For our GPU PSO algorithm, we store all persistent data in global memory. This includes the position, velocity, fitness, and current local best value/position for each particle, as well as the global best value/position for each swarm. We also store a set of pre-generated random numbers in global memory. We store each of these sets of data in their own one-dimensional array in global memory.

For position, velocity, and particle/swarm best positions, we store the dimension values for the particles of a given swarm in a special ordering. Rather than order the data by particle, we order it by dimension. Figure 6 provides an example of how this data is stored (swarm best positions are stored per swarm, rather than per

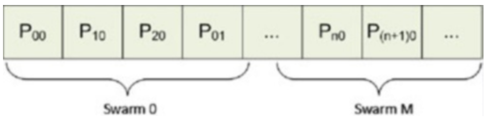


Fig. 6 Global memory layout of position, velocity, and particle best positions (P_{xy} refers to particle x's value along dimension y)

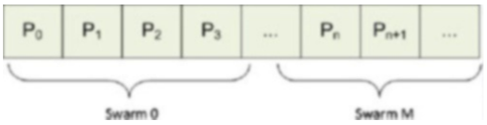


Fig. 7 Global memory layout of fitness and particle best values

particle, however). In a given swarm, we store all of dimension 0's values for each particle, followed by all of dimension 1's values, and so on. We will explain this choice further in Sect. 6.2; however, it is suffice to say for now that this ensures we maintain coalesced accesses to global memory for this data. Per-particle fitness values as well as particle best and swarm best values are stored in a much simpler, linear manner with only one value per particle (or swarm, in the case of the swarm best values). Figure 7 shows this organization.

Outside of the standard PSO data, we know that we also need to store the ETC matrix on the GPU as well. All threads require access to the ETC matrix during the calculation of a particle's fitness (makespan), and they perform the accesses in a very non-deterministic fashion based on their current position at a given iteration. To compute the makespan, each particle must first add to the execution time of tasks assigned to each machine. This is, of course, handled by observing the particle's position along each dimension. As dimensions map to tasks, we are looping through each of the *tasks* and determining which *machine* this particular solution is matching them to. As there are likely to be many more tasks than machines in the problem instance, there will likely be many duplicate reads to the ETC matrix by various threads.

To help improve the performance of reads to the ETC matrix, we place the matrix into texture memory. As discussed in Sect. 2.3, texture memory provides a hardware cache at the SM level. As it is very likely that there will be multiple reads to the same location in the ETC matrix by different threads, using a cached memory provides latency benefits as threads may not have to go all the way to global memory to retrieve the data they are requesting. In fact, if we did nothing but store the ETC matrix in global memory, we may as well just perform the redundant computations instead, as the latency associated with global memory may very well outweigh the computational savings.

6.2 *GPU Algorithm*

We lead into our description of the GPU implementation by first discussing the issue of random number generation. As we know from 1, PSO requires random numbers for each iteration. In order to generate the large quantity of random numbers required, we make use of the CURAND library included in the CUDA Toolkit 3.2 [11] to generate high-quality, pseudorandom numbers on the GPU. We generate a large amount of random numbers at a time (250 MB worth) and then generate more numbers in chunks of 250 MB or less when these have been used up.

Our implementation of multi-swarm PSO on the GPU is split up into a series of kernels that map to the various phases of the algorithm. These phases and kernels are as follows:

6.2.1 Particle Initialization

This phase initializes all of the particles by randomly assigning them a position and a velocity in the solution space. As each dimension of each particle can be initialized independently of one another, we assign multiple threads to each particle: one per dimension. All of the memory writes are performed in a coalesced fashion, as all threads write to memory locations in an ordered fashion.

6.2.2 Update Position and Velocity

This phase updates the velocity of all particles using 1 and then moves the particles based on this velocity. As was the case with particle initialization, each dimension can be handled independently. As a result, we again assign a single thread to handle each dimension of every particle. In the kernel, each thread updates the velocity of the particle's dimension it is responsible for and then immediately updates the position as well. When updating the velocity using 1 and 2, one may note that all threads covering particles in the same swarm will each access the same element from the swarm best position in global memory. While this may seemingly result in uncoalesced reads, global memory provides broadcast functionality in this situation, allowing this read value to be broadcast to all threads in the half warp using only a single transaction.

6.2.3 Update Fitness

Determining the fitness of a particle involves computing the makespan of its given solution. In order to accomplish that, we must first determine the MAT of each machine for the given solution. When computing the MAT we must read from the ETC matrix at a location based on the task-machine matching. We do not know ahead of time which tasks will be assigned to which machines. As a result, we cannot guarantee any structure in the memory requests, and we cannot ensure coalescing.

One option for parallelization involves having a thread compute the makespan for a single machine and then perform a parallel reduction to find the makespan for each particle. The issue with this approach, however, is that a particle's position vector is ordered by task, not by machine. We do not and cannot know which tasks are assigned to which machine ahead of time. If we parallelized this phase at the MAT computation level, then all threads would have to iterate through all of the dimensions of a particle's position anyways, in order to find the tasks matched to the machine the thread is responsible for. As a result, we choose to take the coarser-grained approach and have each particle compute the makespan for a given particle.

We implement two different kernels in order to accomplish this coarser-grained approach. With the first approach, we use shared memory as a scratch space for computing the MAT for each machine. Each thread requires $|M|$ floating-point elements of shared memory. Due to the small size of shared memory, however, larger values of $|M|$ (the exact value is dependent on the number of particles in a swarm) require an amount of shared memory exceeding the capabilities of the GPU. To solve this, we develop a second, less optimal kernel, where we use global memory for scratch space. Given only one thread block executing per SM, the first kernel can support 128 threads (particles) with a machine count of 30, whereas the second kernel supports any value beyond that.

The second, global memory kernel shows our reasoning for choosing the ordering of position elements in global memory (Fig. 6). When computing the makespan, each thread reads the position for its particle in the current dimension being considered in order to discover the task-machine matching. All the threads within a thread block work in lockstep with one another and, thus, work on the same dimension at the same time. The threads within a thread block, therefore, read from a contiguous area in global memory and exploit coalescing. This coalescing results in an approximate 200 % performance improvement over an uncoalesced version based on our brief performance tests.

6.2.4 Update Best Values

This phase updates both the particle best and global best values. We use a single kernel on the GPU and assign a single thread to each particle, as we did with the fitness updating. As was the case with previous phases, we assign all threads covering particles in the same swarm to the same thread block. The first step of this kernel involves each thread determining if it must replace its particle's local best position, by comparing its current fitness value with its local best value. If the current fitness value is greater, then the thread replaces its local best value/position with its current fitness value and position.

In the second step, the threads in a block collaborate to find the minimal local best value out of all particles in the swarm using a parallel reduction. If the minimal value is better than the global best, the threads replace the global best position.

Threads work together and update as close to an equal number of dimensions as possible. This allows us to have multiple threads updating the global best position, rather than relying on only a single thread to accomplish this task. Similar to the initialization phase, this kernel is very straightforward, and, as a result, we do not provide the pseudocode.

6.2.5 Swap Particles

Finally, the swap particles phase replaces the n worst particles in a swarm with the n best particles of its neighboring swarm. Following the work of Vanneschi et al. [19], we set the swarms up as a simple ring topology in order to determine the direction of swaps. We use two kernels for this phase. The first kernel determines the n best and worst particles in each swarm. For this kernel, we again launch one thread per particle, with thread blocks composed only of threads covering particles in the same swarm. In order to determine the n best and worst particles, we iterate n parallel reductions, one after the other. Each parallel reduction determines the nth best/worst particle in the swarm covered by that thread block. We improve the performance of this lengthy kernel by reading from global memory only once: at the beginning of a kernel each thread reads in the fitness value of its particle into a shared memory buffer. This buffer is then copied into two secondary buffers which are used in the parallel reduction (one for managing best values, one for worst).

Once a reduction has been completed, we record the index of the located particles into another shared memory buffer. We then restart the reduction for finding the $n + 1$th particle by invalidating the best/worst particle from the original shared memory buffer and recopying this slightly modified data into the two reduction shared memory buffers. This process continues until all best/worst particles have been found. At this point, n threads per block write out the best/worst particle indices to global memory in a coalesced fashion.

The second kernel handles the actual movement of particles between swarms. This step involves replacing the position, velocity, and local best values/position of any particle identified for swapping by the first kernel. For this kernel we launch one thread per dimension per particle to be swapped.

6.2.6 CPU Control Loop

In our implementation, the CPU only manages the main loop of PSO, the invocation of the various GPU kernels, and determines when new random numbers need to be generated on the GPU.

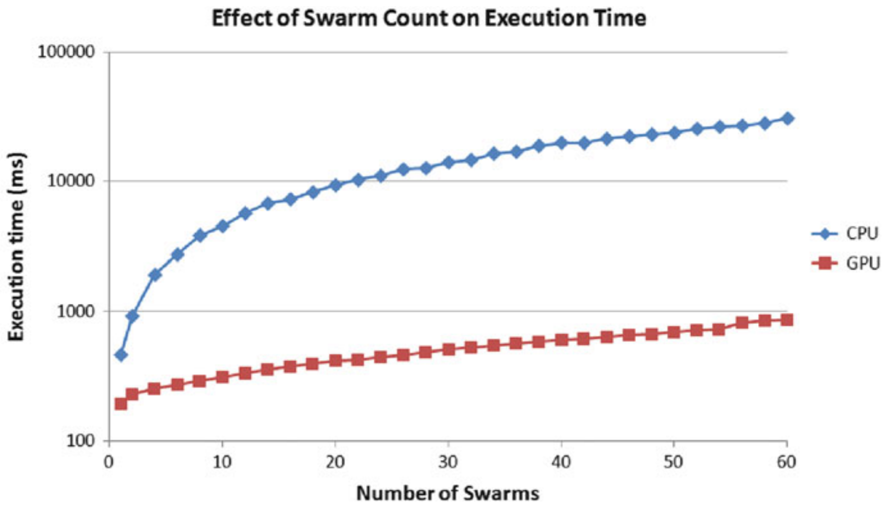


Fig. 8 Comparison between sequential CPU and GPU algorithm as swarm count increases

7 Results

To test the performance of our GPU implementation we compare it against a sequential multi-swarm PSO algorithm. This sequential algorithm has not been optimized for a specific CPU architecture, but it has been tuned for sequential execution. We execute the GPU algorithm on an Nvidia GTX 260 GPU with 27 SMs and the sequential CPU algorithm on an Intel Core 2 Duo running at 3.0 Ghz. Both algorithms have been compiled with the *-O3* optimization flag, and the GPU implementation also uses the *-use_fast_math* flag. Finally, we compare the solution quality against a single-swarm PSO implementation and a FCFS algorithm that sequentially assigns tasks to the machine with the lowest MAT value at the time (in this case, the MAT value includes the time to complete the task in question).

7.1 Algorithm Performance

In order to examine the performance of the algorithm, we first tested how the algorithm scales with swarm count. For these tests, we use 128 particles per swarm, 1,000 iterations, and swap 25 particles every 10 iterations. For the swarm count tests we set the numbers of tasks to 80 and machines to 8. As a result of this low machine count, the shared memory version of the fitness kernel is used throughout. Figure 8 shows the results for swarm counts from 1 to 60. As expected, the GPU implementation outperforms the sequential CPU implementation by a very high degree. With the swarm count set at 60, the GPU algorithm achieves an approximate 32 times speedup over the sequential algorithm.

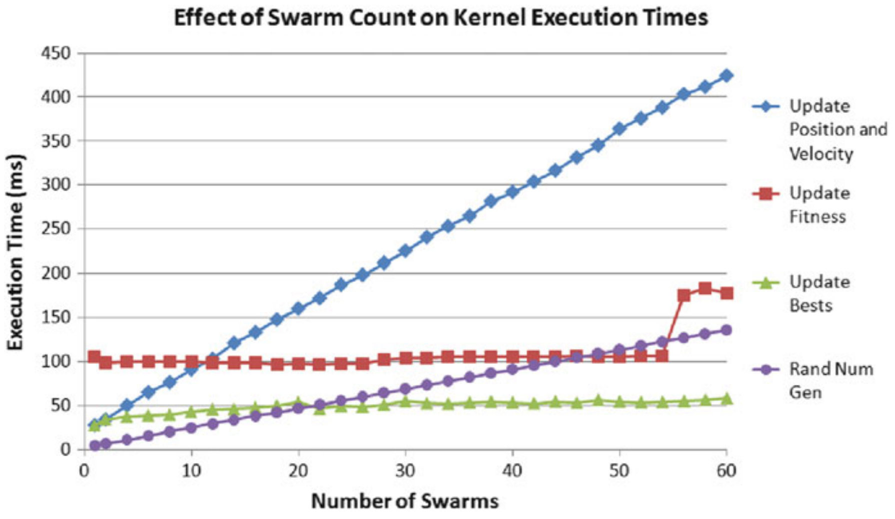


Fig. 9 Total execution time for the various GPU kernels as the swarm count increases

We further measured the total time taken for each of the GPU kernels as the swarm count increases. The results are shown in Fig. 9. The update position and velocity kernel contributes the most to the increase in the GPU's execution time as the swarm count increases. We explain this result by revisiting the overall responsibilities of this kernel. That is, the update position and velocity kernel requires a large number of global memory reads and writes (to read in the many-dimensioned position, velocity, and current bests position data), and updating the velocity is relatively computationally intensive. When combined with the fact that we are launching a thread per dimension per particle, the GPU's resources quickly become saturated.

We explain the "jump" in the fitness kernel's execution time at the last three data points as due to thread blocks waiting for execution. The GTX 260 GPU has 27 SMs available. With, for example, 56 swarms, we have 56 thread blocks assigned to the fitness kernel. With the configuration tested, each SM can support only two thread blocks simultaneously. Hence, the GPU executes *54* thread blocks simultaneously, leaving *2* thread blocks waiting for execution. This serialization causes the performance loss observed.

To prove that the performance loss is due to insufficient SMs, we take our experiments a step further and observe the performance of the algorithm on a GTX 570. While the GTX 570 technically contains less SMs than the GTX 260, each SM contains more CCs and allows for more threads and/or thread blocks to execute on each simultaneously. As a result, we expect the jump in execution time at swarm counts above 56 to be absent in the GTX 570 tests. As we show in Fig. 10, our expectation matched our experimentation. We see that the GTX 570

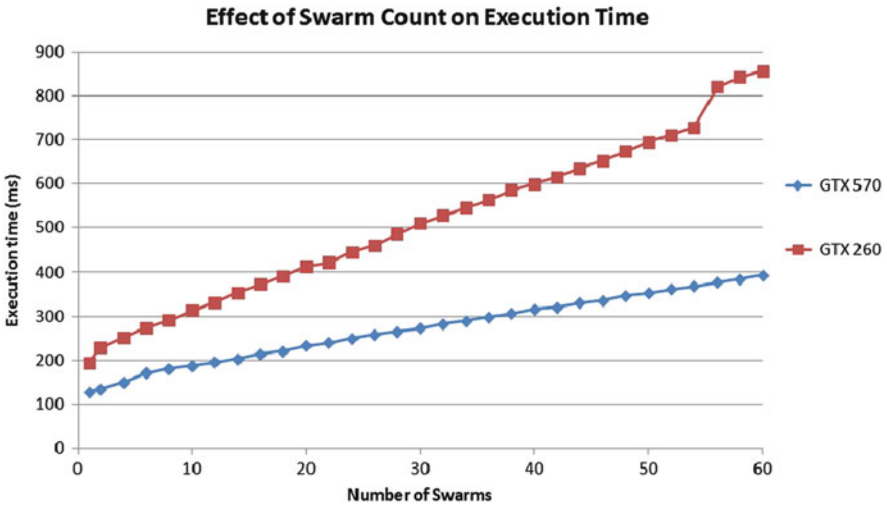


Fig. 10 Comparison between GTX 260 and 570 GPUs as swarm count increases

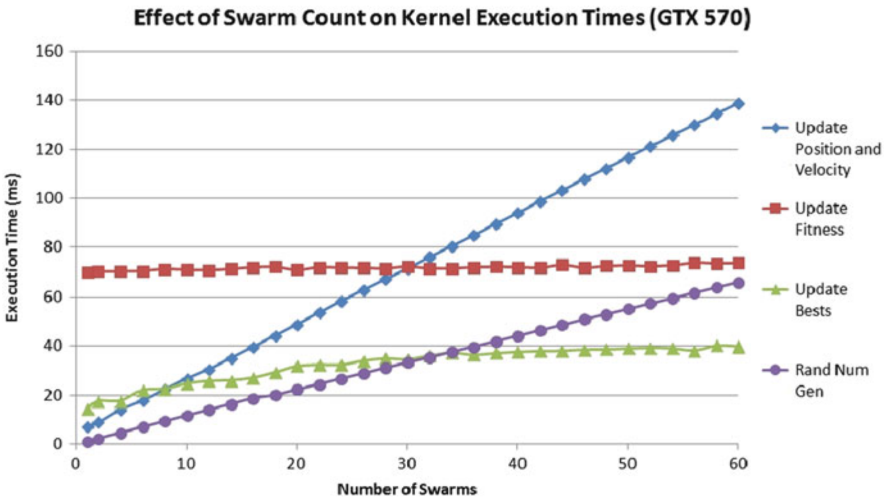


Fig. 11 Total execution time for the various GPU kernels as the swarm count increases for a GTX 570 GPU

performance line does not contain the same jump in the last three data points (at swarm count sizes of 56, 58, and 60). The superior overall performance of the GTX 570 is expected and is simply due to the nature of testing on newer, more powerful hardware with more available hardware parallelism compared to the GTX 260.

We measure the kernel execution times for the GTX 570 as well and have provided the results in Fig. 11. We observe three differences between these new

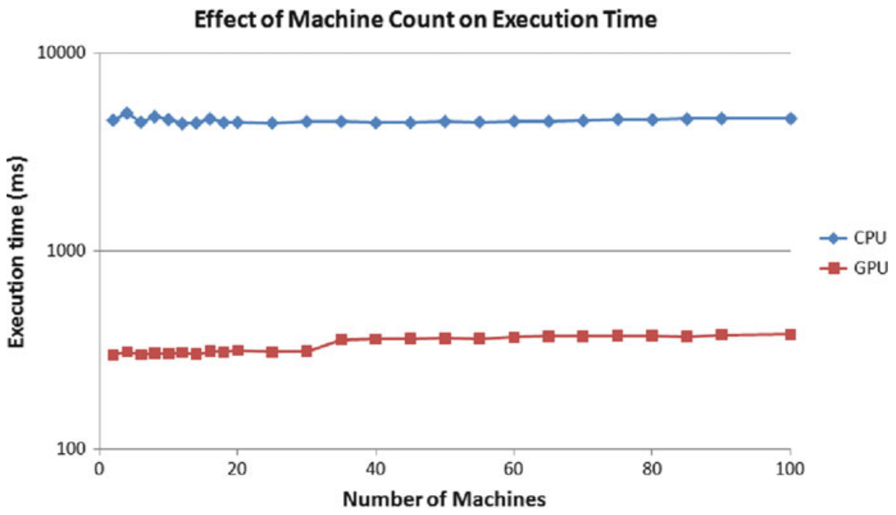


Fig. 12 Comparison between sequential CPU and GPU algorithm as machine count increases

results and those from the GTX 260 in Fig. 9: the first being the overall reduced execution time of each measured kernel. We expect this, of course, for reasons already discussed: the GTX 570 contains more CCs and a higher level of computational performance than the GTX 260. Secondly, we see that the execution time for the fitness update phase does not increase dramatically at swarm counts greater than or equal to 56. This falls in line with the results we observed and discussed in Fig. 10.

Finally, we observe that the update bests kernel appears to have a much more variable execution time than what we saw for the GTX 260 results in Fig. 9. The results between the two are, in fact, very similar. Both the GTX 260 and GTX 570 see these perturbations and a slow increase in the execution time of the update bests kernel as the swarm count increases, but this pattern is not as easily visible in Fig. 9 due to the difference in the scale of the Y-axis. While these GTX 570 results do not hold any surprises, we want to note that our original implementation was run on the GTX 570 without changes (that is: without Fermi-specific optimizations or code changes). Were we to rebuild the algorithm from the ground up for the improved capabilities of the Fermi-based architectures, we may be able to squeeze further performance gains.

Moving back to our GTX 260 results, we also wanted to test our use of texture memory for storing the ETC matrix. In order to accomplish this, we profiled a few runs of the algorithm using the CUDA Visual Profiler tool. The results from this tool showed that we were correct in our hypothesis that texture memory would help the fitness kernel's performance as the profiler reported anywhere from 88 % to 97 % of ETC matrix requests were cache hits, significantly reducing the overall number of global memory reads required to compute the makespan.

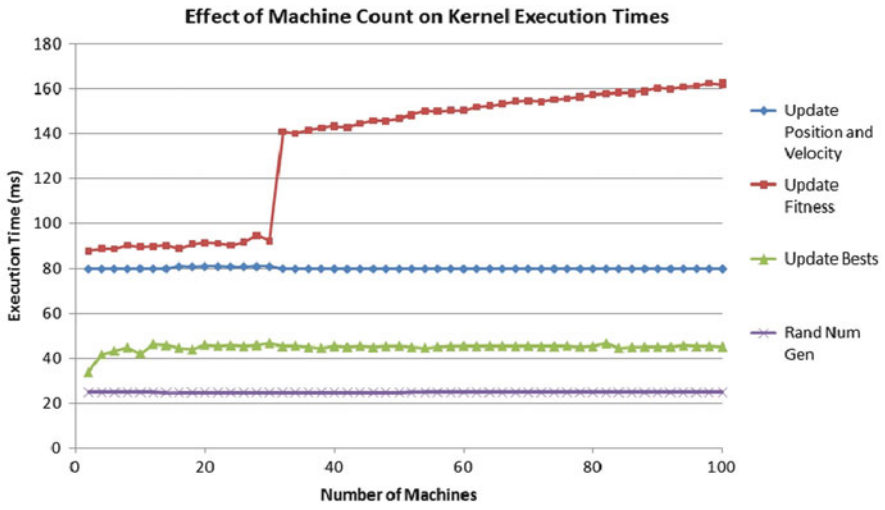


Fig. 13 Total execution time for the various GPU kernels as the machine count increases

Moving on, we examine the performance and scaling of the algorithm when we increase machine counts as well as increase the task counts. For the machine count scaling we keep the task count and the number of swarms static at 80 and 10, respectively. As the machine count increases, we observe the effect that switching to the global memory fitness kernel has on the execution time. Figure 12 shows the results with machine counts from 2 to 100. Unlike the swarm count tests, we see that the execution time does not change dramatically as the machine count increases. However, the GPU execution time still increases by 14 % when the machine count increases from 30 to 32. This occurs due to the shift from shared memory to global memory use for the fitness kernel, which, in turn, results in a 50 % increase in the total execution time for this kernel.

Figure 13 provides the execution time results of the top kernels as the machine count increases. We immediately observe that all but the fitness update kernel exhibit roughly static execution times. We expect these results, as increasing the machine count does not result in any computational or memory access increases for these kernels. We do, however, see a substantial increase in the execution time of the fitness kernel after 30 machines. As we know, this is the point where the algorithm shifts from using the shared memory fitness kernel to the global memory kernel. These results help us to observe the significant improvements in performance achieved by using shared memory over global memory.

Moving on to the task count scaling tests, we keep the machine count and number of swarms static at 8 and 10, respectively. We provide the results for task count scaling in Fig. 14. The results are very similar to those of the swarm count tests in that the GPU algorithm significantly outperforms the sequential CPU algorithm, and the execution time increases as the task count increases. Overall, however, the

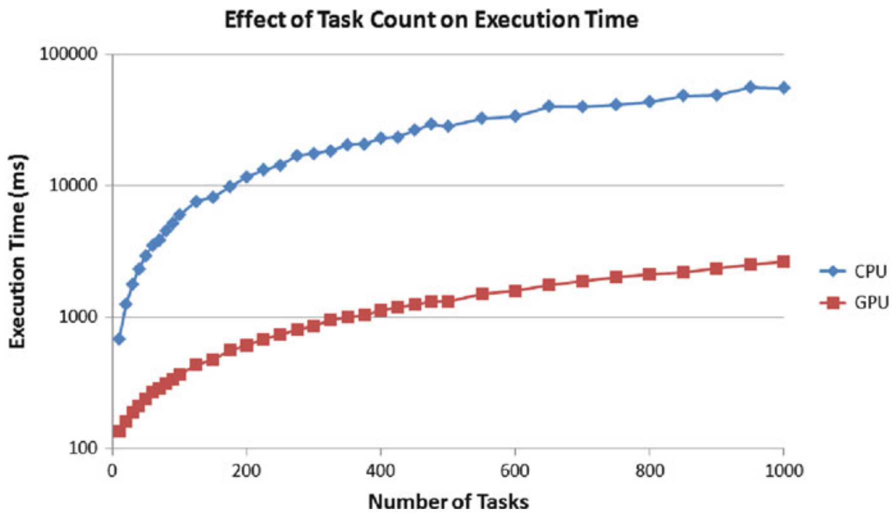


Fig. 14 Comparison between sequential CPU and GPU algorithm as task count increases

GPU cannot provide the same level of speedup while the swarm count remains low, despite increasing task counts. We expect this, as increasing the number of swarms increases the exploitable parallelism at a faster rate than the task count. We do not provide a graph of the various kernel execution times as they are very similar to those in Fig. 9, in that the update position and velocity kernel dominates the run time again as the algorithm uses the shared memory fitness kernel.

Finally, we ran two tests using a large number of tasks and swarms, with one using the shared memory kernel (10 machines) and the other using the global memory kernel (100 machines) in order to gauge the overall performance of the algorithm as well as come up with the overall percentage of execution time each kernel uses. Figure 15 shows the results (the percentages for the initialization and swapping kernels are not included in the figure as, combined, they contribute less than 1 % to the overall execution time). Clearly, the shared memory instance is dominated by the update position and velocity kernel, whereas the global memory instance sees the fitness kernel moving to become the top contributor to the overall execution time. As expected, the shared memory instance sees an improved speedup (compared to the sequential CPU algorithm) of 37 compared to the global memory instance's speedup of 23.5.

7.2 *Solution Quality*

For the solution quality tests we compare the results of the GPU multi-swarm PSO (MSPSO) algorithm with PSO and FCFS (which attempts to assign tasks to

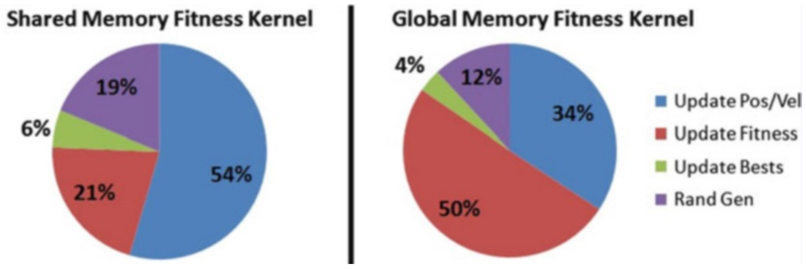


Fig. 15 Percentage of execution time taken by most significant kernels

Table 1 Solution quality of MSPSO and PSO normalized to FCFS solution (< 1 is desired)

Num tasks	Num machines	MSPSO	PSO
60	10	0.906	0.925
60	15	0.935	0.921
70	10	0.939	0.923
70	15	0.941	0.933
80	10	0.964	0.934
200	40	1.322	1.312
1,000	100	3.106	3.109

machines based on the current MAT values for each machine before and after the task is added). We use 10 swarms with 128 particles per swarm. c_1 is set to 2.0, c_2 to 1.4, and w to 1.0. We also introduce a *wDecay* parameter which reduces w each iteration to a minimum value (set to 0.4) and runs $1,000$ iterations of PSO for each problem. Finally, we randomly generate 10 task and machine configurations for each problem size considered and run PSO against each of these data sets. Each data set is run 100 times, and the averaged results are taken over each of the 100 runs.

Table 1 provides the averaged results of our experiments, normalized to the FCFS solution. We first tested small data sets of sizes similar to those from Sadasivam and Rajendran [14] as well as Yan-Ping et al. [21]. We can see from these that, unfortunately, MSPSO does not outperform the single-swarm PSO to any significant degree and performs worse on many occasions. Furthermore, as the problem size increases, both variants of PSO fail to generate improved solutions when compared to FCFS. In short, we do not see a reasonable level of quality improvement from MSPSO with small problem sizes, and both variants of PSO utterly fail to provide acceptable solution quality as the problem size increases.

Our explanation for this failure to provide a reasonable level of quality in the solution rests with the nature of the solution space. Our hypothesis is that the unstructured, random nature of the solution space presents an environment inimical to the intelligence of the particles. These particles attempt to use their memory and intelligence to track down optimal values in the solution space and are influenced by previously known optimal locations. That is, they follow some structure in the solution space and hope their exploration leads to an ideal solution. Unfortunately, with task matching, we have no real structure to the solution space. With even one task changing assignment from one machine to another, the makespan may dramatically change. As a result, the intelligence of the particles cannot help us here. The end result, we believe, is that the PSO algorithm devolves into a randomized algorithm (or worse, since the intelligence of the particles reduces the overall area of the solution space explored). The added cooperation between swarms in MSPSO further provides no benefits and perhaps even serves to cluster particles *between* swarms in all the same areas. In essence, the *exploration* aspects of PSO help us to no greater degree than a randomized algorithm, and, thus, the *exploitation* aspects are rendered useless, as exploitation of areas around local optima provides no help given the unstructured nature of the solution space. As we will discuss in the next and final section, this is an area with definite possibilities for future work and investigation.

8 Conclusions and Future Work

At the start of this chapter, we proposed that collaborative, multi-swarm PSO represented an ideal variant of PSO for parallel execution on the GPU. We described how the synchronous nature of PSO combined with the significant degree of parallelism offered by multi-swarm PSO provided a good complement to the capabilities of the GPU. By implementing the various phases as individual (or, in the case of swapping, multiple) kernels, we have achieved two goals: 1. We have captured the original synchronous nature of the phases within the PSO algorithm via the natural synchronization between GPU kernels. 2. We have allowed for the fine-tuning of parallelism for each phase of PSO. As our performance analysis and results showed, multi-swarm PSO performs exceptionally well on the GPU.

While the quality of solution for multi-swarm PSO left much to be desired for larger problem sizes, the majority of the contributions made in this chapter are easily transferrable to PSO algorithms focusing on solving other problems. In these cases, only the fitness kernel itself requires a significant level of modification—the knowledge gained through the design, implementation, and analysis of all the remaining kernels retain a high level of generality.

For future work we can immediately identify the potential for further analysis to see if this multi-swarm PSO algorithm can be tuned in order to improve the solution quality further. With large problem sizes we saw the solution quality suffer when compared against a deterministic algorithm, and the results were, overall, quite close to a single-swarm PSO algorithm. We believe that a future investigation into whether or not MSPSO can be tuned further to more readily support these types of problems is worthwhile. Furthermore, we believe it may be interesting to see if the onboard cache in Fermi-based GPUs can provide a performance boost for the global memory-based fitness kernel. While we provided some brief experimentation with

Fermi-based GPUs in this work, we did not specifically tune the GPU algorithm for the changes introduced with the Fermi architecture. We leave these performance tests and modifications as future work.

References

1. Change, C.W., Lee, Y.C., Lee, C.N., Chou, T.Y.: Ant colony optimisation for task matching and scheduling. IEEE Proc. Comput. Digit. Tech. **153**(6), 373–380 (1997)
2. de Veronese, P.L., Krohling, R.A.: Swarm's flight: accelerating the particles using C-CUDA. In: IEEE Congress on Evolutionary Computation, Trondheim, pp. 3264–3270 (2009)
3. Flynn, M.: Some computer organizations and their effectiveness. IEEE Trans. Comput. **C-21**(9), 948–960 (1972)
4. Freund, R.F., Gherrity, M., Ambrosius, S., Campbell, M., Halderman, M., Hensgen, D., Keith, E., Kidd, T., Kussow, M., Lima, J.D., Mirabile, F., Moore, L., Rust, B., Siegel, H.J.: Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet. In: The Seventh IEEE Heterogeneous Computing Workshop, Orlando, pp. 184–199 (1998)
5. Kang, Q., He, H., Wang, H., Jiang, C.: A novel discrete particle swarm optimization algorithm for job scheduling in grids. In: Fourth International Conference on Natural Computation, pp. 401–405. IEEE, Jinan (2008)
6. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: IEEE International Conference on Neural Networks, vol. 4, pp. 1942–1948. IEEE, Perth (1995)
7. Liang, J.J., Suganthan, P.N.: Dynamic multi-swarm particle swarm optimizer with local search. In: IEEE Congress on Evolutionary Computation, Edinburgh, pp. 522–528 (2005)
8. Maheswaran, M., Ali, S., Siegel, H.J., Hensgen, D., Freund, R.F.: Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In: The Eighth IEEE Heterogeneous Computing Workshop, San Juan, pp. 30–44 (1999)
9. Mussi, L., Cagnoni, S., Daolio, F.: GPU-based road sign detection using particle swarm optimization. In: Ninth International Conference on Intelligent Systems Design and Applications, pp. 152–157. IEEE, Pisa (2009)
10. Mussi, L., Daolio, F., Cagnoni, S.: Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture. Inform. Sci. **181**(20), 4642–4657 (2011)
11. NVIDIA: CUDA Programming Guide Version 3.1. NVIDIA, Santa Clara (2010)
12. NVIDIA: CUDA C Best Practices Guide. NVIDIA, Santa Clara (2011)
13. Nvidia: Nvidia CUDA developer zone. http://developer.nvidia.com/category/zone/cuda-zone (2011)
14. Sadasivam, G.S., Rajendran, V.: An efficient approach to task scheduling in computational grids. Int. J. Comput. Sci. Appl. **6**(1), 53–69 (2009)
15. Shi, Y., Eberhart, R.: A modified particle swarm optimizer. In: IEEE World Congress on Computational Intelligence, pp. 69–73. IEEE, Anchorage (1998)
16. Solomon, S., Thulasiraman, P., Thulasiram, R.K.: Collaborative multi-swarm PSO for task matching using graphics processing units. In: 13th Annual Conference on Genetic and Evolutionary Computation (GECCO), Dublin, pp. 1563–1570 (2011)
17. Tasgetiren, M.F., Liang, Y.C., Sevkli, M., Gencyilmaz, G.: Particle swarm optimization and differential evolution for the single machine total weighted tardiness problem. Int. J. Prod. Res. **44**(22), 4737–4754 (2006)
18. van den Bergh, F., Engelbrecht, A.P.: A cooperative approach to particle swarm optimization. IEEE Trans. Evol. Comput. **8**(3), 225–239 (2004)
19. Vanneschi, L., Codecasa, D., Mauri, G.: An empirical comparison of parallel and distributed particle swarm optimization methods. In: The Genetic and Evolutionary Computation Conference, Portland, pp. 15–22 (2010)

20. Wang, L., Siegel, H.J., Roychowdhury, V.P., Maciejewski, A.A.: Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. J. Parallel Distr. Comput. **47**(1), 8–22 (1997)
21. Yan-Ping, B., Wei, Z., Jin-Shou, Y.: An improved PSO algorithm and its application to grid scheduling problem. In: International Symposium on Computer Science and Computational Technology, pp. 352–355. IEEE, Shanghai (2008)
22. Zhang, L., Chen, Y., Sun, R., Jing, S., Yang, B.: A task scheduling algorithm based on PSO for grid computing. Int. J. Comput. Intell. Res. **4**(1), 37–43 (2008)
23. Zhou, Y., Tan, Y.: GPU-based parallel particle swarm optimization. In: IEEE Congress on Evolutionary Computation, Trondheim, pp. 1493–1500 (2009)

ACO with Tabu Search on GPUs for Fast Solution of the QAP

Shigeyoshi Tsutsui and Noriyuki Fujimoto

Abstract In this chapter, we propose an ACO for solving quadratic assignment problems (QAPs) on a GPU by combining tabu search (TS) in the Compute Unified Device Architecture (CUDA). In TS on QAPs, there are $n(n-1)/2$ neighbors in a candidate solution. These TS moves form two groups based on computing cost. In one group, the computing of the move cost is $\mathscr{O}(1)$, and in the other group the computing of the move cost is $\mathscr{O}(n)$. We compute these groups of moves in parallel by assigning the computations to threads of CUDA. In this assignment, we propose an efficient method which we call *Move-Cost Adjusted Thread Assignment (MATA)* that can reduce disabling time, as far as possible, in each thread of CUDA. As for the ACO algorithm, we use the Cunning Ant System (cAS). GPU computation with MATA shows a promising speedup compared to computation with CPU. Based on MATA, we also implement two types of parallel algorithms on multiple GPUs to solve QAPs faster. These are the island model and the master/slave model. As for the island model, we used four types of topologies. Although the results of speedup depend greatly on the instances which we use, we show that the island model IM_ELMR has a good speedup feature. As for the master/slave model, we observe reasonable speedups for large sizes of instances, where we use large numbers of agents. When we compare the island model and the master/slave model, the island model shows promising speedup values on class (iv) instances of QAP. On the other hand, the master/slave model consistently shows promising speedup values both on classes (i) and (iv) with large-size QAP instances with large numbers of agents.

S. Tsutsui (✉)
Hannan University, 5-4-33, Amamihigashi, Matsubara, Osaka 580-8502, Japan
e-mail: tsutsui@hannan-u.ac.jp

N. Fujimoto
Osaka Prefecture University, 1-1, Gakuen-Cho, Naka-ku, Sakai, Osaka 599-8531, Japan
e-mail: fujimoto@mi.s.osakafu-u.ac.jp

1 Introduction

Recently, parallel computations using graphics processing units (GPUs) have become popular with great success, especially in scientific fields such as fluid dynamics, image processing, and visualization using particle methods [20]. These parallel computations are reported to see a speedup of tens to hundreds of times compared to CPU computations. Studies on parallel ACO with GPU are found in Bai et al. [4], Fu et al. [12], and Delévacqa et al. [7]. They implemented the MAX-MIN Ant System (MMAS) on a GPU with CUDA and applied it to solve the traveling salesman problem (TSP). In [8], Diego et al. proposed a parallelization strategy to solve the vehicle routing problem (VRP) with ACO on a GPU.

Studies solving the quadratic assignment problem (QAP) on GPUs using an evolutionary model are found in [15, 21, 27, 28]. In [27, 28], distributed GA models were used and no local searches were applied. In [21], a cellular GA model was used and no local searches were used. In [15], parallel hybrid evolutionary algorithms on CPU and GPU were proposed and applied to QAPs. In our previous studies [27, 28], we applied GPU computation to solve quadratic assignment problems (QAPs) using a distributed parallel GA model on GPUs. However, in those studies no local searches were applied.

In this chapter, we propose a parallel ACO for QAPs on a GPU by combining tabu search (TS) with ACO in the Compute Unified Device Architecture (CUDA) [17]). In a QAP, a solution ϕ is presented by a permutation of $\{0, 1, \cdots, n-1\}$ where n is the problem size. Here we consider neighbors $N(\phi)$ which are obtained by swapping two elements (i, j) of ϕ. In $N(\phi)$, there are $n(n-1)/2$ neighbors. In a TS which uses $N(\phi)$ as neighbors, we need to compute move costs for all neighbors in $N(\phi)$. Depending on the pair value (i, j), these moves can be divided into two groups based on computing cost. In one group, the computing of the move cost is $\mathcal{O}(1)$, and in the other group the computing of the move cost is $\mathcal{O}(n)$ [23]. We compute these groups' moves in parallel by assigning the computations to threads in a thread block of CUDA. In this assignment, we propose an efficient method which we call *Move-Cost Adjusted Thread Assignment (MATA)* that can reduce disabling time, as far as possible, in each thread of CUDA. As for the ACO algorithm, we use the Cunning Ant System (cAS) which is one of the most promising ACO algorithms [26].

In this chapter, we further implement the ACO algorithm based on MATA on a PC which has four GTX 480 GPUs for faster solution. Here, two types of ACO models using multiple GPUs are implemented. One is the island model, and the other is the master/slave model. In the island model, we implement one colony on each GPU, and agents (solutions) are exchanged among colonies at defined ACO iteration intervals using several types of topologies. In the master/slave model, we have only one colony in the CPU, and only local search (TS) processes are distributed to each GPU.

In the remainder of this chapter, Sect. 2 gives a brief review of GPU computation. Then, cAS and how we combine it with a local search are described in Sect. 3.

Section 4 describes a TS for combining with ACO to solve QAPs. Section 5 describes implementation of ACO with a TS on a GPU in detail. In Sect. 6, results and discussions are given. In Sect. 7, we describe how the algorithm is implemented on a PC which has four GTX 480 GPUs and discuss the results. Finally, Sect. 8 concludes the chapter.

2 A Brief Review of GPU Computation

2.1 GPU Computation with CUDA

In terms of hardware, CUDA GPUs are regarded as two-level shared-memory machines [17]. Processors in a CUDA GPU are grouped into streaming multiprocessors (SMs). Each SM consists of thread processors (TPs). TPs in an SM can exchange data via fast shared memory. On the other hand, data exchange among SMs is performed via global memory (VRAM). VRAM is also like main memory for processors. Code and data in a CUDA program are basically stored in VRAM. Although processors have no data cache for VRAM (except for the constant and texture memory areas), shared memory can be used as manually controlled data cache. Figure 1 shows a typical example of GPU (GTX 480).

The CUDA is a multi-threaded programming model. In Fig. 2, we describe an overview of the CUDA programming model. In a CUDA program, threads form two hierarchies: the *grid* and *thread blocks*. A block is a set of threads. A block has a dimensionality of 1, 2, or 3. A grid is a set of blocks with the same size and dimensionality. A grid has dimensionality of 1 or 2. Each thread executes the same code specified by the *kernel function*. A kernel-function call generates threads as a grid with given dimensionality and size. As for GPU in this study, we use a single GTX 480 [18].

2.2 Single Instruction, Multiple Threads

To obtain high performance with CUDA, here we need to know how each thread runs in parallel. The approach is called *Single Instruction, Multiple Threads (SIMT)* [19]. In SIMT each MP executes threads in groups of 32 parallel threads called *warps*. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path.

However, if threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. In our implementation to be described in Sect. 5, we designed the kernel function so that the threads that belong to the same warp will have as few branches as possible.

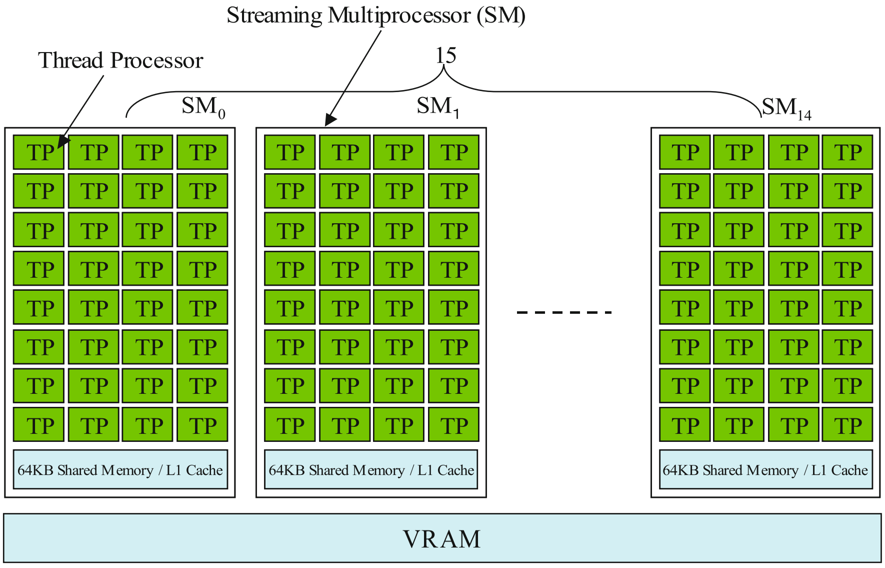
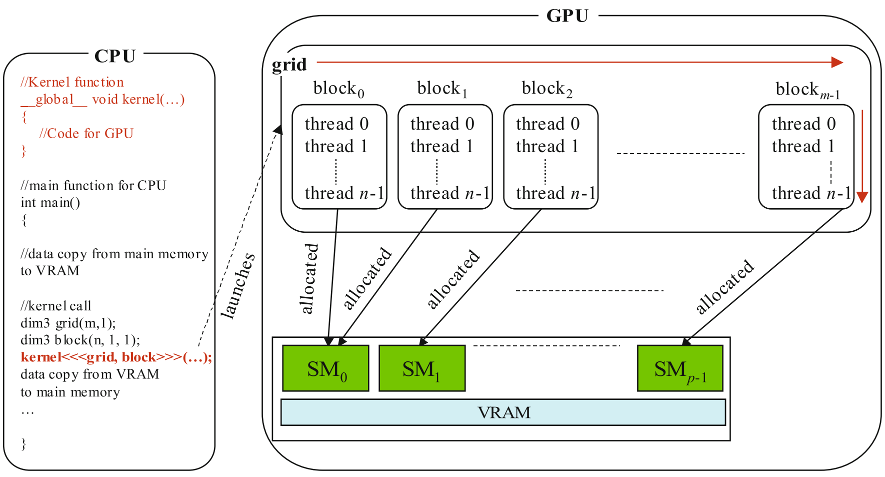


Fig. 1 An example of CUDA architecture (GTX 480). GTX 480 GPU has 15 SMs. Each SM consists of 32 TPs


```
//Kernel function
__global__ void kernel(…)
{
    //Code for GPU
}

//main function for CPU
int main()
{

//data copy from main memory
to VRAM

//kernel call
dim3 grid(m,1);
dim3 block(n, 1, 1);
kernel<<<grid, block>>>(…);
data copy from VRAM
to main memory
…

}
```

Fig. 2 An overview of CUDA programming model

3 Sequential ACO with a Local Search

As a bio-inspired computational paradigm, ACO has been applied with great success to a large number of hard problems such as the traveling salesman problem (TSP), QAP, scheduling problems, and vehicle routing problems [11]. The first example of an ACO was Ant System (AS) [9]. Since then, many variant ACO algorithms have been proposed as extensions of AS. Typical of these are Ant Colony System (ACS) and MAX-MIN Ant System (MMAS) [10]. In our previous study we proposed a new ACO algorithm called the Cunning Ant System (cAS). In this research we use cAS.

Although the ACO is a powerful metaheuristic, in many applications of ACO in solving difficult problems, it is very common to combine it with a local search or metaheuristics [10]. In this study, we combine cAS with a tabu search (TS) [14] which is also a powerful metaheuristic. cAS introduced two important schemes [26]. One is a scheme to use partial solutions, which we call *cunning*. In constructing a new solution, cAS uses preexisting partial solutions. With this scheme, we may prevent premature stagnation by reducing strong positive feedback to the trail density. The other is to use the colony model, dividing colonies into units, which has a stronger exploitation feature, while maintaining a certain degree of diversity among units (using partial solutions to seed solution construction in the ACO framework has been performed by combining an external memory implementation in [1, 2]).

cAS uses an agent called the cunning ant (*c-ant*). It differs from traditional ants in its manner of solution construction. It constructs a solution by borrowing a part of an existing solution. We call it a donor ant (*d-ant*). The remainder of the solution is constructed based on $\tau_{ij}(t)$ probabilistically as usual. Using *c-ant* in this way, we may prevent premature stagnation of the search, because only a part of the cities in a tour is newly generated, and this may prevent overexploitation caused by strong positive feedback to τ_{ij}(t). Let l_s represent the number of nodes of partial solution that are constructed based on $\tau_{ij}(t)$ (note that the number of nodes of partial solutions from its *d-ant* is $n - l_s$, where n is the problem size). Then cAS introduces the control parameter γ which can define $E(l_s)$ (the average of l_s) by $E(l_s) = n \times \gamma$. Using γ values in [0.2, 0.5] is a good choice in cAS [26].

The colony model of cAS is shown in Fig. 3. It consists of m units. Each unit consists of only one $ant^*_{k,t}$ ($k = 0, 1, \cdots, m-1$). At iteration t in unit k, a new $c\text{-}ant_{k,t+1}$ is generated using the existing ant in the unit (i.e., $ant^*_{k,t}$) as the $d\text{-}ant_{k,t}$. Then, the newly generated $c\text{-}ant_{k,t+1}$ and $d\text{-}ant_{k,t}$ are compared, and the better one becomes the next $ant^*_{k,t+1}$ of the unit. Thus, in this colony model, $ant^*_{k,t}$, the best individual of unit k, is always reserved. Pheromone density $\tau_{ij}(t)$ is then updated with $ant^*_{k,t}$ ($k = 0, 1, \cdots, m-1$) and $\tau_{ij}(t+1)$ is obtained as usual as

$$\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \sum_{k=0}^{m-1} \Delta^* \tau^k_{ij}(t), \tag{1}$$

$$\Delta^* \tau^k_{k,t} = \begin{cases} 1/C^*_{k,t} & \text{if } (i, j) \in ant^*_{k,t} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

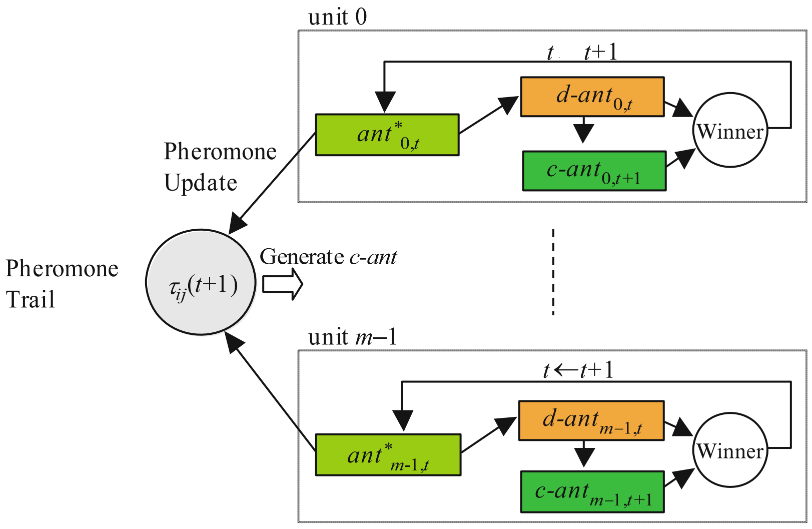


Fig. 3 Colony model of cAS with a local search. At iteration t in unit k, a new $c\text{-}ant_{k,t+1}$ is generated using the existing agent in the archive (i.e., $ant^*_{k,t}$ in unit k) as the $d\text{-}ant_{k,t}$. Then, the newly generated $c\text{-}ant_{k,t+1}$ and $d\text{-}ant_{k,t}$ are compared, and the better one becomes the next $ant^*_{k,t+1}$ of the unit

```
1.  $t \leftarrow 0$
2.  Set the initial pheromone density ($\tau_{ij}(t) = \tau_0$)
3.  Sample $c\text{-}ant_{k,t}$ randomly without using donor ($k$=0, 1, …, $m-1$)
4.  Apply local search (LS) to $c\text{-}ant_{k,t}$ and set it $ant^*_{k,t}$ ($k$=0, 1, …, $m-1$)
5.  Update $\tau_{ij}(t+1)$ using Eqs. (1) and (2)
6.  Construct $c\text{-}ant_{k,t+1}$ based on $d\text{-}ant_{k,t}$ ($ant^*_{k,t}$) and $\tau_{ij}(t+1)$ ($k$=0, 1, …, $m-1$)
7.  Apply local search (LS) to $c\text{-}ant_{k,t+1}$ ($k$=0, 1, …, $m-1$)
8.  Compare $c\text{-}ant_{k,t+1}$ and $d\text{-}ant_{k,t}$, and set the best one as $ant^*_{k,t+1}$
    ($k$=0, 1, …, $m-1$)
9.  $t \leftarrow t+1$
10. If the termination criteria are met, terminate the algorithm.
    Otherwise, go to 5.
```

Fig. 4 Algorithm description of sequential cAS with a local search

where the parameter ρ ($0 \leq \rho < 1$) models the trail evaporation, $\Delta^*\tau^k_{ij}(t)$ is the amount of pheromone by $ant^*_{k,t}$, and $C^*_{k,t}$ is the fitness of $ant^*_{k,t}$. Values of $\tau_{ij}(t+1)$ are set to be within $[\tau_{\min}, \tau_{\max}]$ as in MMAS [22]. Sequential cAS with a local search can be summarized as shown in Fig. 4 (please see [26] for detail).

4 Tabu Search for Combining with ACO to Solve QAPs

4.1 Quadratic Assignment Problem (QAP)

QAP is the problem which assigns a set of facilities to a set of locations and can be stated as a problem to find a permutation ϕ which minimizes

$$cost(\phi) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_{ij} b_{\phi(i)\phi(j)}, \tag{3}$$

where $A = (a_{ij})$ and $B = (b_{ij})$ are two $n \times n$ matrices and ϕ is a permutation of $\{0, 1, \cdots, n-1\}$. Matrix A is a flow matrix between facilities i and j, and B is the distance between locations i and j. Thus, the goal of the QAP is to place the facilities on locations in such a way that the sum of the products between flows and distances is minimized. The functional value $cost(\phi)$ in (3) relates to both distances between locations and flows between facilities. As a result, the problem structure of QAP is more complex and harder to solve than TSP [22].

4.2 Tabu Search for QAP

Tabu search (TS) [14] has been successfully applied to solving large combinatorial optimization problems [13]. Although TS is a powerful metaheuristic, it is often used in conjunction with other solution approaches such as evolutionary computation. In [22], the Robust TS (Ro-TS) by Taillard [23] was combined with MMAS and was used as a local search in solving QAPs.

The main idea of TS is as follows [23]. TS defines a neighborhood, or a set of moves that may be applied to a given solution to produce a new one. Among all the neighboring solutions, TS seeks one with the best evaluation. If there are no improving moves, TS chooses one that least degrades the objective function. In order to avoid returning to the local optimum just visited, a *taboo list* is used. An aspiration criterion is introduced to allow taboo moves to be chosen if they are judged to be interesting. There are several implementations of TS for QAP. In this study, we implement a TS to combine cAS on a GPU and use it as a local search with short runs. In this implementation, we used Ro-TS with some modification. In the following, we describe the TS for QAP in this study.

Following [24], the taboo list is a two-dimensional array $(n \times n)$ of integers where the element (i, j) identifies the value of the future iteration at which these two elements may again be exchanged with each other. For every facility and location, the latest iteration which the facility occupied that location is recorded. A move is taboo if it assigns both interchanged facilities to locations they had occupied within the taboo list size ($T_{\text{list-size}}$) in most recent iterations. The choice of $T_{\text{list-size}}$ is critical; if its value is too small, cycling may occur in the search process, while if its value is too large, appealing moves may be forbidden and lead to the exploration of lower quality solutions, resulting in a larger number of iterations [24]. To overcome the problem related to the search of the optimal taboo list size, Ro-TS introduced randomness in the value of $T_{\text{list-size}}$. Following [25], we set the taboo list size as $T_{\text{list-size}} \times r^3$, where r is a uniform random number in [0, 1).

We use the classical aspiration criterion that allows a taboo move to be selected if it leads to a solution better than the best found so far. In [24], an aspiration function that is useful for a longer term diversification process is proposed. However, since we use TS as a local search in ACO in this study, we do not use this kind of aspiration function here.

4.3 Move and Computation of Move Cost in QAP

As described in Sect. 4.2, TS seeks one with the best evaluation among all the neighboring solutions. If there are no improving moves, TS chooses one that least degrades the objective function. Thus, we need to calculate costs of all neighboring solutions efficiently. Let $N(\phi)$ be the set of neighbors of the current solution ϕ. Then a neighbor, $\phi' \in N(\phi)$, is obtained by exchanging a pair of elements (i, j) of ϕ. Then, we need to compute move costs $\Delta(\phi, i, j) = cost(\phi') - cost(\phi)$ for all the neighboring solutions. The neighborhood size of $N(\phi)$ ($|N(\phi)|$) is $n(n-1)/2$ where n is the problem size.

When we exchange r and s elements of ϕ (i.e., $\phi(r)$, $\phi(s)$), the change of $cost(\phi)$, $\Delta(\phi, r, s)$, can be computed in computing cost $\mathcal{O}(n)$ as follows:

$$\begin{aligned}\Delta(\phi, r, s) = {} & a_{rr}(b_{\phi(s)\phi(s)} - b_{\phi(r)\phi(r)}) + a_{rs}(b_{\phi(s)\phi(r)} - b_{\phi(r)\phi(s)}) \\ & + a_{sr}(b_{\phi(r)\phi(s)} - b_{\phi(s)\phi(r)}) + a_{ss}(b_{\phi(r)\phi(r)} - b_{\phi(s)\phi(s)}) \\ & + \sum_{k=0, k\neq r,s}^{n-1} \begin{pmatrix} a_{kr}(b_{\phi(k)\phi(s)} - b_{\phi(k)\phi(r)}) + a_{ks}(b_{\phi(k)\phi(r)} - b_{\phi(k)\phi(s)}) \\ + a_{rk}(b_{\phi(s)\phi(k)} - b_{\phi(r)\phi(k)}) + a_{sk}(b_{\phi(r)\phi(k)} - b_{\phi(s)\phi(k)}) \end{pmatrix}.\end{aligned} \tag{4}$$

Let ϕ' be obtained from ϕ by exchanging r and s elements of ϕ, then fast computation of $\Delta(\phi', u, v)$ is obtained in computing cost $\mathcal{O}(1)$ if u and v satisfy the condition $\{u, v\} \cap \{r, s\} = \emptyset$, as follows [24]:

$$\begin{aligned}\Delta(\phi', u, v) = {} & \Delta(\phi, u, v) + \\ & (a_{ru} - a_{rv} + a_{sv} - a_{su}) \\ & \times (b_{\phi'(s)\phi'(u)} - b_{\phi'(s)\phi'(v)} + b_{\phi'(r)\phi'(v)} - b_{\phi'(r)\phi'(u)}) \\ & + (a_{ur} - a_{vr} + a_{vs} - a_{us}) \\ & \times (b_{\phi'(u)\phi'(s)} - b_{\phi'(v)\phi'(s)} + b_{\phi'(v)\phi'(r)} - b_{\phi'(u)\phi'(r)}).\end{aligned} \tag{5}$$

To use this fast update, additional memorization of the $\Delta(\phi, i, j)$ values for all pairs (i, j) in a table is required.

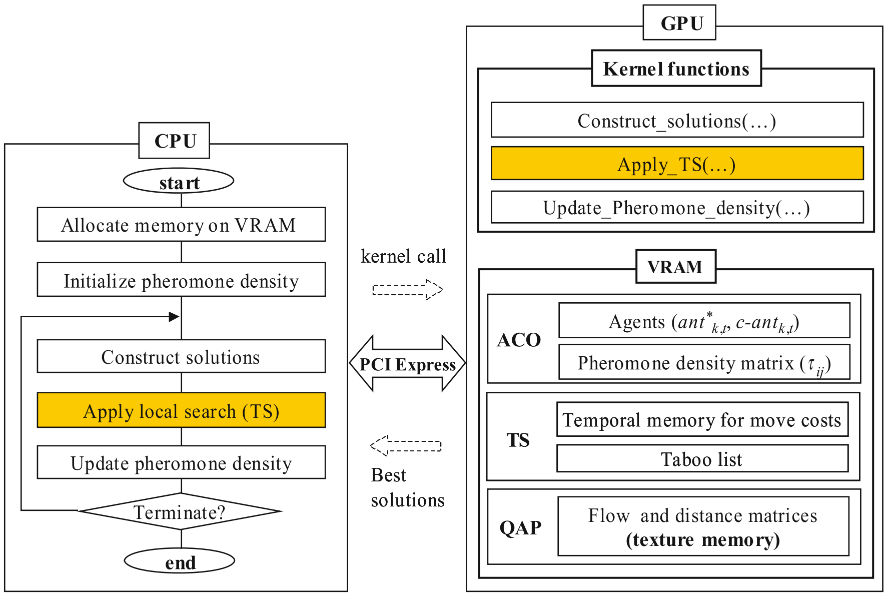


Fig. 5 Configuration of ACO with TS for solving QAPs on a GPU with CUDA

5 Implementation Details of ACO with TS on a GPU with CUDA

5.1 Overall Configuration

We coded the process of each step in Fig. 4 as a *kernel function* of CUDA. The overall configuration of ACO with TS for solving QAPs on a GPU with CUDA is shown in Fig. 5.

All of the data of the algorithm are located in VRAM of GPU. They include ACO data (the population pools ($ant^*_{k,t}$, $c\text{-}ant_{k,t}$), the pheromone density matrix τ_{ij}), TS data (the temporal memory for move costs, the tabu list), and QAP data (the flow matrix A and distance matrix B). Since matrices A and B are constant data, reading them is performed through texture fetching. On shared memory, we located only working data which are shared among threads tightly in a block.

As for the local search in Fig. 4, we implement TS described in Sect. 4.2. Construction of a new candidate solution (c-ant) is performed by the kernel function "Construct_solutions($\cdots$)" in a single block. Then each m solution is stored in VRAM. In the kernel function "Apply_tabu_search($\cdots$)", m solutions are distributed in m thread blocks. The Apply_tabu_search($\cdots$) function, in each block, performs the computation of move cost in parallel using a large number of threads. Kernel

Table 1 Summary of kernel functions. n is the problem size, m is the number of agents, and T_{TOTAL} is the total number of threads in a block (see Sect. 5.2.2)

Kernel functions		Dim3		Function
		Grid	Block	
Update pheromone density	Initialize_pheromone density()	n	n	$\tau_{ij}=t_0$
	Evaporate_pheromone()	n	n	τ_{ij} *= ρ
	Lay_pheromone()	m	1	τ_{ij} += $1/cost$
	Max_min_pheromone()	n	n	Adjust τ_{ij} in $[\tau_{\min}, \tau_{\max}]$
Construct solutions	Constuct_solutions(...)	m	1	Construct *c-ant*
Local search	Apply_tabu_search(...)	m	T_{TOTAL}	Improve solutions by TS

Table 2 Distribution of computation time of *c*AS in solving QAP with sequential runs on a CPU

Instances	Construction of solutions%	TS%	Updating trail%
tai40a	0.007	99.992	0.001
tai50a	0.005	99.994	0.000
tai60a	0.004	99.996	0.000
tai80a	0.002	99.997	0.000
tai100a	0.002	99.998	0.000
tai50b	0.022	99.976	0.002
tai60b	0.017	99.982	0.001
tai80b	0.011	99.988	0.001
tai100b	0.008	99.991	0.000
tai150b	0.005	99.995	0.000

function "Pheromone_update($\cdots$)" consists of four separate kernel functions for implementation easiness. Table 1 summarizes these kernel functions.

Kernel functions are called from the CPU for each ACO iteration. In these iterations of the algorithm, only the *best-so-far* solution is transferred to CPU from GPU. It is used for checking whether termination conditions are satisfied. Thus, in this implementation, overhead time used for data transfer between CPU and GPU can be ignored. In the end of a run, whole solutions are transferred from GPU to CPU.

Table 2 shows the distribution of computation time of *c*AS in solving QAP with sequential runs on a CPU. Here, we used QAP instances tai40a, tai50a, tai60a, tai80a, tai100a, tai50b, tai60b, tai80b, tai100b, and tai150b which will be used in experiments in Sect. 6. The conditions of the runs are the same as will be described in Sect. 6. From this table, we can see that TS uses over 99.9 % of the computation time. Thus, we can see that the efficient implementation of TS is the most important factor in increasing speedup of this algorithm.

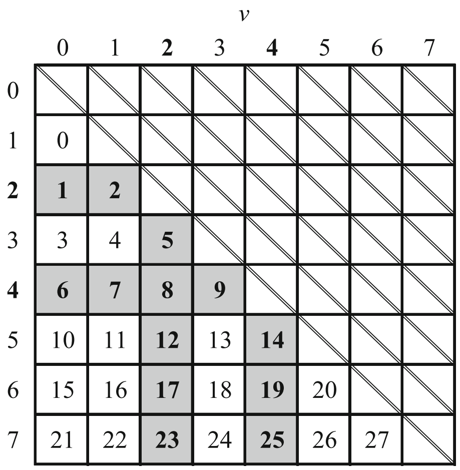


Fig. 6 An example of indexing of moves. In this example, we assume a problem size of $n = 8$. Thus, the neighborhood size $|N(\phi)|$ is $8 \times 7/2 = 28$

5.2 *Efficient Implementation of TS with CUDA*

5.2.1 An Inefficient Assignment of Move Cost Computations to Threads in a Block

In TS in this study, we use a table which contains move costs so that we can compute the move cost in $\mathcal{O}(1)$ using (5). For each move, we assign an index number as shown in Fig. 6. In this example, we assume a problem size of $n = 8$. Thus, the neighborhood size $|N(\phi)|$ is $8 \times 7/2 = 28$. As described in Sect. 5.1, each set of move cost calculations of an agent is being done in one block. The simplest approach to computing the move costs in parallel in a block is to assign each move indexed i to the corresponding sequential thread indexed i in a block.

Here, consider a case in which a solution ϕ' is obtained by exchanging positions 2 and 4 of a current solution ϕ in a previous TS iteration. Then the computation of $\Delta(\phi', u, v)$, shown numbers in gray squares, must be performed in $\mathcal{O}(n)$ using (4). The computation of the remaining moves are performed in $\mathcal{O}(1)$ quickly using (5).

Thus, if we simply assign each move to the block thread, threads of a warp diverge via the conditional branch ($\{u, v\} \cap \{2, 4\} = \emptyset$) into two calculations, threads in one group run in $\mathcal{O}(n)$ of (4), and threads in the other group run in $\mathcal{O}(1)$ of (5). In threads of CUDA, all instructions are executed in SIMT (please see Sect. 2.2 for detail). As a result, the computation time of each thread in a warp becomes longer, and we cannot receive the benefit of the fast calculation of (5) in GPU computation. Figure 7 shows this situation for the case shown in Fig. 6. Thus, if threads which run in $\mathcal{O}(1)$ and threads which run in $\mathcal{O}(n)$ coexist in the same warp, then their parallel computation time in the warp becomes longer than their own respective computation time.

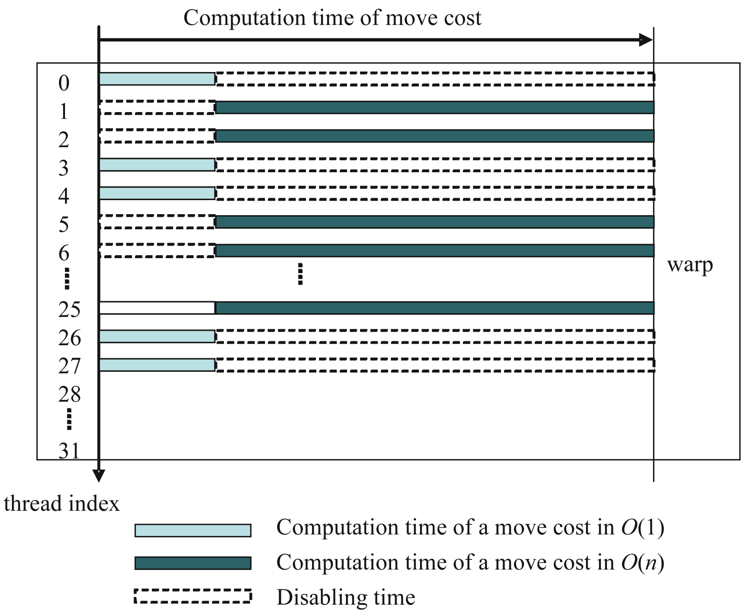

Computation time of move cost
0
1
2
3
4
5
6
25
26
27
28
31
warp
thread index
Computation time of a move cost in O(1)
Computation time of a move cost in O(n)
Disabling time

Fig. 7 Simple assignment of calculations of move costs to threads in a block. Due to disabling time in SIMT, the parallel computation by thread in a block is inefficient

Table 3 Neighborhood sizes for various problem sizes

Problem size n	Neighborhood size $\|N(n)\|$	$\|N(n)\|$ in $O(1)$	$\|N(n)\|$ in $O(n)$	$\frac{\|N(n)\| \text{ in } O(n)}{\|N(n)\|}$
	$n(n-1)/2$	$(n-2)(n-3)/2$	$2n-3$	
40	780	703	77	0.099
80	3,160	3,003	157	0.050
120	7,140	6,903	237	0.033
160	12,720	12,403	317	0.025
200	19,900	19,503	397	0.020

5.2.2 Move-Cost Adjusted Thread Assignment (MATA)

In general, for problem size n, the number of moves having move cost in $\mathcal{O}(1)$ is $(n-2)(n-3)/2$ and the number of moves having move cost in $\mathcal{O}(n)$ is $2n-3$. Table 3 shows these values for various problem sizes n. For larger-size problems, ratios of $|N(\phi)|$ in $\mathcal{O}(n)$ to $|N(\phi)|$ have smaller values than those in smaller-sized problems.

In this study, we assign move cost computations of a solution ϕ which are in $\mathcal{O}(1)$ and in $\mathcal{O}(n)$ to threads which belong to different warps in a block as described below.

Since the computation of a move cost which is $\mathcal{O}(1)$ is smaller than the computation which is $\mathcal{O}(n)$, we assign multiple number N_S of computations which are $\mathcal{O}(1)$ to a single thread in the block. Also, it is necessary to assign multiple calculations of the move costs to a thread, because the maximum number of threads in a block is limited (1024 for GTX 480 [18]).

Let C be $|N(\phi)|$ ($C = n(n-1)/2$). Here, each neighbor is numbered by $\{0, 1, 2, \cdots, C-1\}$ (see Fig. 6). Then, thread indexed as $t = \lfloor k/N_S \rfloor$ computes moves for $k \in \{tN_S, tN_S + 1, \cdots, tN_S + N_S - 1\}$. In this computation, if k is a move in $\mathcal{O}(n)$, then the thread indexed as t skips the computation. The total number of threads assigned for computations in $\mathcal{O}(1)$ is $T_S = \lceil C/N_S \rceil$. For each thread indexed as t, we need to know the move pair values (i, j) corresponding to each move assigned to it. In a thread indexed as t, if the pair (i, j) for its initial move tN_S is given, move pairs for $tN_S + 1, \cdots, tN_S + N_S - 1$ can be easily calculated. So, we prepared a lookup table to provide the pair values only for initial move in each t (move indexed as tN_S).

For the computation in $\mathcal{O}(n)$, we assign only one computation of move cost to one thread in the block. Although the total number of moves in $\mathcal{O}(n)$ is $2n-3$, we used $2n$ threads for these parallel computations for implementation convenience. Since the threads for these computations must not share the same warp with threads used for computations in $\mathcal{O}(1)$, the starting thread index should be a multiple of warp size (32), which follows the index of the last thread used for computation in $\mathcal{O}(1)$. Thus, the index of the first thread that computes move in $\mathcal{O}(n)$ is $T_{\mathrm{L-START}} = \lceil T_S/32 \rceil \times 32$.

This assignment is performed according to move pairs as follows. Let r and s be a pair of previous move. Then we assign pairs $(r, x), x \in \{0, 1, 2, \cdots, n-1\}$ to threads indexed from $T_{\mathrm{L-START}}$ to $T_{\mathrm{L-START}} + n - 1$ and assign pairs $(y, s), y \in \{0, 1, \cdots, n-1\}$ to threads indexed from $T_{\mathrm{L-START}} + n$ to $T_{\mathrm{L-START}} + 2n - 1$. Among these $2n$ threads, three threads assigned pairs (r, r), (s, s), and (r, s) do nothing. Note that thread assigned pair (s, r) does the move cost computation.

Thus, the total number of threads $T_{\mathrm{TOTAL}} = T_{\mathrm{L-START}} + 2n$ and this kernel function is called from CPU by kernel call "Apply_TS<<< m, T_{TOTAL} >>>($\cdots$);". Figure 8 shows the thread structure in a block in computing move costs for TS in this study. Hereafter, we refer to this thread structure as *Move-Cost Adjusted Thread Assignment*, or *MATA* for short.

6 Experiments

6.1 Experimental Platform

In this study, we used a PC which has one Intel Core i7 965 (3.2 GHz) processor and a single NVIDIA GeForce GTX 480 GPU. The OS was Windows 7 Professional. Microsoft Visual Studio 2010 Professional Edition with optimization option /O2 and CUDA 4.0 SDK were used.

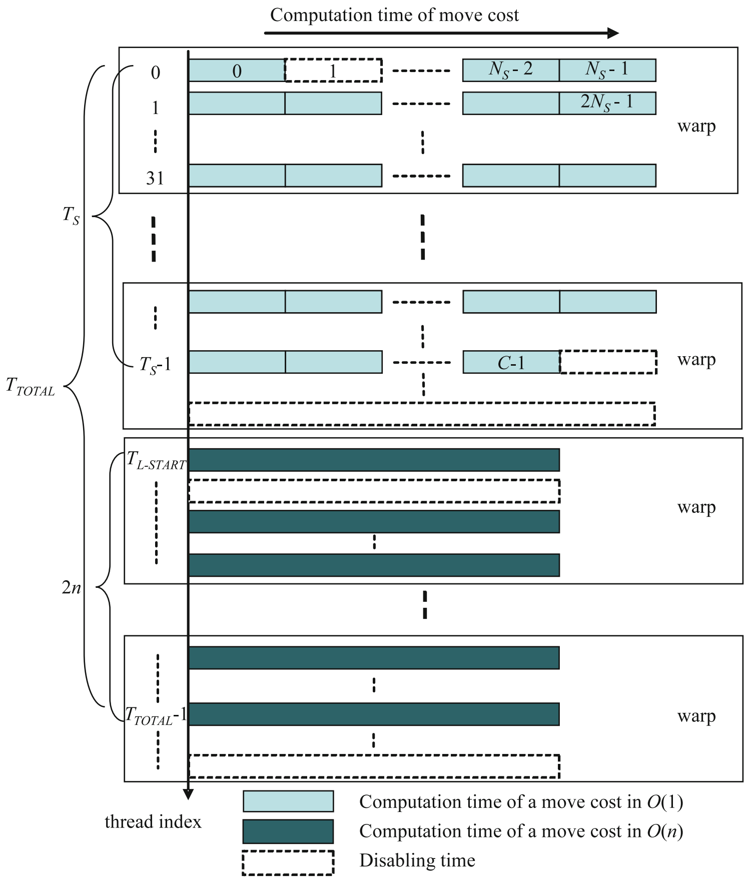


Fig. 8 The thread structure in a block in computing move costs for TS using Move-Cost Adjusted Thread Assignment (MATA)

The instances on which we tested our algorithm were taken from the QAPLIB benchmark library [5]. QAP instances in the QAPLIB can be classified into four classes: (i) randomly generated instances, (ii) grid-based distance matrix, (iii) real-life instances, and (iv) real-life-like instances [24]. In this experiment, we used the following ten instances which were classified as either (i) or (iv): tai40a, tai50a, tai60a, tai80a, tai100a, tai50b, tai60b, tai80b, tai100b, and tai150b (the numbers indicate the problem size n). Here, note that instances classified into class (i) are much harder to solve than those in class (iv).

Table 4 Revised parameter values. γ is a control parameter of cAS [26]

Parameters		Values	
		Class (i) QAPs	Class (iv) QAPs
ACO	Number of ants: m	n	n $(4n)$
	Evaporation factor: ρ	0.2 (0.5)	0.2 (0.5)
	γ	0.4	0.5
TS	Length of TS: IT_{TS}	$64n$ $(16n)$	n $(4n)$
	Taboo list size: $T_{\text{list\_size}}$	$4n$ (n)	$n/2$ (n)
	N_S	$n/4$	$n/4$

6.2 *Experimental Results*

In this section, we present revised results from a previous study [29]. We tuned TS parameters so that we can get better performance as shown in Table 4. Twenty-five runs were performed for each instance. In Table 4, values in parentheses are values used in [29].

Let IT_{TS} be the length of TS applied to one solution which is constructed by ACO and IT_{ACO} be the iterations of ACO, respectively. Then, $IT_{\text{TOTAL}} = m \times IT_{\text{ACO}} \times IT_{TS}$ represents a total length of TS in the algorithm. We define a value $IT_{\text{TOTAL-MAX}} = m \times n \times 3200$. In this revised experiment, if IT_{TOTAL} reaches $IT_{\text{TOTAL-MAX}}$ or a known optimal solution is found, the algorithm terminates. This $IT_{\text{TOTAL-MAX}}$ is larger than the $IT_{\text{TOTAL-MAX}}$ in [29].

In runs, we set the following three types of models: (1) runs on a GPU with the *MATA* in Sect. 5.2.2, (2) runs on a GPU without using MATA (non-MATA), and (3) sequential runs on a CPU using a single thread. In runs without using MATA, we assign to one thread N_S number of move calculations, similar to how it is done in MATA. Thus, in these runs we used a total number of $T_S = \lceil C/N_S \rceil$ threads in a block. In runs on a CPU, we use Intel Core i7 965 (3.2 GHz) processor using a single thread. The results are summarized in Table 5. First, see the effect of the approach including MATA. As seen in the table, the run time results with MATA in T_{avg} were faster than those of the non-MATA run time in T_{avg}, although these speedup values are different among instances. For example, on tai60a, T_{avg} with MATA is 34.8 and T_{avg} with non-MATA is 283.9, respectively. The speedup ratio by MATA is 8.2x. On tai60b, the speedup ratio by MATA is 7.7x. These speedup values range from 4.7x to 8.2x and the average value of the speedup values over ten instances is 6.4x. Thus, we can confirm that the MATA in Sect. 5.2 is a useful approach for fast execution in solving QAPs by ACO with TS on a GPU.

Now see the results of GPU computation of the proposed approach (MATA) compared to the results with CPU computations. Please note that in runs on a CPU, there is no parameter N_S in Table 4. Other parameter values are the same as runs on a GPU. On tai100a, e.g., GPU computation with MATA obtained T_{avg} of 431.5 and CPU computation obtained T_{avg} of 7907.0 showing a speedup of 18.3x. Note here

Table 5 Revised results. T_{avg} and Error(%) are mean run time and mean error over 25 runs, respectively

QAP instances		GPU computation (GTX 480)				CPU computation (i7 965 3.2 GHz)		Speedup in T_{avg}	
		T_{avg} (sec)			Error (%)	T_{avg} (sec)	Error (%)	$\frac{\text{CPU}}{\text{MATA}}$	$\frac{\text{CPU}}{\text{Non-MATA}}$
		MATA	Non-MATA	$\frac{\text{Non-MATA}}{\text{MATA}}$					
Class (i)	tai40a	9.2	70.8	7.7	0.15	191.4	0.13	20.8	2.7
	tai50a	17.8	126.6	7.1	0.36	464.0	0.35	26.1	3.7
	tai60a	34.8	283.9	8.2	0.36	963.8	0.36	27.7	3.4
	tai80a	153.9	728.6	4.7	0.38	3131.8	0.38	20.3	4.3
	tai100a	431.5	2357.3	5.5	0.35	7907.0	0.34	18.3	3.4
Class (iv)	tai50b	0.2	1.5	6.3	**0**	6.0	**0**	24.9	3.9
	tai60b	0.4	2.8	7.7	**0**	12.7	**0**	35.5	4.6
	tai80b	6.6	33.9	5.1	**0**	141.6	**0**	21.4	4.2
	tai100b	10.1	55.2	5.5	**0**	296.5	**0**	29.5	5.4
	tai150b	2888.4	16348.8	5.7	0.05	48953.8	0.06	16.9	3.0
Average		–	–	6.4	–	–	–	24.2	3.8

that if we compare T_{avg} of GPU with non-MATA and T_{avg} of CPU, the speedup ratio on this instance is only 3.4x. The speedup ratios of GPU with MATA to CPU are in the range from 16.9x to 35.5x, showing their average is 24.2x. Values of *Error* are smaller and values of *Speedup* in T_{avg} are larger than observed in [29] due to revised parameter settings and longer runs.

7 Implementation of ACO on Multiple GPUs for Faster Solution of the QAP

Since there are four PCIe x16 slots in the system described in Sect. 6.1, we added an additional three GTX 480 GPUs and constructed a multi-GPU environment with a total of four GTX 480 GPUs. In this section, we propose two types of multi-GPU models for ACO with MATA, to attain a fast computation speed in solving QAPs. They are the island model and the master/slave model, the most popular parallel EAs [3, 6].

7.1 Island Model

Island models for EAs in a massively parallel platform are intensively studied in [16]. The cAS, which is used as our ACO model in this study, has an archive which maintains m solutions ($ant^*_{k,t}$, $k = 0, 1, \cdots, m-1$ in Fig. 3). This archive is similar to a population in EAs. In our implementation, we exchange (immigrate) the solutions among GPUs. In our implementation, one ACO model on a GPU in Sect. 5 composes one island. In the configuration of ACO on a GPU in Fig. 5, all m

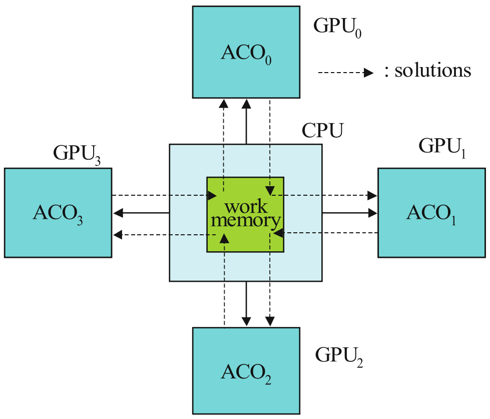


Fig. 9 Island model with four GPUs. In our implementation of island models, solutions in VRAM are transferred between GPU and CPU using usual "cudaMemcpy(···)" function when immigrations are required

solutions in the archive are maintained in VRAM of the GPU. In an island model, we need to exchange solutions among islands (GPUs) depending on its topology.

It is possible to exchange solutions among GPUs using "cudaMemcpyPeer(···)" function with CUDA 4.x. without via CPU. However, to perform exchange solutions depending on a defined topology, the CPU needs to know which data should be exchanged. This means that the CPU can't execute the cudaMemcpyPeer(···) function without having solutions from each GPU. Since the data needs to be sent to the CPU anyway, it is most efficient to exchange this data through the CPU rather than doing a direct exchange between GPUs. Thus, in our implementation of island models, solutions in VRAM are transferred between GPU and CPU using usual "cudaMemcpy(···)" function when immigrations are required as shown in Fig. 9. As for control multiple GPUs in the CPU, we use OpenMP API.

Although there are many topologies for island models [6], in this study we implement the following four models:

(1) *Island model with independent runs (IM_INDP):* Four GPUs are executed independently. When at least one ACO in a GPU finds an acceptable solution, then the algorithm terminates.
(2) *Island model with elitist (IM_ELIT):* In this model, at defined ACO iteration interval I_{interval}, the CPU collects the global best solution from the four GPUs and then distributes it to all GPUs except the GPU that produced that best solution. In each GPU, the worst solution in each archive is replaced with the received solution.
(3) *Island model with ring connected (IM_RING):* The best solution in each GPU g ($g = 0, 1, 2, 3$) is distributed to its neighbor GPU $(g + 1)$ *Mod* 4 at defined ACO iteration interval I_{interval}. In each GPU, the worst solution in each archive is replaced with the received solution if the received one is better than the worst one.
(4) *Island model with elitist and massive ring connected (IM_ELMR):* In this model, first the global best solution is distributed, as performed in IM_ELIT. Then,

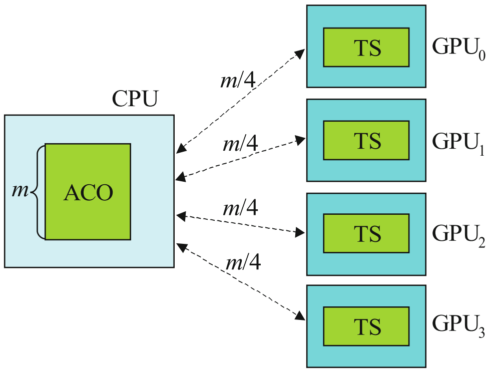


Fig. 10 Master/slave model with four GPUs. In the master/slave model, the ACO algorithm is executed in the CPU

in addition to this immigration operation, randomly selected $m \times d_{\text{rate}}$ $(0 < d_{\text{rate}} < 1)$ solutions in the archive in each GPU are distributed to its neighbor. Received solutions in each GPU are compared with randomly selected, non-duplicate $m \times d_{\text{rate}}$ solutions. We use d_{rate} value of 0.5 in this study.

7.2 Master/Slave Model

As described in Table 2 in Sect. 5, more than 99 % of computation time was used for execution of TS when we ran the algorithm using CPU with a single thread. In the master/slave model in this study, the ACO algorithm is executed in the CPU as shown in Fig. 10. Let m be the number of agents in the archive of cAS, and then we assign $m/4$ number of solutions to each GPU. When new solutions are generated in the CPU, first, $m/4$ number of solutions is transferred to each GPU, and then "Apply_tabu_search$(\cdots)$" kernel function is launched to apply the TS with MATA to these solutions. The improved solutions are sent back to the CPU from each GPU.

Note here that in practical implementation of the master/slave model, the value of m must be divisible by the number 4. So, we assigned $\lfloor m/4 \rfloor$ number of agents to each slave GPU and we used an agent number of $m' = \lfloor m/4 \rfloor \times 4$ instead of m.

7.3 Results of Using Multiple GPUs

7.3.1 Experimental Setup

We used four GTX 480 GPUs in four PCIe slots. We use the control parameter values shown in Table 4. We used the same QAP instances described in Sect. 6.1. Termination criteria are slightly different from those in Sect. 6.2. When we perform a fair comparison of different algorithms, sometimes it is difficult to determine their termination criteria. In this experiment, we run the algorithms until predetermined acceptable solutions are obtained and effectiveness of using four GPUs is measured by average time ($T_{\text{avg},4}$) to obtain the solutions. We obtain the speedup by $T_{\text{avg},1}/T_{\text{avg},4}$, where $T_{\text{avg},1}$ is average time to obtain acceptable solutions by ACO using a single GPU configured as described in Sect. 5. We performed 25 runs for each experiment.

We determined acceptable solutions as follows. For the class (i) instances, since it is difficult to obtain known optimal solutions 25 times in 25 runs with reasonable run time, we set their acceptable solutions to be within 0.5 % of the known optimal solutions. For the class (iv) instances, except tai150b, we set them to known optimal solutions. We set tai150b to be within 0.2 % of the known optimal solution. We used I_{interval} value of 1.

7.3.2 Results of the Island Models

Results of the island models are summarized in Table 6. The IM_INDP is the simplest of the island models. Thus, we use results of IM_INDP as bench marks for other island models. Except for the results from tai40a, all other island models had improved speedup values compared to IM_INDP. In the table, we showed the average number of iterations of the ACO to obtain the acceptable solutions (IT_{ACO}). On tai40a, this value is only 1.7x. Thus, on this instance, there was no benefit from immigration operations.

The speedup values of IM_RING and IM_ELIT showed very similar results with each other. On tai80b and tai150b, we can see superlinear speedup values. We performed t-test between IM_ELIT and IM_INDP, showing a clear effect of using this topology, especially for class (iv) instances. Since we used long-tabu search length for class (i) instances (see Table 4), values of IT_{ACO} are smaller than those of class (iv) instances. This could have caused the reduced effect of immigration on class (i) instances, compared with (iv) instances.

Among the four island models, IM_ELMR showed the best speedup, except for tai40a. However, the t-test between IM_ELIT and IM_ELMR shows that the advantage of using IM_ELMR over IM_RING and IM_ELIT on class (i) instances again becomes smaller than on class (iv) instances. The speedup values are different among instances. Consider why these differences occur using IM_INDP as a parallel model. Let probability density function of the run time on a single GPU be represented by $f(t)$ and probability distribution function of $f(t)$ be $F(t)$. Here, consider an IM_INDP with p GPUs. Let the probability distribution function of run time of the IM_INDP with p GPUs be represented by $G(t, p)$. Since there is no interaction among GPUs in IM_INDP, the $G(t, p)$ can be obtained as

$$G(p,t) = 1 - (1 - F(t))^p, \tag{6}$$

Table 6 Results of the island models with four GPUs. t-test between IM_ELIT and IM_INDP and t-test between IM_ELIT and IM_ELMR were also shown

QAP instances		Acceptable error (%)	$T_{\text{avg},1}$ (sec)	Average IT_{ACO}	Speedup ($T_{\text{avg},1}/T_{\text{avg},4}$)				p-values	
					IM_INDP	IM_RING	IM_ELIT	IM_ELMR	IM_INDP v.s. IM_ELIT	IM_ELIT v.s. IM_ELMR
Class (i)	tai40a	0.5	0.4	1.7	1.7	**1.8**	1.7	1.7	0.90	0.85
	tai50a	0.5	4.2	11.4	2.1	2.6	2.9	**3.3**	0.07	0.44
	tai60a	0.5	10.6	14.8	1.9	2.2	2.3	**2.5**	0.03	0.61
	tai80a	0.5	58.3	18.9	2.4	2.5	2.7	**2.9**	0.45	0.75
	tai100a	0.5	125.9	14.6	1.7	2.1	2.2	**2.5**	0.03	0.13
Class (iv)	tai50b	0	0.2	31.8	1.5	2.3	2.5	**3.0**	6.50E–05	0.08
	tai60b	0	0.4	25.7	1.2	1.4	1.9	**2.6**	4.68E–05	5.83E–05
	tai80b	0	6.6	121.3	1.5	4.7	4.3	**6.5**	9.16E–11	4.00E–09
	tai100b	0	10.1	67.0	1.4	2.3	2.3	**3.2**	6.00E–08	1.24E–05
	tai150b	0.2	105.9	116.6	1.8	4.2	4.3	**4.9**	2.10E–05	0.22

Table 7 Estimation of *Speedup* in IM_INDP

$f(t)$	Speedup(p)	Speedup(4)
$f(t) = 1, \quad 0 \le t \le 1$	$(p+1)/2$	2.5
$f(t) = 2(1-t), \quad 0 \le t \le 1$	$(2p+1)/3$	3
$f(t) = 3(1-t)^2 \quad 0 \le t \le 1$	$(3p+1)/4$	3.25
$f(t) = \lambda e^{-\lambda t} \quad 0 \le t$	p	4

and the average run time $T_{\text{avg},p}$ is obtained as

$$T_{\text{avg},p} = \int_0^\infty t \cdot G'(p,t)dt \tag{7}$$

Thus, the speedup with p GPUs is obtained as $Speedup(p) = T_{\text{avg},1}/T_{\text{avg},p}$. Table 7 shows the values of $Speedup(p)$ and $Speedup(4)$ for assuming various functions of $f(t)$. This analysis gives us a good understanding of the results of IM_INDP in Table 6. But for more detailed analysis, we need to identify $f(t)$ by sampling the data of run times.

7.3.3 Results of the Master/Slave Model

Since computation times of TS occupy more than 99 % of the algorithm (Table 2), we expected the master/slave model to show good results in the speedup. However, as shown in Fig. 11, results on the small-size instances in this study (tai40a, tai50a, tai60a, tai50b, tai60b) showed relatively small speedup values against the ideal speedup value of 4. In the figure, the *Speedup* values are defined in Sect. 7.3.1. Results on large-size instances (tai80a, tai100a, tai80b, tai100b, tai150b), the speedup values were nearer to ideal speedup values.

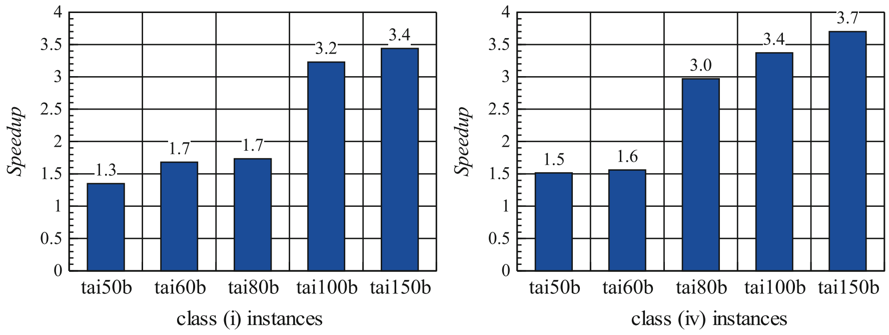


Fig. 11 Results of the master/slave model with four GPUs

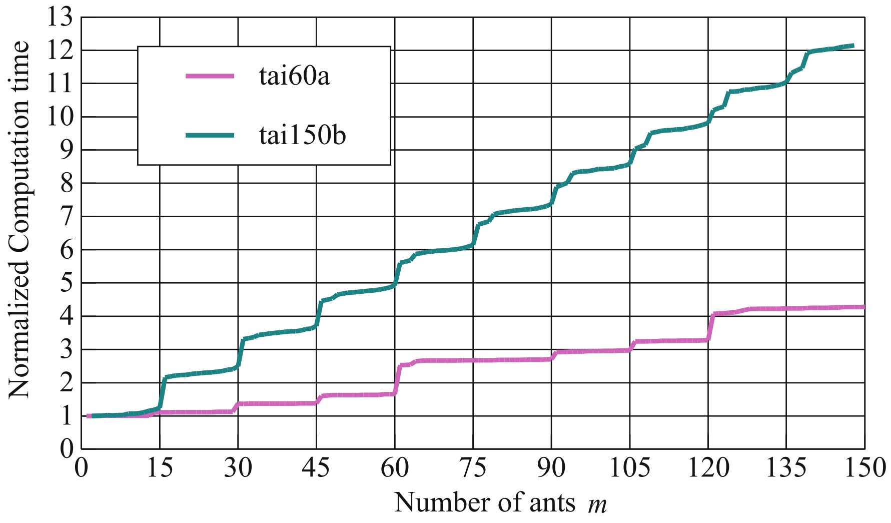


Fig. 12 Computation times for various number of agents on tai60a and tai150b over 10 runs for 10 ACO iterations

Now we will analyze why these results were obtained on the master/slave model. Figure 12 shows the average computation times of tai60a and tai150b over 10 runs for ten ACO iterations by changing the number of agents m from 1 to 150 with step 1. Here, the ACO algorithm is the master/slave model in Sect. 7.2 with a GPU number setting of 1, and the computation time is normalized by the time of $m = 1$. Since the number of MPs of GTX 480 is 15, we can see the computation times increase nearly 15 interval of m. However, the increasing times are different between these two instances.

On tai60a ($n = 60$) instance, the difference of computation times among $1 \leq m \leq 15$, $16 \leq m \leq 30$, $31 \leq m \leq 45$, and $46 \leq m \leq 60$ is very small. In our implementation of TS on a GPU, we assigned one thread block to each agent (solution), and thus the number of agents is identical to the number of thread blocks. In CUDA, multiple blocks are allocated to one MP if computation resources, such as registers, are available. In the execution of tai60a ($n = 60$), this situation occurs and multiple blocks are executed in parallel in one MP. Since in this experiment we set $m = 60$ (the number of agents m equals to the problem size, i.e., $m = n$, see Sect. 7.3.1 and Table 4), the solution assigned to one slave GPU is only 15. This means that the speedup using the master/slave model becomes very small, as was seen in Fig. 11.

On the other hand, on tai150b ($n = 150$) instances, computation times proportionally increase according to m with every 15 intervals. This means that on tai150b, a single thread block is allocated to one MP at the same time, with the resulting speedup shown in Fig. 11. Note here that on this instance, the number of agents assigned to one GPU is $\lfloor 150/4 \rfloor = 37$ and the total agent number of $37 \times 4 = 148$ was used in this experiment.

8 Conclusions

In this chapter, we propose an ACO for solving quadratic assignment problems (QAPs) on a GPU by combining tabu search (TS) in CUDA. In TS on QAPs, there are $n(n-1)/2$ neighbors in a candidate solution. These TS moves form two groups based on computing cost. In one group, the computing of the move cost is $\mathscr{O}(1)$, and in the other group, the computing of the move cost is $\mathscr{O}(n)$. We compute these groups of moves in parallel by assigning the computations to threads of CUDA. In this assignment, we proposed an efficient method which we call *Move-Cost Adjusted Thread Assignment (MATA)* that can reduce disabling time, as far as possible, in each thread of CUDA. As for the ACO algorithm, we use the Cunning Ant System (cAS).

The results showed that GPU computation with MATA showed a promising speedup compared to computation with CPU. Based on MATA, we also implemented two types of parallel algorithms on multiple GPUs to solve QAPs faster. These are the island model and the master/slave model. As for the island model, we used four types of topologies. Although the results of speedup much depend on the instances we used, we showed that the island model with elitist and massive ring connected (IM_ELMR) has a good speedup feature. As for the master/slave model, we observed reasonable speedups for large sizes of instances, where we used large numbers of agents. When we compared the island model and the master/slave model, the island model showed promising speedup values on class (iv) instances of QAP. On the other hand, the master/slave model consistently showed promising speedup values both on classes (i) and (iv) with large-size QAP instances with large numbers of agents.

As regards this comparison, a more intensive analytical study is an interesting future research direction. Implementation using an existing massively parallel platform such as EASEA [16] is also an interesting future research topic.

Acknowledgements This research is partially supported by the Ministry of Education, Culture, Sports, Science and Technology of Japan under Grant-in-Aid for Scientific Research No. 22500215.

References

1. Acan, A.: An external memory implementation in ant colony optimization. Proceedings of the 4th International Workshop on Ant Algorithms and Swarm Intelligence (ANTS-2004) pp. 73–84 (2004)
2. Acan, A.: An external partial permutations memory for ant colony optimization. Proceedings of the 5th European Conf. on Evolutionary Computation in Combinatorial Optimization pp. 1–11 (2005)
3. Alba, E.: Parallel Metaheuristics: A New Class of Algorithms. Wiley, Hoboken (2005)
4. Bai, H., OuYang, D., Li, X., He, L., Yu, H.: MAX-MIN ant system on GPU with CUDA. In: Innovative Computing, Information and Control, Jilin Univ., Changchun, China, pp. 801–804, 2009
5. Burkard, R., Çela, E., Karisch, S., Rendl, F.: QAPLIB - a quadratic assignment problem library (2009). www.seas.upenn.edu/qaplib. Accessed 17 December 2010
6. Cantú-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers, Norwell, MA (2000)
7. Delévacqa, A., Delislea, P., Gravelb, M., Krajeckia, M.: Parallel ant colony optimization on graphics processing units. J. Parallel Distr. Comput. **73**(1), 52–61 (2013)
8. Diego, F., Gómez, E., Ortega-Mier, M., García-Sánchez, Á.: Parallel CUDA architecture for solving the VRP with ACO. In: Industrial Engineering: Innovative Networks, pp. 385–393. Springer, London (2012)
9. Dorigo, M., Maniezzo, V., Colorni, A.: The ant system: Optimization by a colony of cooperating agents. IEEE Trans. Syst. Man. Cybern. B Cybern. **26**(1), 29–41 (1996)
10. Dorigo, M., Stützle, T.: Ant Colony Optimization. MIT Press, Massachusetts (2004)
11. Dorigo, M., Stützle, T.: Ant colony optimization: overview and recent advances. Handbook of Metaheuristics, 2nd edn., pp. 227–263. Springer, New York (2010)
12. Fu, J., Lei, L., Zhou, G.: A parallel ant colony optimization algorithm with GPU-acceleration based on all-in-roulette selection. In: Workshop on Advanced Computational Intelligence, Wuhan Digital Engineering Institute, Wuhan, China, pp. 260–264, 2010
13. Gendreau, M., Potvin, J.: Tabu search. Handbook of Metahewristics, 2nd edn., pp. 41–59. Springer, New York (2010)
14. Glover, F., Laguna, M.: Tabu Search. Kluwer, Boston (1997)
15. Luong, T.V., Melab, N., Talbi, E.G.: Parallel hybrid evolutionary algorithms on GPU. In: IEEE Congress on Evolutionary Computation, Université de Lille 1, Lille, France, pp. 2734–2741, 2010
16. Maitre, O., Krüger, F., Querry, S., Lachiche, N., Collet, P.: EASEA: specification and execution of evolutionary algorithms on GPGPU. Soft Comput. **16**(2), 261–279 (2012)
17. NVIDIA: (2010). www.nvidia.com/object/cuda_home_new.html. Accessed 17 December 2010
18. NVIDIA: (2010). www.nvidia.com/object/fermi_architecture.html. Accessed 17 December 2010
19. NVIDIA: (2010). developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_CProgramming_Guide.pdf. Accessed 17 December 2010
20. Ryoo, S., Rodrigues, C.I., Stone, S.S., Stratton, J.A., Ueng, S.Z., Baghsorkhi, S.S., Hwu, W.: Program optimization carving for GPU computing. J. Parallel Distr. Comput. **68**(10), 1389–1401 (2008)

21. Soca, N., Blengio, J.L., Pedemonte, M., Ezzatti, P.: PUGACE, a cellular evolutionary algorithm framework on GPUs. In: IEEE Congress on Evolutionary Computation, Universidad de la Republica, Montevideo, Uruguay, pp. 3891–3898, 2010
22. Stützle, T., Hoos, H.: Max-Min Ant System. Future Generat. Comput. Syst. **16**(9), 889–914 (2000)
23. Taillard, É.: Robust taboo search for quadratic assignment problem. Parallel Comput. **17**, 443–455 (1991)
24. Taillard, É.: Comparison of iterative searches for the quadratic assignment problem. Location Science **3**(2), 87–105 (1995)
25. Taillard, É.: taboo search tabou_qap code (2004). http://mistic.heig-vd.ch/taillard/codes.dir/tabou_qap.cpp
26. Tsutsui, S.: *c*AS: Ant colony optimization with cunning ants. Parallel Problem Solving from Nature, pp. 162–171. Springer, Berlin (2006)
27. Tsutsui, S., Fujimoto, N.: Solving quadratic assignment problems by genetic algorithms with GPU computation: a case study. In: Genetic and Evolutionary Computation Conference (Companion), pp. 2523–2530. ACM, New York (2009)
28. Tsutsui, S., Fujimoto, N.: An analytical study of GPU computation for solving QAPs by parallel evolutionary computation with independent run. In: IEEE Congress on Evolutionary Computation, Hannan University, Matsubara, Japan, pp. 889–896, 2010
29. Tsutsui, S., Fujimoto, N.: ACO with tabu search on a GPU for solving QAPs using move-cost adjusted thread assignment. In: Genetic and Evolutionary Computation Conference, pp. 1547–1554. ACM, Dublin (2011)

New Ideas in Parallel Metaheuristics on GPU: Systolic Genetic Search

Martín Pedemonte, Francisco Luna, and Enrique Alba

Abstract This chapter presents an in-depth study of a novel parallel optimization algorithm specially designed to run on Graphic Processing Units (GPUs). The underlying operation relates to systolic computing and is inspired by the systolic contraction of the heart that makes possible blood circulation. The algorithm, called Systolic Genetic Search (SGS), is based on the synchronous circulation of solutions through a grid of processing units and tries to profit from the parallel architecture of GPUs to achieve high time performance. SGS has shown not only to numerically outperform a random search and two genetic algorithms for solving the Knapsack Problem over a set of increasingly sized instances, but also its parallel implementation can obtain a runtime reduction that, depending on the GPU technology used, can reach more than 100 times. A study of the performance of the parallel implementation of SGS on four different GPUs has been conducted to show the impact of the Nvidia's GPU compute capabilities on the runtimes of the algorithm.

1 Introduction

The design of parallel metaheuristics [1], thanks to the increasing power offered by modern hardware architectures, is a natural research line in order to reduce the execution time of the algorithms, especially when solving problems of a high dimension, highly restricted, and/or time bounded. Parallel metaheuristics are often based

M. Pedemonte (✉)
Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay
e-mail: mpedemon@fing.edu.uy

F. Luna · E. Alba
Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Málaga, Spain
e-mail: flv@lcc.uma.es; eat@lcc.uma.es

on new different search patterns that are later implemented on physically parallel hardware, then improving the quality of results obtained by traditional sequential algorithms and also reducing their execution time. As a consequence, research on parallel metaheuristics has grown substantially in recent years, motivated by the excellent results obtained in their application to the resolution of problems in search, optimization, and machine learning.

Among parallel metaheuristics, Parallel Evolutionary Algorithms (PEAs) have been extensively adopted and are nowadays quite popular, mainly because Evolutionary Algorithms (EAs) are naturally prone to parallelism. For this reason, the study of parallelization strategies for EAs has laid the foundations for working in parallel metaheuristics. The most usual criterion for categorizing PEAs distinguishes three categories [3, 17]: the master–slave model (functional distribution of the algorithm), the distributed or island model (the population is partitioned into a small number of subpopulations that evolve in semi-isolation) [3], and the cellular model (the population is structured in overlapping neighborhoods with limited interactions between individuals) [2]. Although tied to hardware (clusters and multicores, especially), the previous research actually deals a lot with decentralization of search techniques, leaving the running platform sometimes as a background aid just to reduce time.

The use of Graphics Processing Units (GPUs) for general purpose computing (GPGPU) has experienced tremendous growth in recent years. This growth has been based on its wide availability, low economic cost, and inherent parallel architecture and also on the emergence of general purpose programming languages, such as CUDA [11, 20]. This fact has also motivated many scientists from different fields to take advantage of the use of GPUs in order to tackle general problems in various fields like numerical linear algebra [8], databases [21], model order reduction [5], and scientific computing [11].

GPGPU also represents an inspiring domain for research in parallel metaheuristics. Naturally, the first works on GPUs have gone in the direction of implementing the mentioned three categories of PEAs on this new kind of hardware [14]. Thus, many results show the time savings of running master–slave [18], distributed [30], and cellular [28, 29] metaheuristics on GPU, mainly Genetic Algorithms (GAs) [18, 22] and Genetic Programming [9, 14–16, 19] but also other types of techniques like Ant Colony Optimization [6], Differential Evolution [7], and Particle Swarm Optimization [31].

A different approach is not to take a regular existing family of algorithms and port it to a GPU but to propose new techniques based on the architecture of GPUs. Recently, two optimization algorithms have been proposed, Systolic Neighborhood Search (SNS) [4] and Systolic Genetic Search (SGS) [23], based on the idea of systolic computing.

The concept *systolic computing* was developed at Carnegie-Mellon University by Kung and Leiserson [12, 13]. The basic idea focuses on creating a network of different simple processors or operations that rhythmically compute and pass data through the system. Systolic computation offers several advantages, including simplicity, modularity, and repeatability of operations. This kind of architecture also offers transparent, understandable, and manageable but still quite powerful parallelism. However, this architecture had difficulties in the past, since building systolic computers was not easy and, especially, because programming high-level algorithms on such a low-level hardware was hard, error prone, and too manual. Now with GPUs we can avoid such problems and get only the advantages.

In the leading work on optimization algorithms using systolic computing, the SNS algorithm has been proposed, based on local search [4]. Then, we have explored a new line of research that involved more diverse operations. Therefore, we have proposed a novel parallel optimization algorithm that is based on the synchronous circulation of solutions through a grid of cells and the application of adapted evolutionary operators. We called this algorithm SGS [23]. In SGS [23], solutions flow across processing units according to a synchronous execution plan. When two solutions meet in a cell, adapted evolutionary operators (crossover and mutation) are applied to obtain two new solutions that continue moving through the grid. In this way, solutions are refined again and again by simple low-complexity search operators.

The goal of this work is to evaluate one of the new algorithms that can be proposed based on the premises of SGS. In the first place, we evaluate SGS regarding the quality of solutions obtained. Then, we focus on analyzing the performance of the GPU implementation of the algorithm on four different cards. This evaluation shows how changes produced in GPUs in a period of only 2 years impact on the runtime of the algorithm implemented. Considering the results in a broader perspective, it also shows the impressive progress on the design and computing power of these devices.

This chapter is organized as follows. The next section introduces the SGS algorithm. Section 3 describes the implementation of the SGS on a GPU. Then, Sect. 4 presents the experimental study considering several instances of the Knapsack Problem (KP) evaluated over four different Nvidia's GPUs. Finally, in Sect. 5, we outline the conclusions of this work and suggest future research directions.

2 Systolic Genetic Search

In a SGS algorithm, the solutions are synchronously pumped through a bidimensional grid of cells. The idea is inspired by the systolic contraction of the heart that makes possible that ventricles eject blood rhythmically according to the metabolic needs of the tissues.

It should be noted that although SGS has points of contact with the behavior of Systolic Arrays (SAs) [10] and both ideas are inspired by the same biological phenomenon, they have an important difference. In SAs, the data is pumped from the memory through the units before returning to the memory, whereas in SGSs the data only circulates through the cells, all time.

At each step of SGS, as it is shown in Fig. 1, two solutions enter each cell, one from the horizontal ($S_{\mathrm{H}}^{i,j}$) and one from the vertical ($S_{\mathrm{V}}^{i,j}$), where adapted evolutionary/genetic operators are applied to obtain two new solutions that leave the cell, one through the horizontal ($S_{\mathrm{H}}^{i,j+1}$) and one through the vertical ($S_{\mathrm{V}}^{i+1,j}$).

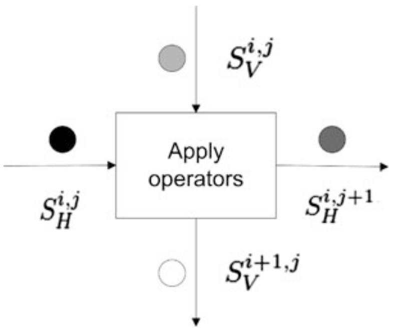


Fig. 1 Systolic cell

Algorithm 1 Systolic Genetic Search

```
1: for all $c$ Cell do
2:     $c.h$ =generateRandomSolution();
3:     $c.v$ =generateRandomSolution();
4: end for
5: for $i = 1$ to $maxGeneration$ do
6:     for all $c$ Cell do
7:         $(temp_H, temp_V)$ =crossover($c.h, c.v$);
8:         $temp_H$ =mutation($temp_H$);
9:         $temp_V$ =mutation($temp_V$);
10:        $c_1$ =calculateNextHorizontalCell($c$);
11:        $c_2$ =calculateNextVerticalCell($c$);
12:        $temp_H$ =elitism($c.h, temp_H$);
13:        $temp_V$ =elitism($c.v, temp_V$);
14:        moveSolutionToCell($temp_H, c_1.h$);
15:        moveSolutionToCell($temp_V, c_2.v$);
16:    end for
17: end for
```

The pseudocode of the SGS algorithm is presented in Algorithm 1. Each cell applies the basic evolutionary search operators (crossover and mutation) but to different, preprogrammed fixed positions of the tentative solutions that circulate throughout the grid. In this way, the search process is actually structured. The cells use elitism to pass on to the next cells the best solution among the incoming solution and the newly generated one by the genetic operators. The incorporation of elitism is critical, since there is no global selection process as in standard EAs.

Each cell sends the outgoing solutions to the next horizontal and vertical cells that are previously calculated, as shown in the pseudocode. Thus it is possible to define different cell interconnection topologies. In this work, and in order to make the idea of the method more easily understood, we have used a bidimensional toroidal grid of cells, as shown in Fig. 2.

In order to better illustrate the working principles of SGS, let us consider that the problem addressed can be encoded as binary string, with bit-flip mutation and two-point crossover as evolutionary search operators. We want to remark that the idea of the proposal can be adapted to other representations and operators. In this case (binary representation), the positions in which operators are applied in each cell

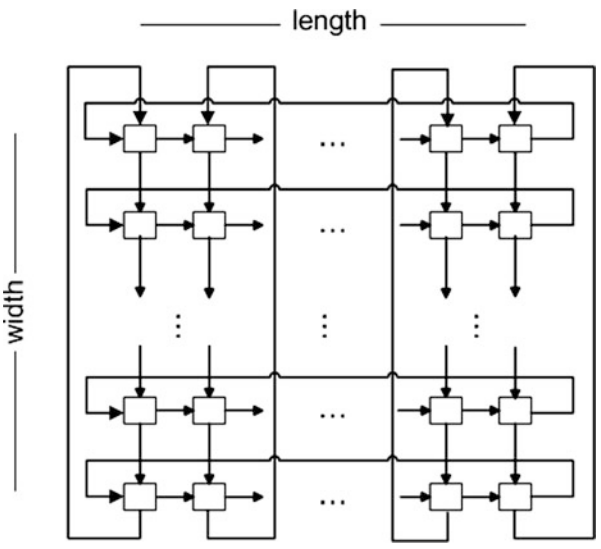


Fig. 2 Two-dimensional toroidal grid of cells

are computed by considering the coordinates occupied by the cell in the grid, thus avoiding the generation of random numbers. Some key aspects of the algorithm such as the size of the grid and the calculation of the mutation point and the crossover points are discussed next.

2.1 *Size of the Grid*

The length and width of the grid should allow the algorithm to have a good exploration, but without increasing the population to sizes that compromise performance. In order to generate all possible mutation points at each row, the grid length is l (the length of the tentative solutions), and therefore each cell in a row modifies a different position of the solutions. For the same reason, the natural value for the width of the grid is also l. However, in order to keep the total number of solutions of the population within an affordable value, the width of the grid has been reduced to $\tau = \lceil \lg l \rceil$. Therefore, the number of solutions of the population is $2 \times l \times \tau$ (two solutions for each cell).

2.2 *Mutation*

The mutation operator always changes one single bit in each solution. Figure 3 shows the general idea followed to distribute the points of mutation over the entire grid.

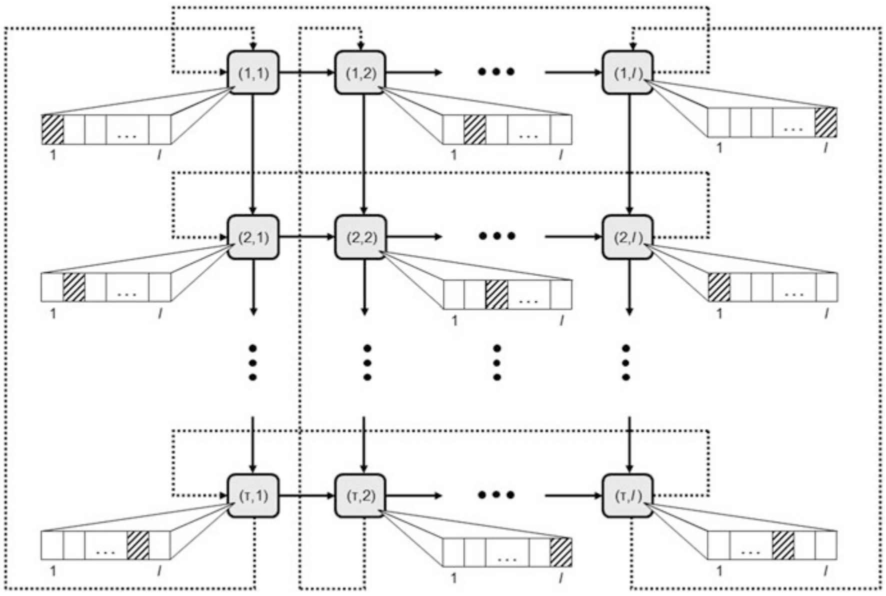


Fig. 3 Distribution of mutation points across the grid

Each cell in the same row mutates a different bit in order to generate diversity by encouraging the exploration of new solutions. On the other hand, cells in the same column should not mutate the same bit in order to avoid deteriorating the search capability of the algorithm. For this reason, the mutation points on each row are shifted $\mathrm{div}(l, \tau)$ places.

Figure 4 shows an example of the mutation points of the cells from column j. The general formula for calculating the mutation point of a cell (i, j) is:

$$1 + \left(\left(i - 1\right)\frac{l}{\tau} + j - 1\right) \mod l \ , \tag{1}$$

where mod is the modulus of the integer division and the division in the formula is an integer division.

2.3 Crossover

Figure 5 shows the general idea followed to distribute the crossover points over the entire grid.

For the first crossover point two different values are used in each row, one for the first $l/2$ cells and another one for the last $l/2$ cells. These two values differ by $\mathrm{div}(l, 2\tau)$, while cells of successive rows in the same column differ by $\mathrm{div}(l, \tau)$. This allows using a large number of different values for the first crossover point following

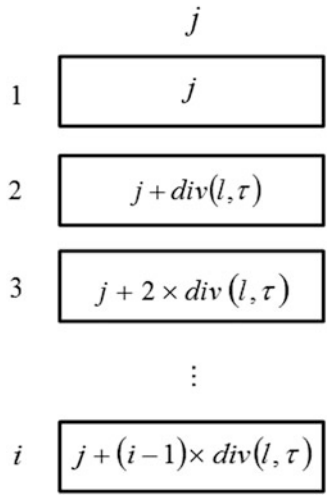


Fig. 4 Mutation points for column j

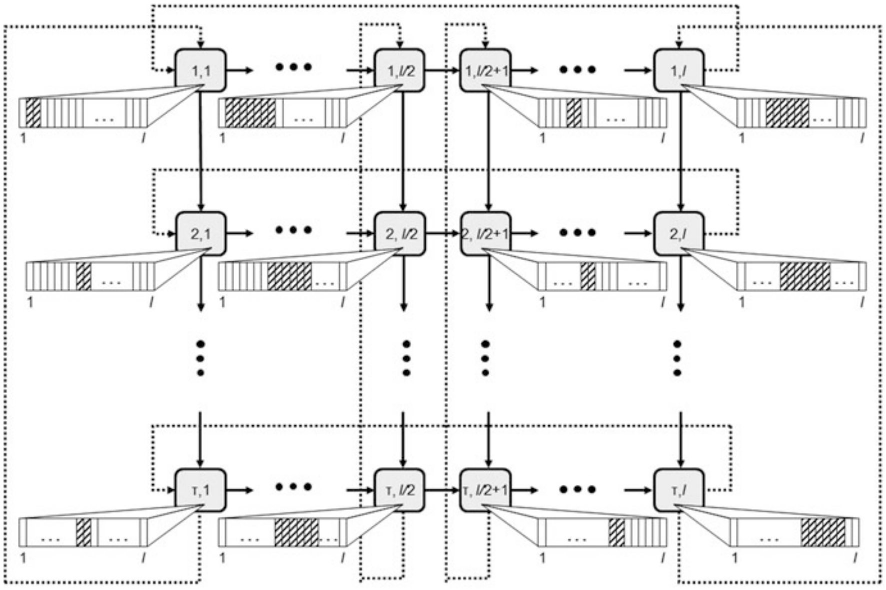


Fig. 5 Distribution of crossover points across the grid

a pattern known a priori. Figure 6 illustrates the first crossover point calculation. The general formula for calculating the first crossover point of a cell (i, j) is:

$$2 + \frac{l}{\tau}(i - 1) + \frac{j - 1}{\frac{l}{2}} \frac{l}{2\tau} \quad , \tag{2}$$

where all the divisions in the formula are integer divisions.

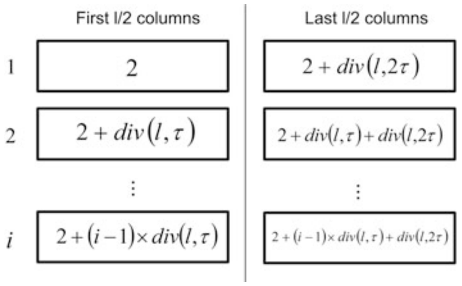


Fig. 6 First crossover point calculation

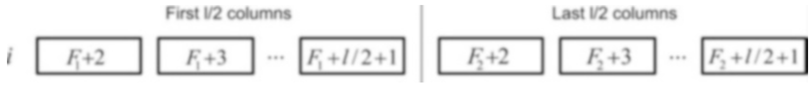


Fig. 7 Second crossover point calculation

For the second crossover point, the distance to the first crossover point increases with the column, from a minimum distance of 2 positions to a maximum distance of $\text{div}(l, 2) + 1$ positions. In this way, cells in successive columns exchange a larger portion of the solutions. Figure 7 illustrates the second crossover point calculation, being F_1 the first crossover point for the first $l/2$ cells and F_2 the first crossover point for the last $l/2$ cells. If the value of second crossover point is smaller than the first one, the values are swapped. The general formula for calculating the second crossover point of a cell (i, j) is:

$$1 + \left(3 + \frac{l}{\tau}\left(i - 1\right) + \frac{j-1}{\frac{l}{2}}\frac{l}{2\tau} + \left((j-1) \mod \frac{l}{2}\right)\right) \mod l \ , \tag{3}$$

where all the divisions in the formula are integer divisions.

2.4 Exchange of Directions

As the length of the grid is larger than the width of the grid, the solutions moving through the vertical axis would be limited to only τ different mutation and crossover points, while those moving horizontally use a wider set of values. In order to avoid this situation, which can cause the algorithm not to reach its full potential, every τ iterations the two solutions being processed in each cell exchange their directions. That is, the solution moving horizontally leaves the cell through the vertical axis, while the one moving vertically continues through the horizontal.

3 SGS Implementation

This section is devoted to presenting how SGS has been deployed on a GPU. In this work we study the performance of the GPU implementation of SGS in four different Nvidia's devices, so we provide a general snapshot of GPU devices and briefly comment some important differences between the devices used. Then, all the implementation details are introduced.

3.1 Graphics Processing Units

The architecture of GPUs was designed following the idea of devoting more transistors to computation than traditional CPUs [11]. As a consequence, current GPUs have a large number of small cores and are usually considered *many-cores* processors.

The CUDA architecture abstracts these computing devices as a set of shared memory multiprocessors (MPs) that are able to run a large number of threads in parallel. Each MP follows the Single Instruction Multiple Threads (SIMT) parallel programming paradigm. SIMT is similar to SIMD but in addition to data-level parallelism (when threads are coherent) allows thread-level parallelism (when threads are divergent) [11, 20].

When a *kernel* is called in CUDA, a large number of threads are generated on the GPU. The threads are grouped into blocks that are run concurrently on a single MP. The blocks are divided into *warps* that are the basic scheduling unit in CUDA and consist of 32 consecutive threads. Threads can access data on multiple memory spaces during their lifetime, being the most commonly used: registers, global memory, shared memory, and constant memory. Registers and shared memory are fast memories, but registers are only accessible by each thread, while shared memory can be accessed by any thread of a block. The global memory is the slowest memory on the GPU and can be accessed by any executing thread. Finally, constant memory is fast although is read-only for the device [27].

The compute capability allows knowing some features of a GPU device. In this work we use a GPU with compute capability 1.1 (a GeForce 9800 GTX+), two with compute capability 1.3 (a Tesla C1060 and a GeForce GTX 285), and one with compute capability 2.0 (a GeForce GTX 480). Table 1 presents some of the most important features for devices with the compute capabilities considered in this work [20].

It should also be noted that the global memory access has improved significantly in devices with newer compute capabilities [20]. In devices with compute capability 1.1, each half-warp must read 16 words in sequence that lie in the same segment to produce a single access to global memory for the half-warp, which is known as coalesced access. Otherwise, when a half-warp reads words from different segments (misaligned reads) or from the same segment but the words are not accessed

Table 1 Features according to compute capabilities of the device

	Compute capability		
Technical specification	1.1	1.3	2.0
Maximum number of threads per block	512	512	1,024
Number of 32-bit registers per multiprocessor (K)	8	16	32
Maximum amount of shared memory per multiprocessor (KB)	16	16	48
Constant memory size (KB)	64	64	64
Number of CUDA cores per multiprocessor	8	8	32
Warp schedulers	1	1	2

sequentially, it produces an independent access to global memory for each thread of the half-warp. The access to global memory was improved considerably in devices with compute capabilities 1.2 and 1.3, and threads from the same half-warp can read any words in any order in a segment with a single access (including when several threads read the same word). Finally, the devices with compute capabilities 2.0 include cached access to global memory.

3.2 Implementation Details

Figure 8 shows the structure of the GPU implementation of the SGS algorithm. The execution starts with the initialization of the population that runs in the CPU and solutions are transferred to the global memory of the GPU. Thus, the results obtained working with the same seed are exactly the same for the CPU and GPU implementations. This restriction can increase the runtime of the GPU implementation but simplifies the experimental analysis by focusing in the performance. Then, the constant data required for computing the fitness values is copied to the constant memory of the GPU. At each iteration, the crossover and mutation operators (*crossoverAndMutation* kernel), the fitness function evaluation (*evaluate* kernel), and the elitist replacement (*elitism* kernel) are executed on the GPU. Additionally, the exchange of directions operator (*exchange* kernel) is applied on the GPU in given iterations (when $\text{div}(generation, \tau) == 0$). Finally, when the algorithm reaches the stop condition, the population is transferred back from the GPU to the CPU.

The kernels are implemented following the idea used in [22], in which operations are assigned to a whole block and all the threads of the block cooperate to perform an operation. If the solution length is larger than the number of threads in the block, each thread processes more than one element of the solution but the elements used by a single thread are not contiguous. Thus, each operation is applied to a solution in chunks of the size of the thread block (T in the following figure), as it is shown in Fig. 9.

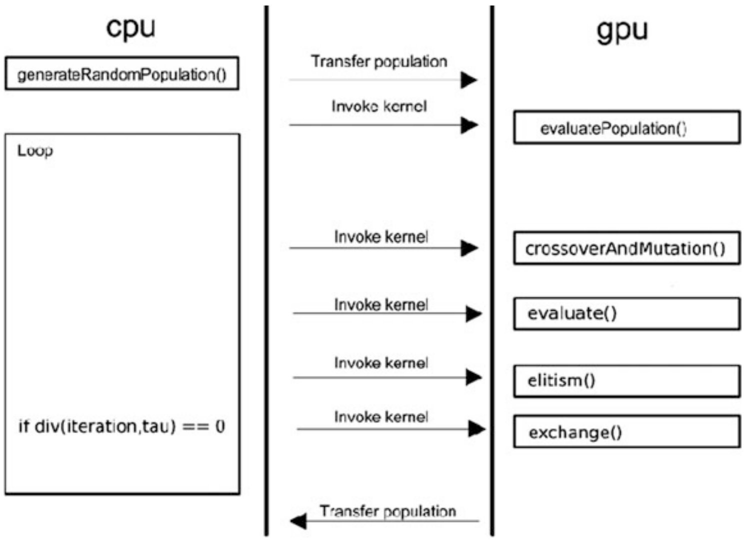


Fig. 8 Structure of GPU implementation

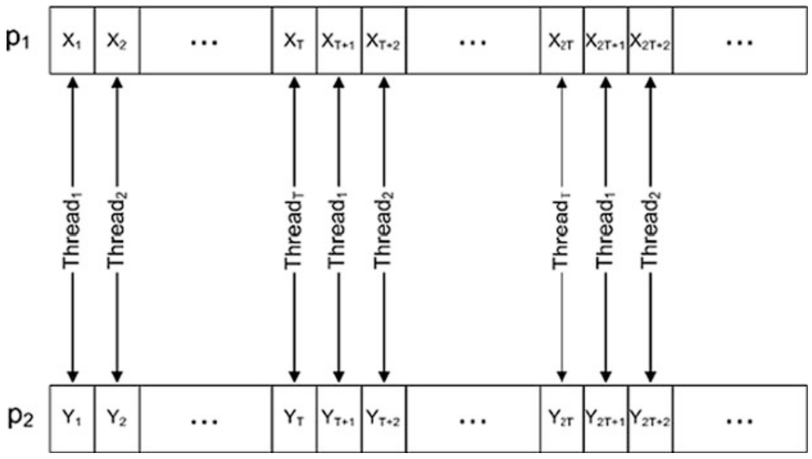


Fig. 9 Threads organization

$l \times \tau$ Blocks are launched for the execution of *crossoverAndMutation* kernel (one for each cell). Initially, the global memory location of the solutions of the cell, the global memory location where the resulting solutions should be stored, the crossover points, and the mutation point are calculated from the block identifiers. Then, the crossover is applied to the two solutions, processing the solution components in chunks of size of the thread block. Finally, the thread zero of the block performs the mutation of both solutions.

The *elitism*, *exchange*, and *evaluate* kernels follow the same idea regarding the thread organization and behavior and are also launched for execution organized in $l \times \tau$ blocks. The fitness function evaluation (*evaluate* kernel) uses data from the constant memory of the GPU and an auxiliary structure in shared memory to store the partial fitness values computed by each thread. Finally, it applies a reduction to calculate the full fitness value.

4 Experimental Results

This section describes the problem used for the experimental study, the parameters setting, and the execution platforms. Then, the results obtained are presented and analyzed.

4.1 Test Problem: The Knapsack Problem

The KP is a classical combinatorial optimization problem that belongs to the class of $\mathcal{NP}$-hard problems [25]. It is defined as follows. Given a set of n items, each of them having associated an integer value p_i called profit or value and an integer value w_i known as weight, the goal is to find the subset of items that maximizes the total profit keeping the total weight below a fixed maximum capacity (W) of the knapsack or bag. It is assumed that all profits and weights are positive, that all the weights are smaller than W (items heavier than W do not belong to the optimal solution), and that the total weight of all the items exceeds W (otherwise, the optimal solution contains all the items of the set).

The most common formulation of the KP is as the integer programming model presented in (4a)–(4c), being x_i the binary decision variables of the problem that indicate whether the item i is included or not in the knapsack.

$$\text{(KP)} \quad \text{maximize} \quad f(\mathbf{x}) = \sum_{i=1}^{n} p_i x_i \tag{4a}$$

$$\text{subject to:} \quad \sum_{i=1}^{n} w_i x_i \leqslant W \tag{4b}$$

$$x_i \in \{0, 1\}, \forall i = 1, \ldots, n \tag{4c}$$

It should be noted that there are very efficient specific heuristics for solving the KP [26], but we have selected this problem because it is a classical problem that can be used to evaluate the proposed algorithm against similar techniques.

Table 2 Knapsack instances used in the experimental evaluation and their exact optimal solutions

Instance	n	R	W	Profit of opt. sol.	Weight of opt. sol.
100–1,000	100	1,000	1,001	5,676	983
100–10,000	100	10,000	10,001	73,988	9,993
200–1,000	200	1,000	1,001	10,867	1,001
200–10,000	200	10,000	10,001	100,952	9,944
500–1,000	500	1,000	1,001	19,152	1,000
500–10,000	500	10,000	10,001	153,726	9,985
1,000–1,000	1,000	1,000	1,001	27,305	1,000
1,000–10,000	1,000	10,000	10,001	231,915	9,996

Table 2 presents the instances used in this work. These instances have been generated with no correlation between the weight and the profit of an item (i.e., w_i and p_i are chosen randomly in $[1, R]$) using the generator described in [25]. The Minknap algorithm [24], an exact method based on dynamic programming, was used to find the optimal solution of each of the instances.

The algorithms studied use a penalty approach to manage infeasibility. In this case, the penalty function subtracts W from the total profit for each unit of the total weight that exceeds the maximum capacity. The formula for calculating the fitness for infeasible solutions is:

$$f(\mathbf{x}) = \sum_{i=1}^{n} p_i x_i - \left(\sum_{i=1}^{n} w_i x_i - W \right) \times W \ . \tag{5}$$

4.2 Algorithms

In addition to the algorithm proposed in this chapter, we have included two algorithms, a random search (RS) and a simple GA with and without elitism, in order to compare the quality of the solutions obtained. The former is considered as a sanity check, just to show that our algorithmic proposals are more intelligent than a pure random sampling. On the other hand, the GAs have been chosen because of their popularity in the literature and also they share the same basic search operators (crossover and mutation) so we can properly compare the underlying search engine of the techniques. Briefly, the details of the algorithms used in this work are:

- RS: The RS algorithm processes the items sequentially. If an item in the knapsack exceeds the maximum capacity, it is discarded. Otherwise, the item is included at random with probability 0.5.
- Simple Genetic Algorithm (SGA): It is a generational GA with binary tournament, two-point crossover, and bit-flip mutation.

Table 3 Hardware platforms used for experimental analysis

GPU	CPU	RAM memory (GB)
GeForce 9800 GTX+	Pentium Dual-Core E5200 at 2.50 GHz	2
Tesla C1060	Quad Core Xeon E5530 at 2.40 GHz	48
GeForce GTX 285	Quad Core i7-920 at 2.67 GHz	4
GeForce GTX 480	Core 2 Duo E7400 at 2.80 GHz	2

- Elitist Genetic Algorithm (EGA): It is similar to SGA but with elitist replacement, i.e., each child solution replaces its parent solution only if it has a better (higher) fitness value.
- SGS: The SGS algorithm presented in Sect. 2.

Each of the algorithms studied has been implemented on CPU. The implementation is straightforward so no further details are provided. Additionally, SGS has been implemented on GPU. Both the CPU and GPU implementations of SGS have exactly the same behavior.

4.3 Parameters Setting and Test Environment

The SGA and EGA parameter values used are 0.9 for the crossover probability and $1/l$ for the mutation probability, where l is the length of the tentative solutions. The population size and the number of iterations are defined by considering the features of SGS, using exactly the same values for the two GA versions. In this study, the population size is $2 \times l \times \tau$ and the number of iterations is $10 \times l$ (recall that $\tau = \lceil \lg l \rceil$). Finally, $2 \times l \times \tau \times 10 \times l$ solutions are generated by *RS* to perform a fair comparison.

The execution platform for the CPU versions of the algorithms is a PC with a Quad Core Xeon E5530 processor at 2.40 GHz with 48 GB RAM using Linux operating system. These CPU versions have been compiled using the `-O2` flag and are executed as single-thread applications.

Several GPU cards have been used to evaluate the GPU version of the algorithm. Each Nvidia's GPU is connected to a PC with different features (see Table 3 for the main characteristics of the entire computing platform). All the PCs use Linux operating system. Table 4 provides the details of each one of the GPUs used in the evaluation (PTMB stands for Peak of Theoretical Memory Bandwidth and MP for Multiprocessors) [20]. The GPU versions were also compiled using the `-O2` flag. Considering the features of all the GPU platforms, executions with 32, 64, 128, and 256 threads per block were made.

All the results reported in the next subsection are over 50 independent runs and rounded to two significant figures. The CPU and GPU versions of SGS were executed using the same seeds in order that the numerical results of the two versions are exactly the same. The transference times of data between CPU and GPU are always included in the reported total runtime of the GPU version.

Table 4 GPU cards used for experimental analysis

GPU	MP	CUDA cores	Processor clock (MHz)	GPU memory (GB)	PTMB (GB/s)	Compute capability
GeForce 9800 GTX+	16	128	1,836	0.5	70.4	1.1
Tesla C1060	30	240	1,296	4	102	1.3
GeForce GTX 285	30	240	1,476	1	159	1.3
GeForce GTX 480	15	480	1,401	1.5	177.4	2.0

4.4 *Experimental Analysis*

This section describes the experimental analysis conducted to validate SGS. The experiments include a study of the numerical efficiency of the algorithm proposed and a study of the performance of the parallel GPU implementation of SGS.

4.4.1 Numerical Efficiency

Table 5 presents the experimental results regarding the quality of the solutions obtained. The table includes the best and worst solution found, the average solution, the standard deviation, and the number of times the optimal solution is found (#Hits).

The results obtained show that SGS is able to find the optimal solution in 7 out of the 8 instances for every run and in the remaining instance (1,000–1,000) in 45 out of 50 trials. EGA also performs well, finding the optimal solution for all the instances, but it is noticeable that SGS is clearly superior to EGA in five instances (200–10,000, 500–1,000, 500–10,000, 1,000–1,000 and 1,000–10,000) concerning the hit rate. Figure 10 graphically displays this fact. It is clear that the structured search of SGS has allowed this algorithm to identify the region where the optimal solution is located for the considered instances. SGS is likely to provide a more robust search under this experimental conditions. On the other hand, SGA only reaches the optimal solution in one of the smallest instance and generally cannot obtain good solutions. Finally, RS presents uncompetitive results with the other algorithms, which means that SGS performs a rather intelligent exploration of the search space. Within the context of this experimental evaluation, the potential of the algorithm proposed in this work regarding the quality of the obtained solutions has been shown.

Table 6 shows the mean runtime in seconds and the standard deviation for SGA, EGA, and SGS executed on CPU. The runtime of RS is not included in the table due to the poor numerical results obtained. The results show that SGS is the best performing algorithm. This is mainly caused because the crossover and mutation points of each cell are calculated from its position on the grid, thus avoiding the generation of random numbers during the execution of the algorithm.

Table 5 Experimental results on CPU

Instance	Algorithm	Best	Worst	Mean$_{\pm\text{Std}}$	#Hits
100–1,000	RS	4,488	4,040	$4086.86_{\pm 85.59}$	0
	SGA	5,494	5,000	$5273.58_{\pm 133.88}$	0
	EGA	**5,676**	5,521	$5670.86_{\pm 25.99}$	48
	SGS	**5,676**	**5,676**	–	**50**
100–10,000	RS	60,568	51,499	$54389.30_{\pm 1776.28}$	0
	SGA	73,988	63,809	$67417.88_{\pm 2049.59}$	1
	EGA	**73,988**	**73,988**	–	**50**
	SGS	**73,988**	**73,988**	–	**50**
200–1,000	RS	6,020	5,270	$5599.54_{\pm 165.37}$	0
	SGA	10,464	8,692	$9162.54_{\pm 287.83}$	0
	EGA	**10,867**	10,805	$10863.36_{\pm 14.56}$	47
	SGS	**10,867**	**10,867**	–	**50**
200–10,000	RS	75,160	67,234	$71321.76_{\pm 1768.17}$	0
	SGA	95,770	87,195	$90110.86_{\pm 1833.58}$	0
	EGA	**100,952**	1,00,681	$100938.42_{\pm 38.62}$	29
	SGS	**100,952**	**100,952**	–	**50**
500–1,000	RS	7,704	6,920	$7241.82_{\pm 173.60}$	0
	SGA	16,062	14,198	$14850.76_{\pm 408.36}$	0
	EGA	**19,152**	19,003	$19129.88_{\pm 37.38}$	34
	SGS	**19,152**	**19,152**	–	**50**
500–10,000	RS	90,449	79,086	$83063.16_{\pm 2202.37}$	0
	SGA	125,643	115,003	$119379.78_{\pm 2552.46}$	0
	EGA	**153,726**	153,048	$153571.24_{\pm 165.13}$	23
	SGS	**153,726**	**153,726**	–	**50**
1,000–1,000	RS	9167	7685	$8048.40_{\pm 289.26}$	0
	SGA	20,929	18,605	$19377.40_{\pm 465.07}$	0
	EGA	**27,305**	27,172	$27252.34_{\pm 41.92}$	16
	SGS	**27,305**	**27,236**	$27299.48_{\pm 17.11}$	**45**
1,000–10,000	RS	104,916	92,906	$98130.06_{\pm 2626.53}$	0
	SGA	176,900	157,820	$166420.80_{\pm 4022.98}$	0
	EGA	**231,915**	229,970	$231515.06_{\pm 636.43}$	35
	SGS	**231,915**	**231,915**	–	**50**

4.4.2 Parallel Performance

Table 7 shows the mean runtime in seconds and the standard deviation of SGS executed on the GeForce 9800 GTX+, the Tesla C1060, the GeForce GTX 285, and the GeForce GTX 480.

Let us start with an analysis of the performance by graphic card. The first results examined are those obtained by the SGS deployment on the oldest of the GPU available in our laboratory, the GeForce 9800 GTX+. The best performance of the algorithm is obtained when using 32 threads per block for all the instances

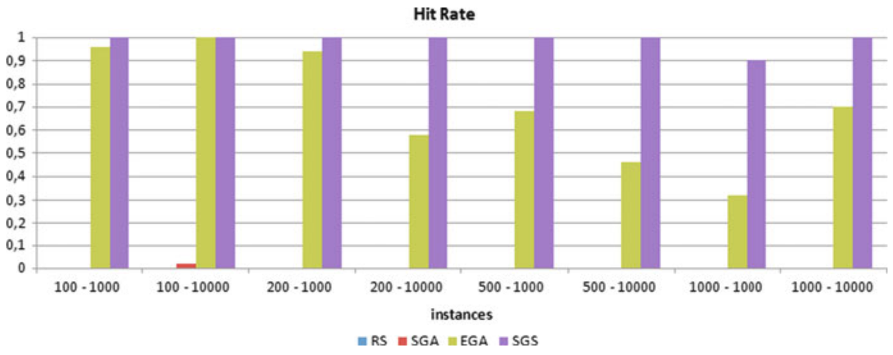


Fig. 10 Hit rate reached by all the algorithms evaluated (no bar means zero hit rate)

Table 6 Runtime in seconds of CPU versions (mean$_{\pm \text{std}}$)

Instance	SGA	EGA	SGS
100–1,000	$2.03_{\pm 0.13}$	$2.22_{\pm 0.05}$	$\mathbf{0.98_{\pm 0.01}}$
100–10,000	$2.18_{\pm 0.90}$	$2.32_{\pm 0.01}$	$\mathbf{1.03_{\pm 0.01}}$
200–1,000	$17.40_{\pm 1.08}$	$19.09_{\pm 0.58}$	$\mathbf{7.91_{\pm 0.47}}$
200–10,000	$17.98_{\pm 1.00}$	$18.80_{\pm 0.06}$	$\mathbf{8.09_{\pm 0.49}}$
500–1,000	$295.18_{\pm 17.83}$	$314.16_{\pm 6.07}$	$\mathbf{116.24_{\pm 3.54}}$
500–10,000	$298.51_{\pm 18.24}$	$308.82_{\pm 1.21}$	$\mathbf{115.29_{\pm 5.60}}$
1,000–1,000	$2591.67_{\pm 126.15}$	$2785.61_{\pm 86.42}$	$\mathbf{915.74_{\pm 12.78}}$
1,000–10,000	$2606.81_{\pm 152.53}$	$2740.27_{\pm 70.99}$	$\mathbf{901.12_{\pm 16.95}}$

studied. In general, the results reached are poor as the reduction in execution time is less than two when comparing to the runtime of the CPU version. However, these results should be put in perspective. The GPU used is quite old and has very limited capacities, while the CPU used is a powerful server with a large amount of RAM. The main reason of such a poor performance of the algorithm on this GPU is how the access to global memory is handled by devices with compute capability 1.1. Indeed, in the implemented algorithm, the solutions are stored sequentially in global memory and may have an arbitrary length (the length is given by the instance being addressed).

When solutions are read from global memory, the same word should be accessed by different threads of the half-warp (data is stored in 8 bits and the size of the minimum addressable word is 32 bits) and there are misaligned accesses. For these reasons, each thread of the half-warp accesses independently to global memory instead of coalescing the access to global memory for the whole half-warp. This was improved in devices with larger compute capability, as it was commented in Sect. 3.1, and it is verified in the results of the following cards.

The best performance of the algorithm when executed on the Tesla C1060 is obtained when using 32 threads per block for smallest instances and 64 threads per block for the rest of the instances. The results show an impressive reduction of up

Table 7 Runtime in seconds of SGS on GPU (mean$_{\pm \text{std}}$), highlighted in bold the shortest runtime for each GPU in each instance

		Threads per block			
Instance	GPU	32	64	128	256
100–1,000	9800 GTX+	$\mathbf{0.86_{\pm 0.02}}$	$0.92_{\pm 0.02}$	$1.15_{\pm 0.01}$	$1.83_{\pm 0.01}$
	Tesla C1060	$\mathbf{0.30_{\pm 0.01}}$	$\mathbf{0.30_{\pm 0.01}}$	$0.35_{\pm 0.01}$	$0.46_{\pm 0.01}$
	GTX 285	$\mathbf{0.21_{\pm 0.01}}$	$0.22_{\pm 0.01}$	$0.26_{\pm 0.01}$	$0.35_{\pm 0.01}$
	GTX 480	$\mathbf{0.19_{\pm 0.01}}$	$\mathbf{0.19_{\pm 0.01}}$	$0.20_{\pm 0.01}$	$0.24_{\pm 0.01}$
100–10,000	9800 GTX+	$\mathbf{0.86_{\pm 0.01}}$	$0.92_{\pm 0.01}$	$1.15_{\pm 0.01}$	$1.83_{\pm 0.01}$
	Tesla C1060	$\mathbf{0.30_{\pm 0.01}}$	$\mathbf{0.30_{\pm 0.01}}$	$0.35_{\pm 0.01}$	$0.46_{\pm 0.01}$
	GTX 285	$\mathbf{0.21_{\pm 0.01}}$	$0.22_{\pm 0.01}$	$0.26_{\pm 0.01}$	$0.35_{\pm 0.01}$
	GTX 480	$\mathbf{0.19_{\pm 0.01}}$	$\mathbf{0.19_{\pm 0.01}}$	$0.20_{\pm 0.01}$	$0.24_{\pm 0.01}$
200–1,000	9800 GTX+	$\mathbf{5.06_{\pm 0.01}}$	$5.43_{\pm 0.01}$	$6.65_{\pm 0.01}$	$9.94_{\pm 0.01}$
	Tesla C1060	$0.96_{\pm 0.01}$	$\mathbf{0.90_{\pm 0.01}}$	$1.11_{\pm 0.01}$	$1.61_{\pm 0.01}$
	GTX 285	$0.73_{\pm 0.01}$	$\mathbf{0.71_{\pm 0.01}}$	$0.86_{\pm 0.01}$	$1.29_{\pm 0.01}$
	GTX 480	$0.60_{\pm 0.01}$	$\mathbf{0.54_{\pm 0.01}}$	$0.57_{\pm 0.01}$	$0.75_{\pm 0.01}$
200–10,000	9800 GTX+	$\mathbf{5.06_{\pm 0.01}}$	$5.43_{\pm 0.01}$	$6.65_{\pm 0.01}$	$9.93_{\pm 0.01}$
	Tesla C1060	$0.96_{\pm 0.01}$	$\mathbf{0.90_{\pm 0.01}}$	$1.11_{\pm 0.01}$	$1.61_{\pm 0.01}$
	GTX 285	$0.73_{\pm 0.01}$	$\mathbf{0.71_{\pm 0.01}}$	$0.86_{\pm 0.01}$	$1.29_{\pm 0.01}$
	GTX 480	$0.61_{\pm 0.01}$	$\mathbf{0.54_{\pm 0.01}}$	$0.57_{\pm 0.01}$	$0.75_{\pm 0.01}$
500–1,000	9800 GTX+	$\mathbf{76.92_{\pm 0.01}}$	$80.87_{\pm 0.01}$	$90.95_{\pm 0.02}$	$115.00_{\pm 0.01}$
	Tesla C1060	$8.46_{\pm 0.03}$	$\mathbf{7.71_{\pm 0.23}}$	$8.26_{\pm 0.02}$	$11.41_{\pm 0.04}$
	GTX 285	$6.36_{\pm 0.01}$	$\mathbf{5.56_{\pm 0.01}}$	$6.42_{\pm 0.01}$	$9.25_{\pm 0.04}$
	GTX 480	$5.61_{\pm 0.03}$	$4.23_{\pm 0.01}$	$\mathbf{4.05_{\pm 0.01}}$	$5.34_{\pm 0.01}$
500–10,000	9800 GTX+	$\mathbf{76.89_{\pm 0.01}}$	$80.83_{\pm 0.01}$	$90.97_{\pm 0.02}$	$115.05_{\pm 0.01}$
	Tesla C1060	$8.42_{\pm 0.04}$	$\mathbf{7.70_{\pm 0.22}}$	$8.25_{\pm 0.02}$	$11.41_{\pm 0.04}$
	GTX 285	$6.34_{\pm 0.01}$	$\mathbf{5.55_{\pm 0.01}}$	$6.41_{\pm 0.01}$	$9.24_{\pm 0.01}$
	GTX 480	$5.57_{\pm 0.01}$	$4.21_{\pm 0.01}$	$\mathbf{4.04_{\pm 0.01}}$	$5.34_{\pm 0.01}$
1,000–1,000	9800 GTX+	$\mathbf{673.56_{\pm 0.02}}$	$691.65_{\pm 0.02}$	$738.98_{\pm 0.03}$	$833.05_{\pm 0.02}$
	Tesla C1060	$58.10_{\pm 0.19}$	$\mathbf{50.63_{\pm 0.13}}$	$52.93_{\pm 0.13}$	$62.25_{\pm 0.09}$
	GTX 285	$43.97_{\pm 0.01}$	$\mathbf{36.19_{\pm 0.01}}$	$39.84_{\pm 0.03}$	$49.52_{\pm 0.01}$
	GTX 480	$40.69_{\pm 0.03}$	$28.43_{\pm 0.02}$	$\mathbf{25.25_{\pm 0.06}}$	$29.44_{\pm 0.03}$
1,000–10,000	9800 GTX+	$\mathbf{673.14_{\pm 0.02}}$	$691.80_{\pm 0.02}$	$739.28_{\pm 0.03}$	$832.94_{\pm 0.02}$
	Tesla C1060	$57.71_{\pm 0.15}$	$\mathbf{50.40_{\pm 0.13}}$	$52.56_{\pm 0.10}$	$62.18_{\pm 0.09}$
	GTX 285	$43.62_{\pm 0.01}$	$\mathbf{36.06_{\pm 0.03}}$	$39.47_{\pm 0.01}$	$49.26_{\pm 0.01}$
	GTX 480	$40.24_{\pm 0.02}$	$28.39_{\pm 0.06}$	$\mathbf{25.39_{\pm 0.01}}$	$29.90_{\pm 0.02}$

to 14 times in the runtime with respect to the 9800 GTX+. This reduction is mainly caused by the changes in coalesced access conditions of the devices and the increase in the number of CUDA cores.

The best performance of the algorithm when executed on the GeForce GTX 285 is obtained when using 32 threads per block for smallest instances and 64 threads per block for the rest of the instances. The results show a reduction in the execution time of up to 40 % regarding the Tesla C1060. This result is interesting since both cards have the same compute capability and the same number of cores and multiprocessors. The main difference between both cards is the theoretical peak of memory bandwidth that is 50 % greater in the GTX 285 than in the Tesla C1060 GPU (see Table 4). This fact is the main reason that explains the difference in execution times. It should be noted that the algorithm is divided into three kernels, one for the crossover and mutation, one for the calculation of the fitness values, and one for applying the elitism (in certain iterations there is an additional kernel for the exchange of directions of the solutions). Each of these kernels copy data from global memory to the multiprocessors, apply the operation, and copy the modified data back to the global memory, so the bandwidth of global memory is critical. Another reason that could explain the improvement in the performance is that the processors clock of the GTX 285 is 10 % faster than the processors clock of the Tesla C1060.

Finally, the best performance of the algorithm when executed on the GTX 480 is obtained when using 64 threads per block for smaller instances and 128 threads per block for the bigger instances. The results show an additional reduction in the execution time of up to 73 % regarding the GTX 285. The greatest reductions in runtime are achieved in executions with a large number of threads. This is provoked by the increase in the number of CUDA cores per multiprocessor as well as the inclusion of a double warp scheduler in each multiprocessor in devices with Fermi architecture (like the GeForce GTX 480 card). The results also show that the SGS implementation on a modern GPU can achieve significant performance improvements.

A further analysis on the comparative performance of the CPU and GPU implementations for the same configuration is elaborated next. Figures 11 and 12 show the improvement in performance (in terms of the wall-clock time reduction) of the GPU implementation versus the CPU implementation of SGS on the instances with 100, 200, 500, and 1000 items. The reductions range from 5.09 (100–1,000) to 35.71 (1,000–1,000) for the best setting for the number of threads per block on GTX 480. The tendency for more modern GPU models and for a given GPU card and a given number of threads is clear, the larger the instance, the higher the time reduction. The reason is twofold. On the one hand, larger tentative solutions allow SGS to better profit from the parallel computation of the threads, and on the other hand, the SGS model requires larger populations when the size of the instances increases (the grid has to be enlarged to meet the SGS structured search model), so a higher number of blocks have to be generated and the algorithm takes advantage of the capabilities offered by the GPU architecture. The experimental evaluation also allows to see the impressive evolution of GPUs in a period of only 2 years, since the runtime of SGS in the GeForce 9800 GTX+ is reduced up to 26× when executing in a GeForce GTX 480.

Although the improvements in the performance achieved are satisfactory, there is still room for larger improvements. Indeed, the GPU implementation can be further tuned to the Fermi architecture (GTX 480). If the implementation is tailored to the specific features of this architecture, the runtime of SGS can be reduced even more. In addition to this, most of the kernels used to deploy SGS on GPU

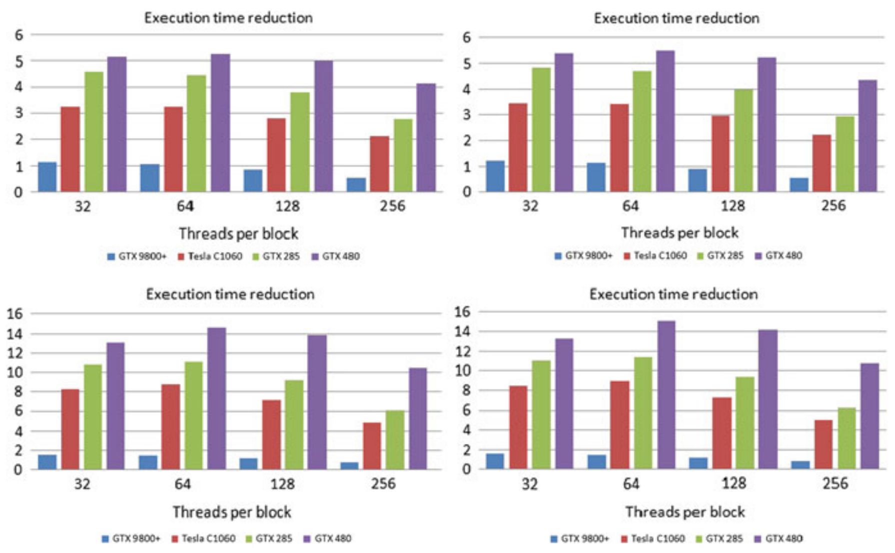


Fig. 11 Runtime reduction on GPU versus CPU for the smaller instances

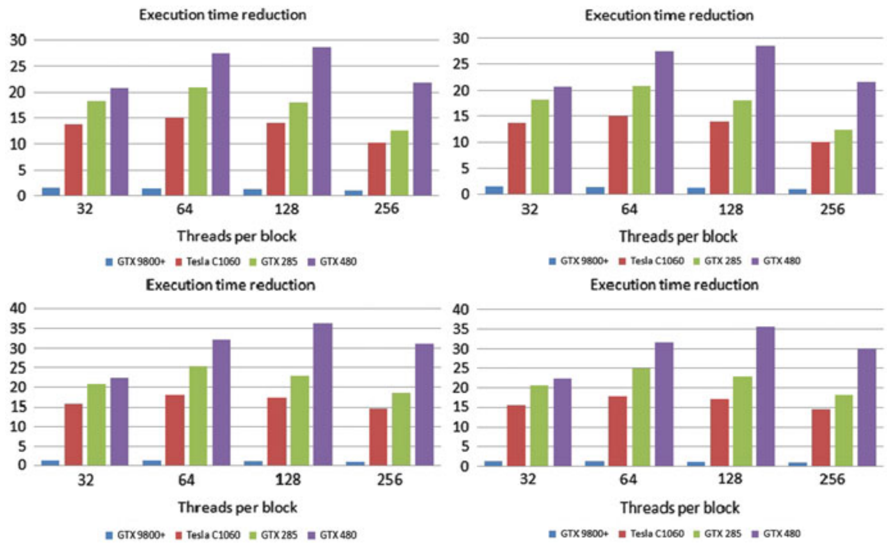


Fig. 12 Runtime reduction on GPU versus CPU for the larger instances

have a similar behavior, making it possible to merge all the operations into one, storing intermediate results in shared memory, and thus reducing accesses to global memory.

Finally, to study the performance of the GPU implementation of SGS versus the other algorithms implemented in CPU, we use the number of solutions built and

Table 8 Solutions (in millions) built and evaluated by the algorithms per second

Instance	SGA_{CPU}	EGA_{CPU}	SGS_{480}
100–1,000	0.690	0.631	**7.520**
100–10,000	0.642	0.602	**7.460**
200–1,000	0.368	0.335	**11.867**
200–10,000	0.356	0.340	**11.919**
500–1,000	0.152	0.143	**11.104**
500–10,000	0.151	0.146	**11.141**
1,000–1,000	0.077	0.072	**7.921**
1,000–10,000	0.077	0.073	**7.878**

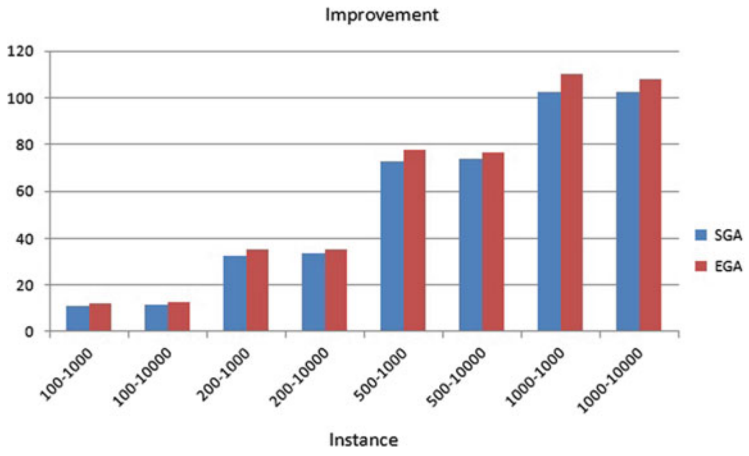


Fig. 13 Factor of improvement of SGS on GPU over SGA and EGA on CPU

evaluated by the algorithms. Table 8 presents the number of solutions (in millions) built and evaluated for each algorithm per second. The values reported were rounded to have three significant decimals.

Figure 13 shows the improvement factor when comparing the number of solutions built and evaluated by SGS on GPU against SGA and EGA on CPU. The results obtained show that the GPU implementation of SGS can build and evaluate solutions more than 100 times faster than the CPU implementation of SGA and EGA on CPU for the larger instances considered in this experimental evaluation.

5 Conclusions

In this work we have presented an in-depth study of a relatively new parallel optimization algorithm, the SGS algorithm. SGS is inspired by the systolic contraction of the heart that makes it possible that ventricles eject blood rhythmically according to the metabolic needs of the tissues, and previous ideas of systolic computing. This is just one of the first algorithms that tries to adapt new concepts (not the algorithms themselves) of existing metaheuristics to the very special architecture of a GPU.

The experimental evaluation conducted here showed the potential of the new SGS search model, outperforming three other algorithms for solving the KP in instances with up to 1000 items. The results have also shown that the GPU implementation of SGS speeds up the runtime up to 35 times when executing in a GTX 480 and automatically scales when solving instances of increasing size. The experimental evaluation also showed that the runtime of SGS executed in the oldest GPU available in our laboratory (the 9800 GTX+) can be reduced up to 26× when executed in the newest GPU available in our laboratory (the GTX 480). Finally, the comparative evaluation between the GPU implementation of SGS and the CPU implementation of SGA and EGA showed that SGS can build and evaluate solutions more than 100 times faster for the biggest instances considered in this work.

The lines of work currently explored are the study of the effect of merging the three kernels into one and providing wider impact analysis by solving additional problems to extend the existing evidence of the benefits of this line of research.

Acknowledgements Martín Pedemonte acknowledges support from Programa de Desarrollo de las Ciencias Básicas, Universidad de la República, and Agencia Nacional de Investigación e Innovación, Uruguay. Enrique Alba and Francisco Luna acknowledge support from the "Consejería de Innovación, Ciencia y Empresa," Junta de Andalucía under contract P07-TIC-03044, the DIRICOM project (http://diricom.lcc.uma.es), and the Spanish Ministry of Science and Innovation and FEDER under contracts TIN2008-06491-C04-01 (the M* project: http://mstar.lcc.uma.es) and TIN2011-28194 (the roadME project).

References

1. Alba, E. (ed.): Parallel Metaheuristics: A New Class of Algorithms. Wiley, London (2005)
2. Alba, E., Dorronsoroso, B. (eds.): Cellular Genetic Algorithms. Springer, New York (2008)
3. Alba, E., Tomassini, M.: Parallelism and evolutionary algorithms. IEEE Trans. Evol. Comput. **6**(5), 443–462 (2002)
4. Alba, E., Vidal, P.: Systolic optimization on GPU platforms. In: 13th International Conference on Computer Aided Systems Theory (EUROCAST 2011) (2011)
5. Benner, P., Ezzatti, P., Kressner, D., Quintana-Ortí, E.S., Remón, A.: A mixed-precision algorithm for the solution of Lyapunov equations on hybrid CPU-GPU platforms. Parallel Comput. **37**(8), 439–450 (2011)
6. Cecilia, J.M., García, J.M., Ujaldon, M., Nisbet, A., Amos, M.: Parallelization strategies for ant colony optimisation on GPUs. In: Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2011), Anchorage, pp. 339–346, 2011
7. de Veronese, L.P., Krohling, R.A.: Differential evolution algorithm on the GPU with C-CUDA. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2010), Barcelona, pp. 1–7, 2010
8. Ezzatti, P., Quintana-Ortí, E.S., Remón, A.: Using graphics processors to accelerate the computation of the matrix inverse. J. Supercomput. **58**(3), 429–437 (2011)
9. Harding, S., Banzhaf, W.: Implementing Cartesian genetic programming classifiers on graphics processing units using GPU.NET. In: Proceedings of the 13th Annual Conference Companion

Material on Genetic and Evolutionary Computation (GECCO 2011), Dublin, pp. 463–470, 2011
10. Johnson, K.T., Hurson, A.R., Shirazi, B.: General-purpose systolic arrays. Computer **26**(11), 20–31 (1993)
11. Kirk, D., Hwu, W.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, Los Altos (2010)
12. Kung, H.T.: Why systolic architectures? Computer **15**(1), 37–46 (1982)
13. Kung, H.T., Leiserson, C.E.: Systolic arrays (for VLSI). In: Proceedings of the Sparse Matrix, pp. 256–282 (1978)
14. Langdon, W.B.: Graphics processing units and genetic programming: an overview. Soft Comput. **15**(8), 1657–1669 (2011)
15. Langdon, W.B., Banzhaf, W.: A SIMD interpreter for genetic programming on GPU graphics cards. In: Proceedings of 11th European Conference on Genetic Programming (EuroGP 2008), Naples, 2008. Lecture Notes in Computer Science, vol. 4971, pp. 73–85. Springer, New York (2008)
16. Lewis, T.E., Magoulas, G.D.: Strategies to minimise the total run time of cyclic graph based genetic programming with GPUs. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2009), Montreal, pp. 1379–1386, 2009
17. Luque, G., Alba, E.: Parallel Genetic Algorithms: Theory and Real World Applications. Studies in Computational Intelligence, vol. 367. Springer, Berlin (2011)
18. Maitre, O., Krüger, F., Querry, S., Lachiche, N., Collet, P.: EASEA: specification and execution of evolutionary algorithms on GPGPU. Soft Comput. **16**(2), 261–279 (2012)
19. Maitre, O., Lachiche, N., Collet, P.: Fast evaluation of GP trees on GPGPU by optimizing hardware scheduling. In: Proceedings of the Genetic Programming, 13th European Conference (EuroGP 2010), Istanbul, 2010. Lecture Notes in Computer Science, vol. 6021, pp. 301–312. Springer, New York (2010)
20. NVIDIA: NVIDIA CUDA C Programming Guide Version 4.0 (2011)
21. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Comput. Graph. Forum **26**(1), 80–113 (2007)
22. Pedemonte, M., Alba, E., Luna, F.: Bitwise operations for GPU implementation of genetic algorithms. In: Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '11), pp. 439–446. ACM, New York (2011)
23. Pedemonte, M., Alba, E., Luna, F.: Towards the design of systolic genetic search. In: IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, pp. 1778–1786. IEEE Computer Society, Silver Spring (2012)
24. Pisinger, D.: A minimal algorithm for the 0–1 knapsack problem. Oper. Res. **45**, 758–767 (1997)
25. Pisinger, D.: Core problems in knapsack algorithms. Oper. Res. **47**, 570–575 (1999)
26. Pisinger, D.: Where are the hard knapsack problems? Comput. Oper. Res. **32**, 2271–2282 (2005)
27. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional, Reading (2010)
28. Soca, N., Blengio, J., Pedemonte, M., Ezzatti, P.: PUGACE, a cellular evolutionary algorithm framework on GPUs. In: Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010, pp. 1–8. IEEE, Brisbane (2010)
29. Vidal, P., Alba, E.: Cellular genetic algorithm on graphic processing units. In: Nature Inspired Cooperative Strategies for Optimization (NICSO 2010), pp. 223–232 (2010)
30. Zhang, S., He, Z.: Implementation of parallel genetic algorithm based on CUDA. In: ISICA 2009. Lecture Notes in Computer Science, vol. 5821, pp. 24–30. Springer, Berlin (2009)
31. Zhou, Y., Tan, Y.: GPU-based parallel particle swarm optimization. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2009), Trondheim, pp. 1493–1500, 2009

Genetic Programming on GPGPU Cards Using EASEA

Ogier Maitre

Abstract Genetic programming is one of the most powerful evolutionary paradigms because it allows us to optimize not only the parameter space but also the structure of a solution. The search space explored by genetic programming is therefore huge and necessitates a very large computing power which is exactly what GPGPUs can provide. This chapter will show how Koza-like tree-based genetic programming can be efficiently ported onto GPGPU processors.

1 Introduction

Evolutionary algorithms are known to be able to give nonoptimal but satisfactory solutions to difficult optimization problems. Most variants of EA use a fixed representation (a genome) and modify all elements (genes) to obtain a globally satisfactory solution.

Genetic programming (GP) aims at optimizing both the structure and the elements of an individual at the same time. As such, GP is generally considered to be a very time-consuming optimizing algorithm, because of the large search space implied by the large number of parameters and possible structures. The individual is said to be a program, which is evaluated by being executed with several input data. These data are points of a learning set that the model created by the GP attempts to match as closely as possible. As the learning set is usually large, the execution time of an individual in GP is generally considered as very time consuming. This time grows linearly with the number of points in the learning set.

O. Maitre (✉)
LSIIT, University of Strasbourg, Pôle API, Bd Sébastien Brant, BP 10413, 67412 Illkirch, France
e-mail: ogier.maitre@unistra.fr

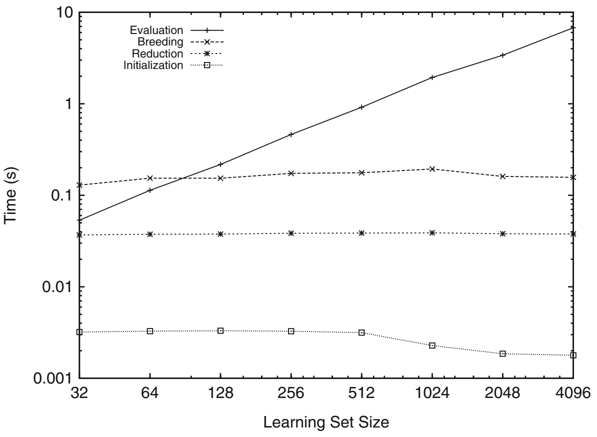


Fig. 1 Time elapsed into different steps of a GP algorithm, *w.r.t.* the learning set size

Figure 1 shows an example of time repartition for a GP algorithm, on only ten generations, with respect to the number of points in the learning set. This example is a symbolic regression problem, and it will be used as an example throughout this chapter.

Generally, GP evaluation is done by comparing the results produced by an individual, on all the points of the learning set. Conversely to GA/ES, the individual needs to be executed several times, in order to be evaluated, and evaluation of each individual produces the execution of different instructions, depending on the individual structure and parameters. This last point explains the analogy between the individual and a computer program.

As the overall GP algorithm works grossly as GA/ES, the same kind of parallelization is possible. Particularly, a master–slave model implementation is possible in order to parallelize the evaluation of the population, each core evaluating a different individual and one core being responsible for executing the rest of the evolutionary algorithm. However, the execution path of the evaluation function being highly dependent on the individual structure and content, this behavior is not adapted to the SIMD architecture, as implemented into GPGPU. Indeed, executing two trees on two cores of the same SIMD bundle is likely to produce divergences between the cores, which will slow down the overall execution time. A divergence occurs when two SIMD cores attempt to execute different instructions at the same time. As the architecture is not able to do that, one core executes its own instruction, when the second remains idle, and then the second executes its instructions and the first remains idle. This behavior can be extended to more than two cores, using the same principle.

Compared to a generic GA/ES, the generic GP symbolic regression algorithm has an additional feature. Indeed, the evaluation of an individual induces the execution of its tree on the whole learning set, as seen above. This translates into the population evaluation function having two nested loops: the first iterating on the member of the population (as for GA/ES) and the second iterating on all the learning cases. Each loop has independent iterations and therefore can be parallelized.

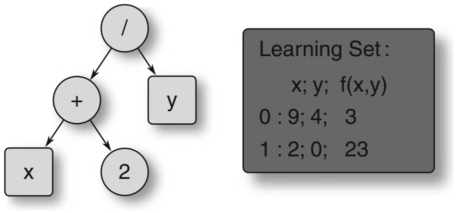


Fig. 2 An example of a GP-tree individual and a problem-related learning set

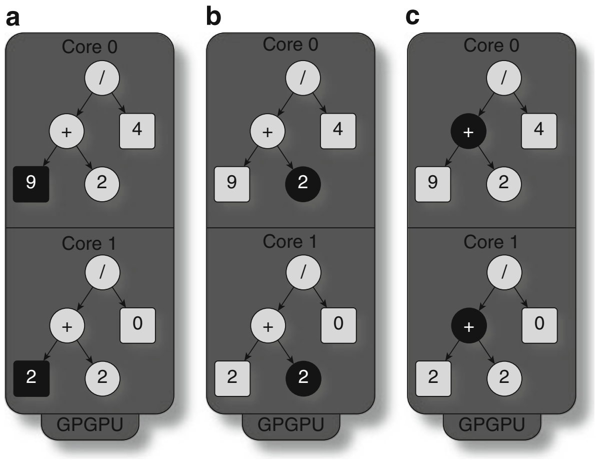


Fig. 3 Execution of the first learning set cases in SIMD mode. (**a**) Push x values; (**b**) push constant 2; (**c**) pop x and 2, apply addition

This means that there are basically two ways of parallelizing the evaluation of the population of a generic regression algorithm. These two methods are described in the next section.

1.1 Option 1: SIMD Parallelization of a Single Individual Evaluation over the Learning Set

The first approach uses the GPGPU processor as a completely SIMD machine. Thus, one individual is loaded on the card, which executes (or interprets) it on one or several learning points per core.

Figure 2 presents such a minimal example of a GP population and a learning set of two two-dimensional points, on which the individual needs to be evaluated. By using this technique, the evaluation implies the sequence of operations described in Fig. 3. At each time, the two cores execute the same instruction, possibly on different

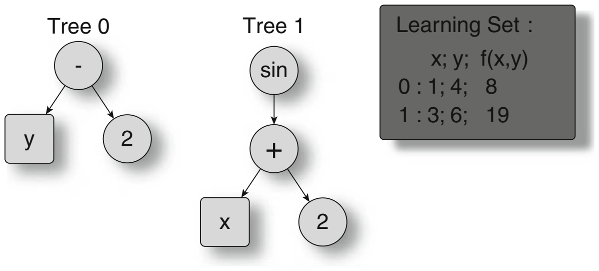


Fig. 4 An example of a GP-tree population and a problem-related learning set

data when the currently evaluated subtree contains a learning variable (here, x or y). The maximum number of cores that can be used here is two and corresponds to the number of points in the learning set.

Using all the resources available on a GPU processor is therefore conditioned by the number of points of the learning set. These resources include principally the number of threads that can be executed at a time (the number of cores times the number of thread that each can schedule). This number can be large and is growing with newer cards. For instance with Kepler architecture, the number of threads that can be executed at the same time by one card is $\approx$50,000. Obviously, using a multi-card system will increase the size of the learning set needed to saturate the card. Finally, this parallelization method does not take into account the MIMD structure of the NVIDIA GPGPUs.

1.2 Option 2: MIMD Parallelization over the Whole Population

A second approach is presented in the literature, which is the one classically used for GA/ES. In this second approach, each core executes the evaluation of a different individual and the number of parallel tasks that are executable here is N, where N is the population size to be evaluated (either λ or μ; Fig. 4).

This parallelization is ill suited for SIMD processors. However, Fig. 5 shows how the two individuals of Fig. 4 would be executed on an SIMD processor:

1. The execution of the two first nodes can be done in SIMD, as they are identical. However, as in this example the loaded variables (x and y) are not the same; the memory accesses cannot be coalesced.
2. The second node also induces the execution of an SIMD instruction, with a coalesced memory access here (“read 2”).
3. As for this 3rd step, the instructions are different ($+$ and $-$) and the SIMD cores will execute them in a serial way (one after another).
4. As the first tree evaluation is finished, one core stays idle during the execution of the other tree on the second core.

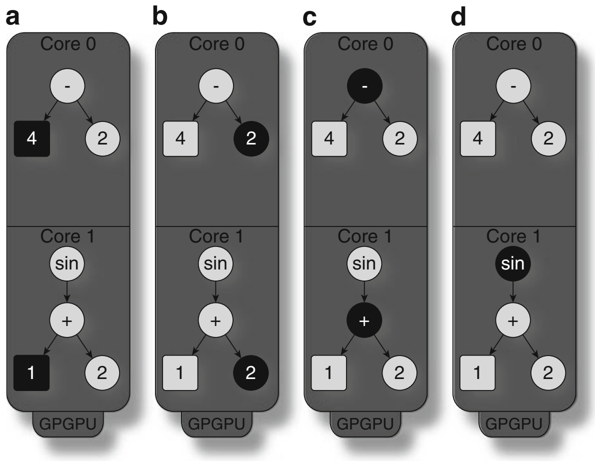


Fig. 5 Execution of the first three nodes of the example tree in an SIMD manner. (**a**) Single instruction, non-coalescent memory access. (**b**) Single instruction, coalescent memory access. (**c**) Divergence, two instructions. (**d**) One idling core

MIMD parallelization on SIMD hardware exhibits some disadvantages. The first major one is the divergences that can occur between different cores of an SIMD bundle of cores. This problem comes from the non-predictability of the individual structures, which are likely to be different. Conversely to a simple "if-else" divergence, in the GP case, the number of paths that take the cores will be at maximum the number of cores into a bundle or the number of different operators into the function set. These numbers could be large enough to penalize the execution, as on a NVIDIA GPU card the number of cores into an SIMD bundle is up to 32 and the number of functions in the set can easily be up to ten functions.

The example shows other problems, as the differences in tree length, and the influence that this factor has on the load balancing of the tasks. A scheduling mechanism needs to be used here in order to balance the workload of each core in order to reduce the overall execution time. Finally, the memory access patterns induced by the trees are highly unpredictable, as they depend on the variable present into a final node. They are unlikely to be coalescent accesses, and there is probably a waste of memory bandwidth here.

2 Related Work

As GP has a very large search space, inducing a prohibitive execution time, a number of scientists were interested to use GPGPU to evaluate GP individuals. The first known implementation of GP on GPU is [1], where a compiled approach is used. The parallelization method selected by the author is option 1 presented in the previous section.

One individual is thus compiled and sent to the card, in order to be evaluated on the whole learning set, which is preliminary converted as a 2D texture. The compiler used in this publication is a Cg compiler (C for Graphics), and the GP algorithm translates the current individual to be evaluated in a Cg program. All the GPU cores execute the same node at the same time, on different data, and finally the error is summed and returned as the individual fitness, on the CPU side. This compiled approach should have a large overhead, related to translation and compilation.

However, the author obtains a speedup of the order of 30× on a NVIDIA 6400GO card, when it is compared to an Intel 1.7 GHz. This speedup is obtained on an 11-way multiplexer, with 2,048 learning points, but it drops at 10× on symbolic regression problem. As the overhead is quasi-constant and principally induced by the Cg compiler, Chitty asserts that enough learning cases should be used on GPU. It is not clear whether the compilation time is taken into account here when timing the individual evaluation, as well as the method used as reference on CPU (compiled or interpreted).

In the same year, Harding et al. apply a similar technique in [2], using *.Net* and *MS Accelerator*, on four different problems. The authors generate a random population of individuals of a fixed size and evaluate them on a GPU and a CPU. Then, they test the average speedup of the GPU implementation vs. the CPU on 100 different individuals, using a 7300GO card vs. an Intel T2400 1.83 GHz CPU. Using this mechanism, they investigate the obtained speedup *w.r.t.* different individual size and training set sizes. They get very interesting speedups, in the order of 7,000×, on the evaluation of a floating expression population, 800× for Boolean expressions. For the complete algorithm, they obtain a maximum speedup of about 100× on a linear regression problem (which uses floating expressions) and 22× for a problem of spiral classification (which uses Boolean expressions). However, the use of the *.Net* and *MS Accelerator* frameworks, the individual optimization during the compilation and the lazy evaluation mechanism make it difficult to compare these results. Nevertheless, the use of *MS Accelerator*, which works on top of DirectX, allows the program to be ported on different types of devices (GPGPU, multi-core processor) but stays limited to Microsoft operating systems. In addition, this implementation uses the GPGPU as a complete SIMD processor, which implies using a large number of training cases, as was stated in previous section for this kind of option 1 implementation.

In [7] paper, Langdon et al. use option 2, as presented above (MIMD execution). Indeed, their interpreter, coded using RapidMind, dispatches a different individual on each core of the card. To cope with RapidMind, which uses the GPU as an SIMD machine, their interpreter implements explicitly the execution of all the functions from the learning function set and keeps only the desirable result, linked to the current node operator. They use a 8800GTX card, against an AMD Athlon 64 3500+ and obtain a speedup of $\approx$12×. Obviously, this implementation exhibits an overhead, because of the MIMD characteristic of the interpreter and the authors evaluate the loss of performance to be around 3×, which is in the same order as the size of the learning function set.

In 2008 and 2009, Robilliard et al. implement an interpreter well fitted to NVIDIA cards, in [11, 12]. This implementation is a mixed approach between options 1 and 2 that allows us to evaluate at the same time several individuals on several learning cases. The interpreter allows us to map an individual to an MP (a bundle of SIMD cores) and to let each SP (a core) compute another learning case. This implementation is closed to the real hardware of NVIDIA cards, as SP are SIMD inside an MP, and the MP are MIMD between them. The authors obtain a speedup of the order of 80× on an 11-way multiplexer and on a symbolic regression problem, using 2,500 learning cases, on a population evaluation. The GPU which is used in these experiments is a 8800GTX and the CPU is an Intel 2.6 GHz.

Similarly, in 2010 Langdon implements a GP algorithm using cuda and NVIDIA hardware in order to solve 20-mux and a 37-mux problem. The implementation is similar to the previous one, except Langdon uses the 32 bits of a register as a 32-boolean evaluating machine. It is therefore possible to evaluate 32 learning cases with one thread on one individual. He obtains a very interesting speedup with a 2,048 learning set for 20 mux and 8,192 for 37 mux, using a 1/4 of a million population. The learning set is randomly picked out of 1,048,576 (resp. 8,192 of 137,438,953,472) for 20-mux (resp. 37-mux). The principle remains the same as the population is represented using RPN and using a mixed approach between options 1 and 2. Finally, he obtains a speedup of 10,000 on the evaluation function, using a Tesla 121 card compared to a CPU RPN implementation. As the implementation presented by Robilliard et al. is well fitted for NVIDIA hardware, it will be the base of the following implementation.

3 Implementation

3.1 Algorithm

Our GP algorithm uses GPGPU processors to execute the population evaluation, as a master–slave model. The rest of the algorithm still works on the CPU, sequentially. The same kind of flowchart is used in all the papers presented above.

Because of this master–slave model, between the two systems (CPU and GPU), the individuals have to be transferred to the GPU memory before evaluation. This and the tree structure of the individuals impose a translation of the pointers that compose the internal nodes of the population. In order to compact the population and to avoid pointer calculation on the GPU side, the population is translated into RPN. As the memory transfer latency is high, the whole population is translated into a single buffer, which is copied to the GPU memory in one go before evaluation.

The population is translated, on the CPU side, into a RPN buffer that will be transferred before evaluation, in one go, to the GPU memory. This kind of representation is also used by Robilliard et al. However, their first implementation handles tree-based representations on the CPU side and translates the population before evaluation in [11], and their second version uses native RPN on both sides. As the evaluation phase remains the same, the RPN representation being the same in both cases, we prefer to use standard representation on CPU side followed by a translation phase.

After the evaluation, the fitness values are gathered back to the CPU representation of the population, in order to be used correctly by the rest of the evolutionary algorithm. Finally, Robilliard et al. implement their interpreter into ECJ library, using JNI in order to call native code from the Java CPU algorithm. Our implementation uses the EASEA internal library, directly coded into C++.

3.2 Evaluation

Option 1, as seen in Sect. 2, needs a large number of learning cases in order to keep all the cores busy. This is even more true, when considering the number of threads that should run in parallel on modern GPU architectures.

Conversely, option 2 uses the number of individuals to keep the cores busy. These parameters can be adapted; as usual GP population sizes are large and the chip occupancy can be easily maintained to a high value. The major drawback of this approach is its lack of adaptation to SIMD architectures.

Finally, the implementation of Robilliard et al. allows us to combine some advantages of both approaches, in one interpreter, that is adapted to modern GPGPUs. Each core executes one interpreter which takes an individual as input, represented in RPN. The interpreter uses a stack to record temporary results from the subtrees already evaluated (in a bottom-up way), and this stack is stored into the shared memory of the MP. The same individual is assigned to all the threads running on the same MP, which hence an SIMD execution, as all the cores will evaluate the same node at the same time. At the end of an individual evaluation, one core sums up all the errors of each learning case, in order to compute the final fitness. This is a limit in parallelism for evaluation, even if it can be noted that a parallel reduction (using divide and conquer reduction) can help maintain a parallel activity as long as possible. As this last step represent only a small portion of computation, it makes sense to implement it as simple as possible.

However, due to shared memory size limitation, only few interpreters can be run in parallel, on the same MP. Robilliard et al. use only 32 learning points on one MP, because it is the minimum number of threads that can be run in parallel (the warp size). This also has the advantage of implicitly synchronizing all the threads evaluating an individual and this synchronization is needed, at least before to sum up the global error. Furthermore, if hardware and software constraints allow it, four blocks will be automatically loaded onto one MP, which hence 128 threads running in parallel. The major software constraints are the number of registers used by an interpreter and the size of a stack.

As the hardware will load up to four individuals, this suggests that it is possible to evaluate several individuals on the same MP. Divergences will only occur between the threads of a warp, as it is described into NVIDIA documentation. Indeed, only these threads are executed together, in an SIMD way.

It is possible to logically load several individuals onto the same MP, in order to increase manually the number of threads executed on an MP. This allows us to have a better memory latency overlapping, by context switching. It is necessary to move stacks from shared memory to global memory, in order to load more interpreters onto one MP. This allows us to increase the number of threads loaded from 128 to 512 and could allow us to reach the limit of the newest card, i.e., 1,024.

Our implementation keeps the SIMD/MIMD option from the Robilliard et al. papers but implements a multi-individual interpreter using global memory for storing stacks. This allows us to decrease the average memory access time, without creating any divergence, as long as an individual is evaluated on a multiple of 32 learning cases in parallel. Using global memory also allows us to scale our implementation to the newest cards, which have larger number of cores on MP. Indeed, 680GTX has now 192 cores per MP, against 8 for the 8800GTX and 32 on GTX480, but the shared memory size has not been increased accordingly.

Another solution to scale for these cards is to increase the number of learning cases computed in parallel, up to fully load scheduling pool. Depending on the card, this number of fitness cases should be between 192 and 256. Considering that the hardware will use four individuals per MP, this will give 768 or 1,024 threads loaded per MP.

Finally, it is possible to use several cards with this model, by distributing a part of the population to evaluate each card. The population needs to be adapted to this new number of executable threads, but no other disadvantages have been observed here.

4 Experiments

4.1 Experimental Process

The experiments have been performed on a GTX295 NVidia card vs. a Intel Quad core Q8200 (2.33 GHz) processor and 4 GB RAM, under GNU/Linux 2.6.27 64 bits with NVidia driver 190.18 and CUDA 2.3. The GTX295 is a dual GPU card, but to simplify things, first we only use one GPU to evaluate GP individuals, the second card being used for the standard display.

The CPU code has been compiled with gcc 4.3.2 using the -O2 optimization option and run on one core of the CPU only, so as to compare the parallel GPGPU algorithm to a sequential one on CPU. By not implementing a multi-core CPU GP, one can still imagine that in the best case scenario, a perfect multi-core CPU implementation would show a linear speedup, with respect to the number of cores of the CPU.

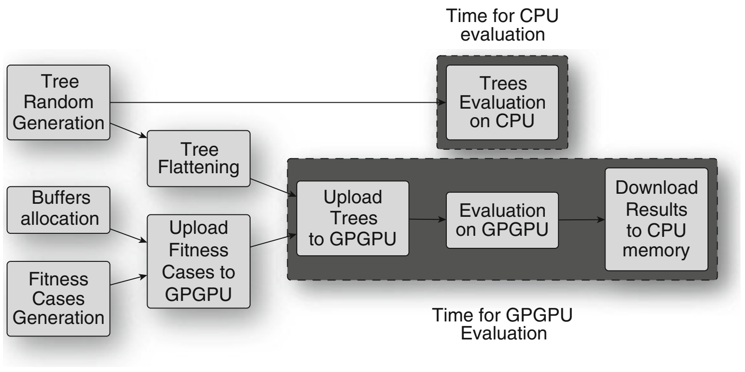


Fig. 6 Global scheme for GP simulation and timing process

4.2 Evaluation Function

These experiments have been designed to evaluate the behavior of the interpreter, that is, only the evaluation step is timed here. We compare a mono-core implementation handling tree individual for the CPU reference to the GPGPU interpreter described above.

Four implementations are first compared, being, BlockGP32, an internal implementation of Robilliard et al. interpreter. BlockGP128 uses global memory to store the stacks and evaluate 128 learning cases in parallel, in order to increase the number of loaded threads. Finally, the two last implementations (BMGP2, resp. BMGP4) use two (resp. four) individuals per MP, to increase this number, without increasing the number of learning cases needed for the algorithm.

Figure 6 describes the methodology used during these tests, where only the execution of the population evaluation is timed. On the CPU side, evaluation only includes the interpreter execution, whereas on the GPU side, evaluation includes population RPN translation of the trees, and copies between the two memory spaces. The time to transfer learning cases is not included, as in a real GP algorithm, this stage is only executed once, at the beginning of the algorithm. The impact of this transfer is thus less important on the execution time of a complete algorithm.

4.2.1 Evaluation Time vs. Number of Fitness Cases

Figure 7 presents the influence of the learning set size on the evaluation time, for different implementations of the interpreter. The population used here contains 60,000 depth 7 individuals, initialized using *full* method described in [3].

The BlockGP32 implementation achieves the worst performance, in all cases. BMGP2 and BMGP4 are more efficient for small-sized learning case sets (less than 128 learning cases). Finally, BlockGP128 is the most efficient implementation for larger sets. The curves are stair shaped, with a width being equal to the number of

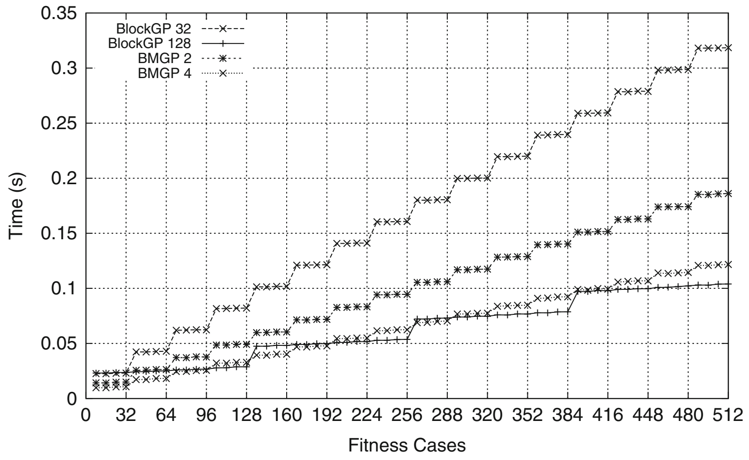


Fig. 7 Evaluation time for depth 7 trees for different implementations with a sampling size of eight fitness cases

learning cases evaluated in parallel. This can be attributed to the number of thread executed in parallel by the hardware.

Finally, scheduling helps to speed up the execution of the GP interpreter. Even if this imposes to store the stacks into the global memory, degrading the access time to the temporary results, it improves all other memory accesses and divides the execution time.

This section was dedicated to the comparison of several evaluation functions running onto GPU. It is important to note that the execution time varies largely with several parameters as we will see in the next sections. Indeed, GP as the majority of the EC algorithms allow us to set a large variety of parameters that could here influence the evaluation speed of the interpreter. Being able to know what is the influence of each parameter on this overall measurement could help to set these parameters to an efficient value, at least from the hardware point of view. Finally, it could provide information on the interpreter behaviour, given a set of parameters.

4.2.2 Influence of Tree Depth on Speedup

We use the same timing technique as in the previous section, in order to define the influence of the tree depth. The evaluation algorithm used here is BMGP4, and the population size is equal to 4,096. The functions used to compose the trees are part of $\{+, -, \oplus, \times, \cos, \sin\}$ (where $\oplus$ is the protected division). As this function set contains operators with different arity (1 and 2), the trees of the population are not guaranteed to have the same number of nodes.

The speedup is computed, between our half GTX295 and an Intel Q8400 quad core processor. Finally Fig. 8 shows that except for extremely low value, the tree

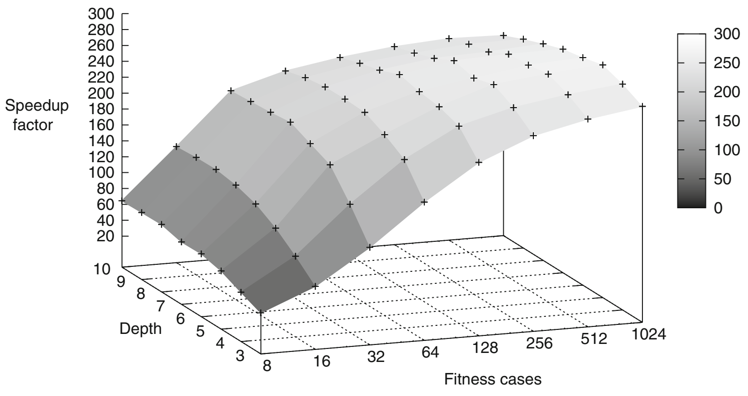


Fig. 8 Resulting speedup between GPGPU and CPU

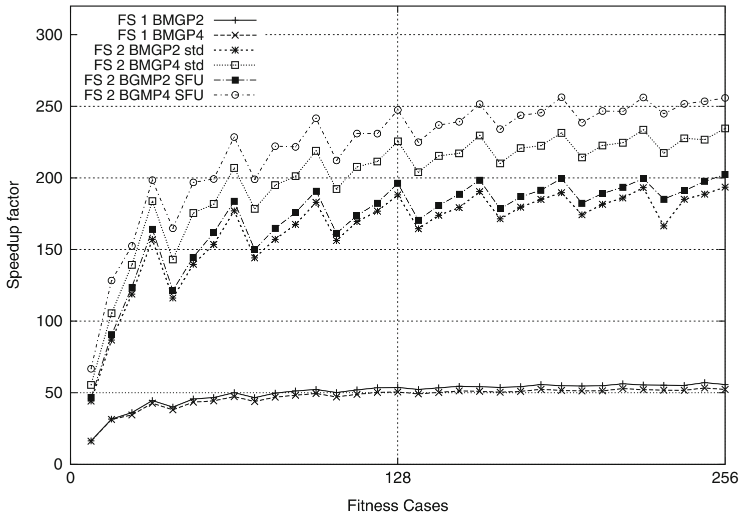


Fig. 9 Resulting speedup for depth 7 trees on GPGPU against CPU for different function sets

size has no real influence on the resulting speedup, but the number of learning cases used for evaluation has largely more. One can note that the speedups reach a plateau after a threshold of 32 fitness cases, having a speedup close to 250×. This shows that our implementation is already efficient at this stage.

4.2.3 Influence of the Function Set

The computation intensity of a learning function set highly depends on the operators in it. This intensity allows us to justify the transfer of a population to the GPU

Table 1 Two different learning function sets

F1	F2
$+, -, \oslash, \times$	$+, -, \oslash, \times$, cos, sin, log, exp

memory. Knowing this, we try several sets having different computational intensities (Table 1).

Function Set 1 is a relatively lightweight function set, in matter of computational intensity. This is obviously due to its content, which is composed only by "easy-to-compute" operators. Their occupancy does not seem to influence the performances, as BMGP2 and BMGP4 obtain approximately the same results.

Function Set 2 has a higher cost in terms of computational intensity. Speedups reach the values of the previous experiments. The teeth-shaped curves match the stair-shaped ones, because CPU implementation has a linear computing time, *w.r.t.* the number of fitness cases. When the CPU time values are divided by the stair-shaped ones, of the GPU implementations, the resulting speedup curves become indeed teeth shaped.

As seen previously, a tooth appears every 32 fitness cases, when all the cores inside an *MP* are fully occupied. Depending on the number of individuals assigned to each *MP*, the speedups range between 180× and 230× (Fig. 9).

The last function set (*FS2*+*FSU*) uses fast approximation for heavy functions as cos, sin, log, and exp. These approximations are used, for instance, for 3D rendering, which explains why they are available here. Speedups are higher, when using this approximation on the GPU side, but the precision is degraded. However, using these units is interesting for GP on GPU, if the target architecture for the final application of the GP individual has the same kind of functions or finally if the accuracy is not a fundamental point.

4.2.4 Influence of Tree Shapes on Speedup

Previous experiments use population initialized by Koza's *full* method. By using such a method and an equal arity function set (such as $\{+, -, \times\}$), the generated trees have the same number of nodes. Even when using function set such as *Function Set 2*, the trees should have in average the same number of nodes. This method introduces an optimistic bias in the case of a multi-individual evaluation as ours. Indeed, with homogeneous populations, load balancing is simplified.

The *grow* construction method uses two sets of operators, but conversely to the full method, both terminal and nonterminal sets are merged into one, where the construction algorithm picks the node for the next child node to be constructed. Finally, if the maximum size limit is reached, the picking is done into the terminal set only, in order to stop the growth of all branches. In [3], the authors discard trees that do not reach a minimum size limit, as for instance choosing a terminal node first, will conduct to generate a single node tree.

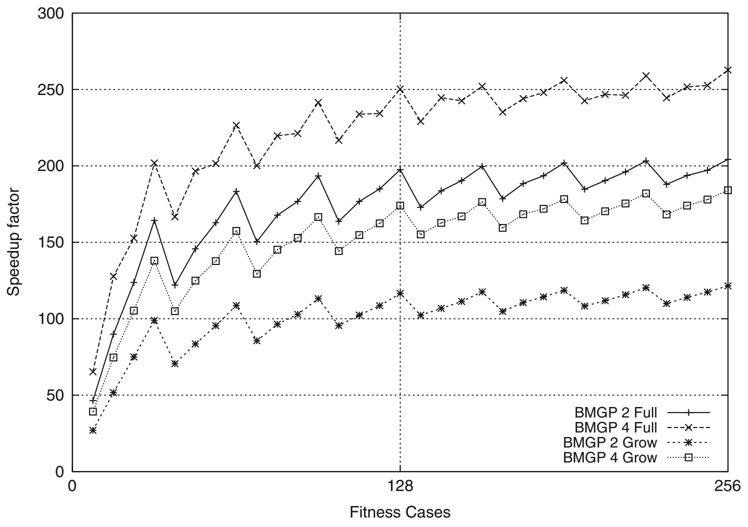


Fig. 10 Resulting speedup for depth 7 trees on GPGPU against CPU for different tree construction methods

This last method generates a population composed with mixed length individuals. It is important to note that during evolution, as a solution is taking place, the population will tend to get homogeneous, even if there is no evidence that trees will become full.

Figure 10 shows that heterogeneous populations tend to degrade performance of multi-individual populations, in absence of real load-balancing algorithm. Indeed, as a short individual can be packed with a longer individual, the set of cores to which the individuals are assigned tends to be less busy when the shortest individual is finished. As the CPU algorithm does not suffer from this kind of consideration, the speedup tends to degrade in this situation. When four individuals are packed per core set, the situation tends to be more averaged, and speedup is higher. With a single individual evaluation algorithm and using a NVIDIA card, as an individual is assigned to a logical block of threads, as soon as an individual is completely evaluated, it is replaced by the scheduler by another one, which is ready to be evaluated. The scheduler acts as a load-balancing helper in this case. Using several individuals into one block works obviously better when all the individuals evaluated on the same bundle of cores have the same evaluation time.

This situation is interesting to evaluate, as real problems use Koza's ramped half and half methods to initialize the parent population. The ramped half and half method uses both *grow* and *full* sub-method to construct each half of the population and generates a heterogeneous population by increasing the tree size during the construction phase between a lower and an upper bound size.

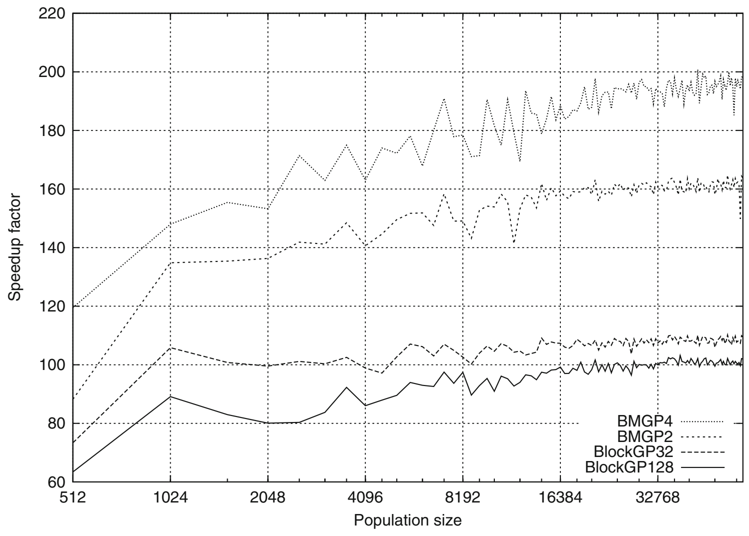


Fig. 11 Resulting speedup for depth 7 trees, function set FS2SFU and 32 fitness cases on GPGPU against CPU *w.r.t.* the population size for different implementations of GP evaluations

4.2.5 Influence of Population Size

As we have seen, the performance of the evaluation method executed on GPGPU, when compared to a CPU implementation, highly depends on the load of the card processor. Using mixed MIMD/SIMD approach imposes the use of a certain number of individuals in order to assign enough individuals to each MP. This assertion is even more true using a multi-individuals method, as it is possible to see in Fig. 11. Indeed, even if passed a certain threshold number in population size, this last factor seems to be non-determinant in a matter of speedup and still has an influence with small-sized populations. Fortunately, usual GP algorithms use a large population which will be largely sufficient to saturate all the cores of the card. As GPGPU hardwares are evolving quickly nowadays, towards always more core into one chip, it is still important to assure that parameter.

4.3 Experiments on the Complete Algorithm

The previous section was dedicated to the analysis of the evaluation function only, as a primary time-consuming part of a GP algorithm. Furthermore, it is the only part that differs, between the CPU and the GPGPU implementation, when one is using the master–slave model, as it is largely the case for GP on GPGPU. Finally, from the final user point of view, the important part should be the final speedup obtained on the overall algorithm and the next sections are dedicated to the whole algorithm.

4.3.1 Theoretical Speedups

A large majority of GP parallelization has been done using the master–slave model. As we have seen before, it is because of the large unbalance that exists between the evaluation and the other part of the algorithm in terms of execution time. Amdahl's law asserts an obvious fact: it is possible to gain execution time on the parallel part only, that is, speedup is bounded by the serial part of the algorithm.

In order to evaluate the maximum obtainable speedup using our GP implementation, we imagine a perfect parallelization running on an infinite number of processors. This gives us an evaluation function that is executed instantaneously. In our experiment, we use fixed fitness values that let the GP algorithm work without the evaluation step. This way, the evolution will take exactly the same direction with and without evaluation, as fitness values depend in our case on the individual population index.

The surface of Fig. 12 presents the results obtained by this implementation. It is possible to see that with a large number of fitness cases, the theoretical speedups are very large. Furthermore, the population size seems to have no influence on the theoretical speedup, as an increase in population size also includes larger execution time into the sequential part of the GP algorithm (crossover, mutation). From this last point of view, an increase in the learning set size is free of sequential part execution time increase.

A remarque can be made about Fig. 12 which is that the numerical results are highly dependent on the GP implementation and cannot be extended to other works without caution.

Figure 13 presents the speedup obtained with the complete algorithm, with the real multi-individual implementation. Obviously the obtained speedup (134) is smaller than the theoretical on, but large speedup values are still obtained. The same trend is observable about the small influence of the population size compared to the one of the learning set size.

4.4 Example on a Real-World Problem

This real-world problem consists of regressing the linear state-space model of an airplane from telemetric data recorded during a flight.

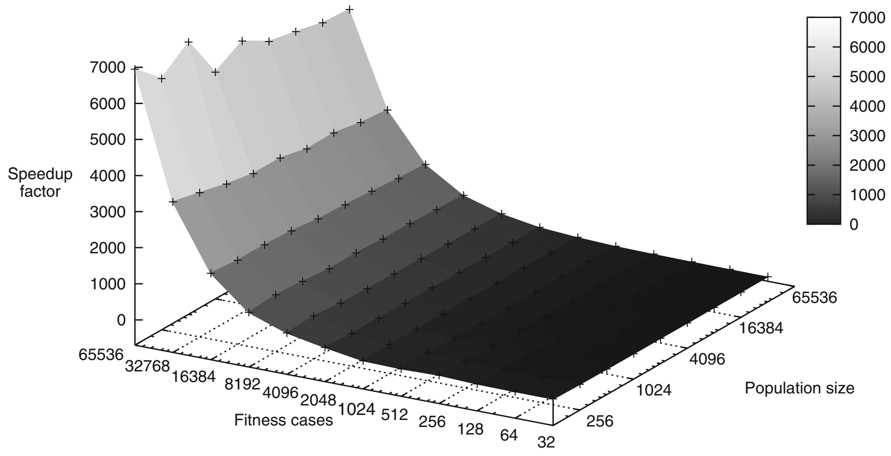


Fig. 12 Maximal speedup with the tree-based GP implementation on the cos $2x$ problem

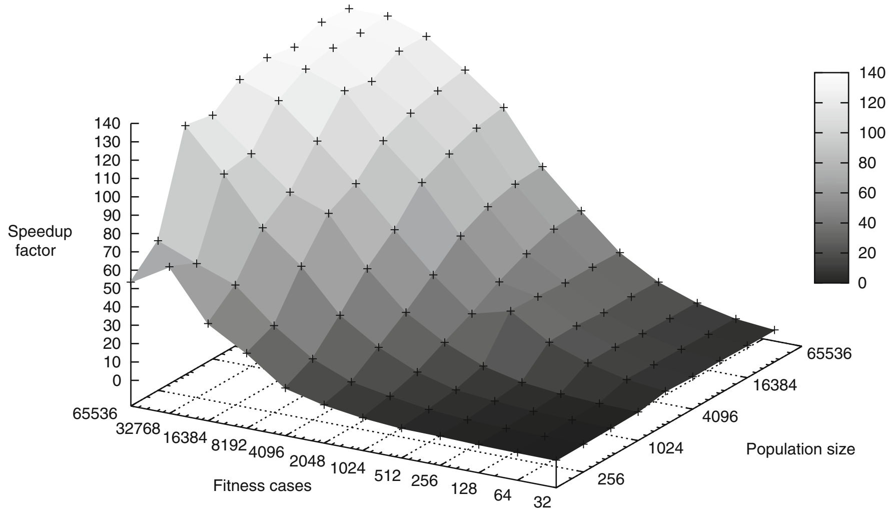


Fig. 13 Speedup with respect to the population size and the number of fitness cases

4.4.1 Principle of an Aircraft Model

Any real-world and dynamic system can be modeled through nonlinear state-space representation, as mathematically shown in [8]. This nonlinear state-space representation consists of mathematical model of a physical system, defined by a set of inputs, outputs, and state variables, related by first-order differential equations, as presented hereafter :

$$\begin{cases} \dot{x}(t) = f(t, x(t), u(t)) \\ \dot{y}(t) = h(t, x(t), u(t)) \end{cases}$$

where $x(t)$ is the state vector, $u(t)$ the control vector, and $y(t)$ the output vector. The state vector is not measurable, it is usually estimated using a Kalman Filter applied to $y(t)$. The first equation is the "state equation" and the second one is the "output equation."

The internal state variables (the state vector) are the smallest possible subset of system variables that can represent the entire state of the system at any given time. The minimum number of state variables required to represent a given system, n, is usually equal to the order of the defining differential equation of the system.

The control variables (the control vector) are the subset of variables which are used to drive the system. If we consider the following state vector X: $x^T = [x_1\ x_2\ x_3\ \ldots\ x_n]$ and the following control vector U: $u^T = [u_1\ u_2\ u_3\ \ldots\ u_m]$, then the nonlinear state-space representation is:

$$\begin{cases} \dot{x}_1(t) = f_1(t, x_1(t), \ldots, x_n(t), u_1(t), \ldots, u_m(t)) \\ \dot{x}_2(t) = f_2(t, x_1(t), \ldots, x_n(t), u_1(t), \ldots, u_m(t)) \\ \ldots \\ \dot{x}_n(t) = f_n(t, x_1(t), \ldots, x_n(t), u_1(t), \ldots, u_m(t)). \end{cases}$$

4.4.2 Aircraft Nonlinear State Space

It is possible to use such nonlinear state-space representation to create autopilots, as it is often the case in the aeronautical field by modeling the aircraft to pilot. Thanks to control engineering, the linearization of a system around its equilibrium point allows us to use all the useful tools provided by the science of automatics.

In this set of experiments, we work with an F3A airplane, which is often used in radio-controlled acrobatic airplane competitions. Indeed this plane has the capabilities to follow various trajectories without major structural constraints.

The choice of the state vector is:

$$x^T = [V, \alpha, \beta, p, q, r, q_1, q_2, q_3, q_4, N, E, h, T],$$

where V is the airspeed, α the angle of attack, β the heeling angle, p the x-axis rotation rate, q the y-axis rotation rate, r the z-axis rotation rate, q_1, q_2, q_3, q_4 the attitude quaternions, N the latitude, E the longitude, h the altitude, and T the real thrust.

The choice of the control vector is $u^T = [T_c\ \delta_e\ \delta_a\ \delta_r]$ where T_c is the commanded throttle, δ_e the commanded elevators, δ_a the commanded ailerons, and δ_r the commanded rudders, as in Fig. 14.

As described earlier, the nonlinear state-space representation is $\dot{x}(t) = f(t, x(t), u(t))$

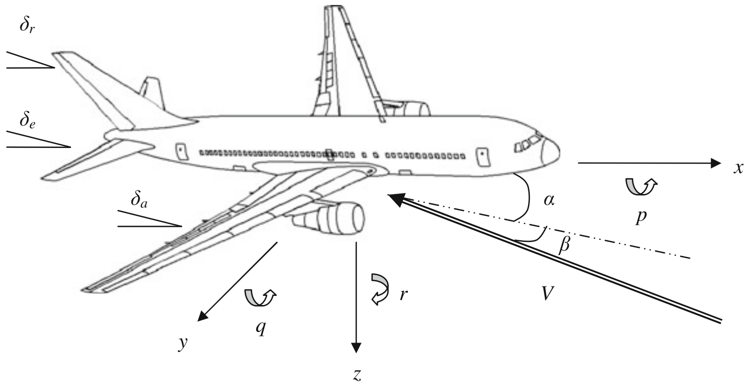


Fig. 14 State and control variables of an airplane

And more precisely:

$$\begin{cases} \dot{V}(t) = f_1(t, V(t), \alpha(t), \dots, T(t), T_c(t), \dots, \delta_r(t)) \\ \dot{\alpha}(t) = f_2(t, V(t), \alpha(t), \dots, T(t), T_c(t), \dots, \delta_r(t)) \\ \dots \\ \dot{q}_1 = f_7(t, V(t), \alpha(t), \dots, T(t), T_c(t), \dots, \delta_r(t)) \\ \dot{q}_2 = f_8(t, V(t), \alpha(t), \dots, T(t), T_c(t), \dots, \delta_r(t)) \\ \dot{q}_3 = f_9(t, V(t), \alpha(t), \dots, T(t), T_c(t), \dots, \delta_r(t)) \\ \dot{q}_4 = f_{10}(t, V(t), \alpha(t), \dots, T(t), T_c(t), \dots, \delta_r(t)) \\ \dots \\ \dot{h}(t) = f_{13}(t, V(t), \alpha(t), \dots, T(t), T_c(t), \dots, \delta_r(t)) \\ \dot{T}(t) = f_{14}(t, V(t), \alpha(t), \dots, T(t), T_c(t), \dots, \delta_r(t)). \end{cases}$$

4.4.3 Considered Functions

The original trajectories are created using the telemetric data of a F3A simulator. Starting from an airplane trajectory, the genetic programming problem consists of being able to recreate the state-space equation that is able to follow the original trajectory.

As each equation is considered at a different time, here we only regress the quaternion functions. It has to be noted that these functions have great influence on the considered trajectory as can be seen in the final trajectory results.

$$\begin{cases} \dot{q}_1 = f_7 = 0.5(q_4 p - q_3 q + q_2 r) \\ \dot{q}_2 = f_8 = 0.5(q_3 p + q_4 q - q_1 r) \\ \dot{q}_3 = f_9 = 0.5(-q_2 p + q_1 q + q_4 r) \\ \dot{q}_4 = f_{10} = 0.5(-q_1 p - q_2 q - q_3 r). \end{cases}$$

Table 2 Parameters used to regress a part of the airplane model

Population size	40,960
Number of generation	100
Function set	$+, -, *, /$
Terminal set	$x[1], \ldots, x[17]$, ERC {0,1}
Learning set	51,000 values

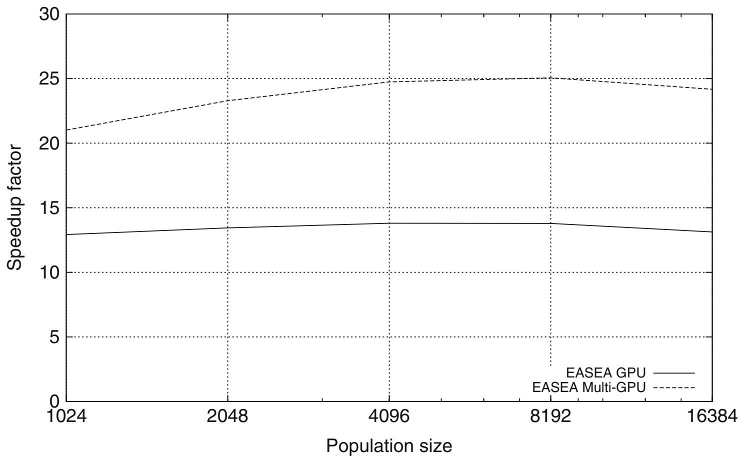


Fig. 15 Speedup obtained on the airplane problem with EASEA tree-based GP implementation

The learning set is constructed by using an artificial model (an F3A simulator) that generates telemetric data, from a few minute long flight. All the state variables $[V, \alpha, \beta, p, q, r, q_1, q_2, q_3, q_4, N, E, h, T]$ have been recorded, as well as the control variables $[T_c, \delta_e, \delta_a, \delta_r]$ and the time. The learning set contains 51,000 points, i.e., around 8 min of flight. Run parameters are summarized in Table 2.

Given a state/control vector, the role of the GP algorithm is to predict the value of the considered variable at the next time step. Using this principle, a run is needed for each function to regress (here four, one per quaternion).

As for the previous artificial problem 4.3.1, a theoretical speedup is computed, using the same technique. Here, these theoretical speedups range between 100 and 500. They are lower than the ones computed for the artificial problem, as known solutions only involve non-intensive arithmetical functions, thus, only simple operators as the one presented in Table 2.

Furthermore, the terminal set is larger (ERC, 17 variables), i.e., the GPU interpreter has to perform more memory accesses than the one presented in the benchmark work, where only variables can be stored in the registers by the optimizer. These drawbacks as well as the evolutionary process can be blamed for the drop in speedup (Fig. 15).

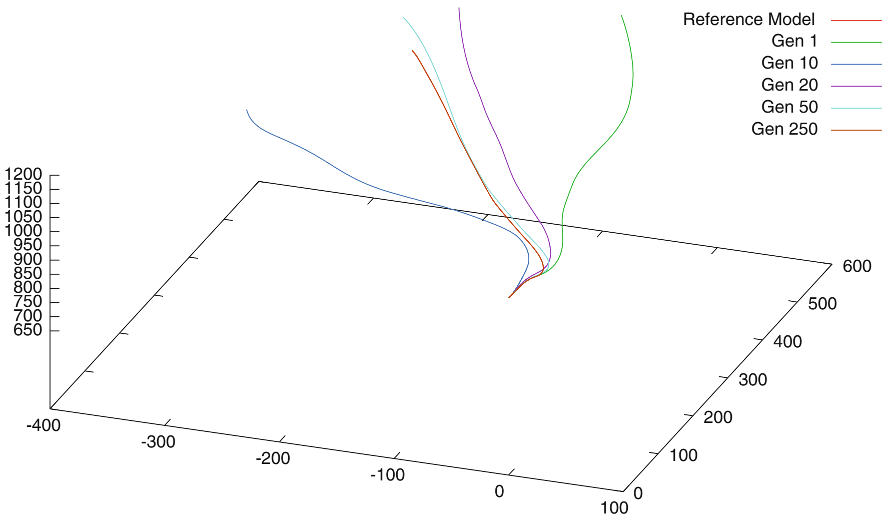


Fig. 16 Simulated trajectories, from the original model and the evolved one

Speedup factors still remain satisfactory given the size of the problem: a real run takes hours of computation on CPU, but just several minutes on GPU. This is very important, because this problem needs to regress 14 different functions.

Finally, Fig. 16 shows an example of a trajectory obtained with the evolved model as well as the trajectory resulting from the trained model.

Functions that have been evolved with the EASEA GP implementation have been used instead of the real ones. It has to be noted that quaternion functions that have been evolved through symbolic regression impact the plane trajectory in a very strong manner, meaning that errors in this part of the model will have noticeable consequences. In this example, the difference in trajectories between the reference model and the best trajectory obtained in generation 250 is so minimal that it is not possible to distinguish between them. The best trajectories obtained by the best individual of generations 1, 10, 20, and 50 show that wrong quaternion equations have a real influence on the resulting trajectory.

These results were presented in conference papers, in [10], which details the parallelization of the evaluation part, and in [6], where the whole algorithm has been detailed. The complete work was part of the ones published in [9].

5 Conclusion

This chapter introduces the use of GPGPUs for genetic programming. These algorithms being particularly greedy in terms of computing resources, the use of parallel architectures was considered soon after the beginnings of research on GP [4, 5].

GPGPUs are now employed in a number of publications, mainly using a master–slave model where the GPGPU serves for the evaluation of the population, while the rest of the algorithm continues to be executed in a conventional manner on the central processor.

Several methods have been developed, and we focus on an approach published in [10] that aims to improve the memory usage, using automatic scheduling, because NVIDIA cards are capable of this. We show that the use of scheduling can speed up the execution of the population evaluation, by improving the number of threads that are loaded on the processor. Some parameters and their impact on the behavior of the evaluation algorithm are analyzed here.

This approach, like the others presented, aims to obtain a maximum gain on the evaluation function, in order to accelerate the execution of the algorithm. Genetic programming, using such approaches, may now allow many researchers to have, with the help of these GPGPUs, great power in order to optimize more complex problems. This power was previously available only with expensive and powerful architectures, such as supercomputers.

References

1. Chitty, D.M.: A data parallel approach to genetic programming using programmable graphics hardware. In: GECCO '07: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, pp. 1566–1573. ACM, New York (2007)
2. Harding, S., Banzhaf, W.: Fast genetic programming on GPUs. In: EuroGP'07: Proceedings of the 10th European Conference on Genetic Programming, pp. 90–101. Springer, Berlin (2007)
3. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems). MIT Press, Cambridge (1992)
4. Koza, J.R., Rice, J.P.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge (1994)
5. Koza, J.R., Bennett III, F.H., Andre, D., Keane, M.A.: Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann, Los Altos (1999)
6. Lachiche, N., Maitre, O., Querry, S., Collet, P.: EASEA parallelization of tree-based genetic programming. In: IEEE CEC 2010 (2010)
7. Langdon, W.B., Banzhaf, W.: A SIMD interpreter for genetic programming on GPU graphics cards. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcazar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008, Naples, 26–28 March 2008. Lecture Notes in Computer Science, vol. 4971, pp. 73–85. Springer, Heidelberg (2008)
8. Lyshevski, S.E.: State-space multivariable non-linear identification and control of aircraft. Proc. Inst. Mech. Eng. G: J. Aerosp. Eng. **213**(6), 387–397 (1999)
9. Maitre, O., Lachiche, N., Collet, P.: Fast evaluation of GP trees on GPGPU by optimizing hardware scheduling. In: Genetic Programming. Lecture Notes in Computer Science, vol. 6021, pp. 301–312. Springer, Berlin (2010)
10. Maitre, O., Kruger, F., Querry, S., Lachiche, N., Collet, P.: EASEA: specification and execution of evolutionary algorithms on GPGPU. J. Soft Comput. **16**(2), 261–179 (2012)
11. Robilliard, D., Marion-Poty, V., Fonlupt, C.: Population parallel GP on the G80 GPU. In: Genetic Programming, pp. 98–109. Springer, Berlin (2008)
12. Robilliard, D., Marion, V., Fonlupt, C.: High performance genetic programming on GPU. In: Proceedings of the 2009 Workshop on Bio-inspired Algorithms for Distributed Systems, Barcelona, Spain, pp. 85–94. ACM, New York (2009)

Cartesian Genetic Programming on the GPU

Simon Harding and Julian F. Miller

Abstract Cartesian Genetic Programming is a form of Genetic Programming based on evolving graph structures. It has a fixed genotype length and a genotype–phenotype mapping that introduces neutrality into the representation. It has been used for many applications and was one of the first Genetic Programming techniques to be implemented on the GPU. In this chapter, we describe the representation in detail and discuss various GPU implementations of it. Later in the chapter, we discuss a recent implementation based on the GPU.net framework.

1 Introduction

Cartesian Genetic Programming (CGP) was one of the first Genetic Programming representations to take advantage of the general purpose computing capabilities of modern GPUs [5]. As with other evolutionary algorithms (EAs), CGP maps well to the GPU architecture and is able to exploit the massive parallelism. But because CGP is based on a fixed length graph that allows node reuse, its implementation is distinct from the typical tree-based Genetic Programming (GP).

For those unfamiliar with CGP, an overview of the representation is provided in Sect. 2. In Sect. 3 we provide a review of the previous work of CGP on GPU. We see that GPU implementations on CGP have been used not only for typical machine

S. Harding (✉)
Machine Intelligence Ltd, Exeter, UK, EX4 IEJ
e-mail: simon@machineintelligence.co.uk

J.F. Miller
Department of Electronics, University of York, York Y01 9UD, UK
e-mail: julian.miller@york.ac.uk

learning problems such as regression, classification and image processing but also for the fitness evaluation in complex fluid dynamics problems. Finally, in Sect. 4 a recent implementation using an unusual programming approach is introduced.

2 Cartesian Genetic Programming

We give a brief overview of CGP. A more detailed account is available in the recently published book [24]. In CGP [20, 21], programs are generally represented in the form of directed acyclic graphs. These graphs are often represented as a two-dimensional grid of computational nodes. The genes that make up the genotype in CGP are integers that represent where a node gets its data, what operations the node performs on the data and where the output data required by the user is to be obtained. When the genotype is decoded, some nodes may be ignored. This happens when node outputs are not used in the calculation of output data. When this happens, we refer to the nodes and their genes as "non-coding". We call the program that results from the decoding of a genotype a phenotype. The genotype in CGP has a fixed length. However, the size of the phenotype (in terms of the number of computational nodes) can be anything from zero nodes to the number of nodes defined in the genotype. The types of computational node functions used in CGP are decided by the user and are listed in a function lookup table.

In CGP, each node in the directed graph represents a particular function and is encoded by a number of genes. One gene is the address of the computational node function in the function lookup table. We call this a *function gene*. The remaining node genes say where the node gets its data from. These genes represent addresses in a data structure (typically an array). We call these *connection genes*. Nodes take their inputs in a feed-forward manner either from the output of nodes in a previous column or from a program input. The number of connection genes a node has is chosen to be the maximum number of inputs (often called the arity) of the node functions used.

In CGP program data inputs are given the absolute data addresses 0 to n_{i} minus 1 where n_{i} is the number of program inputs. The data outputs of nodes in the genotype are given addresses sequentially, column by column, starting from n_{i} to $n_{\mathrm{i}} + L_n - 1$, where L_n is the user-determined upper bound of the number of nodes. The general form of a Cartesian genetic program is shown in Fig. 1. If the problem requires n_{o} program outputs, then n_{o} integers are generally added to the end of the genotype. In general, there may be a number of output genes (O_i) which specify where the program outputs are taken from. Each of these is an address of a node where the program output data is taken from. Nodes in columns cannot be connected to each other. In many cases graphs encoded are directed and feed-forward; this means that a node may only have its inputs connected to either input data or the output of a node in a previous column. The structure of the genotype is seen in the schematic in Fig. 1. All node function genes f_i are integer addresses in a lookup table of functions.

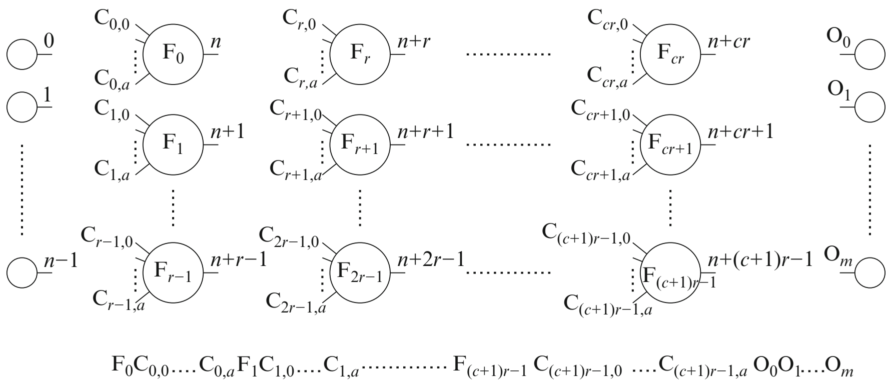


Fig. 1 General form of CGP. It is a grid of nodes whose functions are chosen from a set of primitive functions. The grid has n_c columns and n_r rows. The number of program inputs is n_i and the number of program outputs is n_o. Each node is assumed to take as many inputs as the maximum function arity a. Every data input and node output is labelled consecutively (starting at 0), which gives it a unique data address which specifies where the input data or node output value can be accessed (shown in the figure on the outputs of inputs and nodes)

All connection genes C_{ij} are data addresses and are integers taking values between 0 and the address of the node at the bottom of the previous column of nodes.

In our work here, we have used a one-dimensional geometry (one row of nodes). We have also adopted some of the features used in a developmental form of CGP [12]. We use *relative* connection addresses (rather than absolute), in which a connection gene represents how many nodes back (on the left) the node gets it inputs from. When these addresses point beyond the left end of the graph, zero is returned as a node input. The way we handle inputs and outputs is also different from classical CGP and also follows the method used in [12, 13]. This requires adding additional argument genes to all nodes (so nodes now have function, connection and argument genes), that is, a single positive floating point constant. Inputs are handled via functions: INP, INPP and SKIPINP. These functions ignore their connection genes; they return the input in the array of inputs given by an input pointer variable. After INP (INPP) the pointer is incremented (decremented). In the case of SKIPINP, the pointer is incremented by the argument of SKIPINP and then the mod operation (by the number of inputs) is taken (this ensures a valid input is always obtained). A function OUTPUT allows the input/output interpreter to find which nodes to use as output nodes at run time. The location and number of OUTPUT nodes can change over the run time of a program. When the genotype is decoded, the interpreter starts at the beginning of the encoded graph and iterates over the nodes until it finds the appropriate number of OUTPUT nodes. It then evaluates (recursively) from these nodes. The interpreter has features that allow it to cope when the number of OUTPUT nodes is different from the required number of outputs. If there are more OUTPUT nodes found than are needed, the excess nodes are simply ignored. If there are too few (or none), the interpreter starts using nodes from the end of the graph as outputs. This ensures that programs (of sufficient size) are always “viable”.

3 CGP on GPUs

Since 2007, several different implementations of CGP have been developed using different platforms and targeting different applications.

The first publication that reported an implementation of CGP on GPUs benchmarked the algorithm on regression and Boolean problems [5]. It showed that, compared to a naive, CPU-based $C\sharp$ implementation, the GPU was able to execute evolved programs hundreds of times faster. The paper used the Microsoft Accelerator framework [26], which is a *.Net* library for the GPU. Accelerator is limited to performing vector operations and cannot exploit the rich programming capabilities that recent graphics cards allow. However, it provides a simple method for GPU development. To run CGP using Accelerator, an existing CPU-based implementation was converted to use the Accelerator vector as the data type. The function set contained functions such as `add` and `subtract` that would operate on vectors and perform the chosen operation on the elements of the vectors to produce an output vector. Using this approach, each column of the input dataset was presented as a vector. The output of the program would be a single vector, containing the predicted outputs for each row. The fitness function was also implemented using Accelerator. On the GPU, the difference between the predicted and actual outputs was computed, and then a reduction was performed to find the sum of the differences. In this chapter, the EA and the interpreting of the genotype were done on the CPU, with the fitness evaluation and the execution of the individual done on the GPU.

A follow-up paper investigated the use of Accelerator for artificial developmental systems [6]. In this scenario, an artificial developmental system is defined as 2D cellular automata, where each cell contains a CGP program to define its update rules. For this problem, the GPU was seen as an ideal platform. Cellular automata are inherently parallel systems, with each cell determining the next state based on the state of its neighbours at the current time step. Although such developmental systems had been successfully implemented on the CPU [23], they tended to be small as the computational demands are high. Using the GPU, it was found that it was possible to execute large cellular automata, with complex programs and for more time steps than would have been practical on the CPU.

The update process of the artificial developmental system shares a large number of similarities with morphological image operations. Here, a kernel takes the values of a neighbourhood of pixels and outputs a new value for the centre pixel of that neighbourhood. A simple example is a smoothing operation that outputs a weighted sum of the neighbouring pixels. Although weighted sums are the most common forms of these kernels, it is possible to use complicated expressions, including evolved programs. As expected, it was found that CGP on the GPU was able to rapidly find such programs [4, 7, 8]. Before such GPU implementations were available, most work on evolving image filters was restricted to fitness evaluation with a single 256×256 (or smaller) image. This led to problems of overfitting. However, by implementing the processing on the GPU, many images, each with different properties, could be efficiently tested.

Initially, suitable GPUs for GPGPU were relatively expensive and obscure. However, they quickly became a standard component in recent systems. For example, when a student computer lab was updated, it became possible to test the use of multiple GPUs as part of an ad hoc cluster [11]. Previous versions of CGP on GPUs performed evaluated one individual at a time, with the fitness cases being operated on in parallel. Using a cluster of GPUs, it was possible to evaluate the population in parallel, which in turn increased the speed at which the population could be evaluated.

Evaluating programs with loops and recursion can be extremely computationally expensive, and this is one of the reasons that evolution of such programs is under-represented in the literature. Using GPUs though reduces this computational cost significantly. Although CGP is typically used to evolve feed-forward expressions, by removing the restriction that nodes must connect to previous columns, it is possible to evolve cyclic graphs. It is possible to evaluate cyclic CGP on the GPU very efficiently and process individuals hundreds of times faster than with a single CPU [18].

All of these previous examples looked at more traditional applications of GP, such as regression and classification. However, GP can also be used for other applications—such as shape design [2, 15, 19]. In [10], GPUs were used to evaluate a computational fluid dynamics (CFD) simulation to test an evolved wing design. A form of CGP based on Self-Modifying CGP [12, 13] was used to generate the cross-sectional shape of an airfoil. This was then simulated to determine its lift and drag properties. Using the GPU, the simulation speeds were dramatically improved. To further increase performance, an ad hoc cluster of GPUs was used to evaluate the population in parallel. Care was taken to ensure that the EA could run totally asynchronously, as CFD simulations vary in how long they take to execute depending on factors such as how much turbulence is generated.

4 Example: CGP for Classification

4.1 GPU.NET

As shown in previous sections, GPU programming often requires specialist programming skills. Although the development tools have been considerably improved in recent years, they are still difficult to work with. In the literature, we see that there are many different ways to program GPUs and that EA have been implemented with most of the common types. Most EAs have been written using NVidia's CUDA, with a few more recent examples using OpenCL. It is also possible to write low-level shader kernels (using Cg, OpenGL or DirectX); however these are targeted for graphics programming, and the language semantics make implementing "general purpose programs" trickier. Tools such as MS Accelerator can generate, or use, shader programs—but have the complexity abstracted away from the developer.

CGP has previously been implemented in both CUDA and Accelerator. However, in this chapter we present an implementation based on a recent commercial, closed-source tool for programming GPUs from TidePowerd called GPU.NET. Like cuda.net and MS Accelerator, GPU.NET is designed to work with Microsoft's .Net Common Language Runtime (CLR). GPU.NET's main feature is that it converts Intermediate Language (IL) in an already compiled .Net assembly to device code (e.g. PTX instructions for NVidia graphics cards). GPU.NET's tool rewrites the assembly converting flagged methods into kernels that can run on a GPU; it also automatically adds in new functionality to handle transferring data and launching the kernels. Kernels can be written in any .Net-managed language, such as $C\sharp$, and use the same threading and memory model as CUDA. An example kernel is shown in Listing 1.

Writing GPU code in this way has some benefits and also some drawbacks. Currently there are no supplied GPU debugging tools. However, it is very easy to execute the code on the CPU and use the CPU debugging tools. Further there is difficulty in determining how the code has been translated and what optimisations have been (or can be) made. It should be noted that the generated PTX is further compiled by the device driver before execution. Compared to a compiled native application, there is a small overhead in using .Net to call unmanaged code (i.e. the functions in the device driver). Each function call to native code can have an overhead of a few microseconds. Therefore, as with all GPU programming, care has to be taken to ensure that the unit of work given to the GPU is sufficiently large to make the overheads negligible.

Listing 1 Example showing the kernel code to element-wise add two vectors of numbers together

```
[Kernel]
private static void Add(float[] input1, float[] input2, float[] result) {
    int ThreadId =  BlockDimension.X * BlockIndex.X + ThreadIndex.X;
    int TotalThreads = BlockDimension.X * GridDimension.X;

    for (int i = ThreadId; i < input1.Length; i += TotalThreads)
        result[i] =  input1[i] + input2[i];
}

public static void AddGPU(float[] input1, float[] input2, float[] result) {
    Launcher.SetBlockSize(128);
    Launcher.SetGridSize(input1.Length / (128*128));
    Add(input1, input2, result);
}
```

4.2 A GPU-Based Interpreter for CGP

Looking back at previous CGP implementations and other work in the field, we see that there are two methodologies for implementing GP on GPUs. One is to compile native programs for executions on the GPU, where candidate individuals are converted to some form of source code and then compiled before fitness testing. In an early example of GP on the GPU, one implementation compiled individual programs from generated Cg shader code and then loaded into the GPU for execution [1]. The MS Research Accelerator toolkit generates and compiles a DirectX shader program transparently. It then executes this program. In [9], CUDA C programs were generated from the GP individual, compiled to a GPU "cubin" PTX library and then executed on the GPU. With CUDA, this PTX code needs further just-in-time compiling by the graphics driver.

This process of pre-compilation leads to a significant time overhead and in general means that this approach is most suited where there is a large amount of data to be processed and the evolved programs are sufficiently complicated.

Another methodology involves writing an interpreter that runs on the GPU as a kernel, which executes a set of operations on the data. The interpreter runs the same evolved program over every element in the dataset and does so in parallel. Typically in the interpreters, each thread only considers one fitness case. The benefit of this approach is that the GPU program only needs to be compiled and loaded once and that the only thing that changes is the data that represents the programs to be tested for fitness. The interpreter approach has been used with CUDA [17,25] and DirectX shaders [27]. Without the pre-compilation overhead, the interpreters work well for smaller amounts of data and for shorter programs. However, interpreters introduce their own performance issues, as there is now the continuous overhead of parsing the evolved program in addition to performing the computation. Fortunately on GPUs, the branching that is typically required in an interpreter is handled very efficiently.

With a GP tree, the interpreter for the tree can be a very straightforward implementation. GP trees are typically binary trees, and there is no obvious redundancy in the encoding. This means that interpreters can use and read a reverse Polish version of the tree and use only two registers to store intermediate results.

The representation for CGP introduces a number of features that mean the implementation is slightly more complicated. In CGP nodes in the graph are often reused; therefore, the result for that computation (and all proceeding computations in that node's subtree) can be cached to avoid re-computing their value. With the normal CPU implementations, this is not a problem as there is typically a lot of working memory available and the performance penalties for what memory is used are hidden by the compiler. With a CUDA GPU, there is a multilayered memory hierarchy that can be extremely limited, depending on the "compute capabilities" of the hardware.

For versions 1.0 and 1.1, each multiprocessor has only 8,000 32-bit registers. In version 1.2, this was increased to 16,000 and in version 2 this is 32,000. The registers are the fastest memory to work with on the GPU. The next level up in the hierarchy is the shared memory. This is also fast memory, but again is very limited with 16 KB of storage on 1.x compute capabilities and 48 KB on 2.x compatible hardware. The top level in the hierarchy is the global memory that can be very large, but can have significant performance penalties in use, as there may be conflicts and caching issues.

With the CGP interpreter, we have to be very aware of how these memory constraints will impact the node reuse. CGP genotypes typically include a lot of redundancy in the form of neutral, unconnected nodes. In a good CPU implementation, the nodes in the graph will be analysed before execution to determine which nodes are involved in the computation. This means that not all nodes in the graph need to be executed. In a GP tree, all nodes are active in the computation (unless there are simplifications that can be determined before execution). CGP genotypes are fixed length, so the maximum number of program operations is limited; further the length of the program stored in the genotype does not bloat over time [22], which has implications for perceived efficiency that we observe later in this chapter. Further, CGP also allows for multiple program outputs to be considered. This also impacts the implementation, as each of these outputs needs to be held in memory.

The function set used here consists of the mathematical operations $+, -, \div, \times$, log(), exp(), $\sqrt{()}$ and sin(). The genotype size was set to 100,000 nodes. Each gene has an equality probability (0.01) of being mutated. A 1+4 evolutionary strategy (ES) was used [24].

To test using GPU.NET, three different interpreter strategies were considered. All three strategies were implemented to run the same format instruction code (IC). In the next section, the instruction code is discussed. The following section describes how the different strategies worked.

4.3 Generating the Instruction Code

Generating the "interpreted code" (IC) from CGP requires a number of steps, which are all performed on the CPU side. In the first step, the genotype is analysed to determine which nodes are to be used as outputs. This is done by reading through the nodes to find special "OUTPUT" functions. More information on this approach can be found in [12,13]. Once the output nodes are determined, the active computational nodes can be found. Essentially this is done by recursing backwards through the connections from each of the outputs, but in reality this can be most efficiently implemented in a linear way. At the same time the number of times a node is reused is also counted.

From this the IC itself can be generated. In this interpreter model, each line in the IC contains a function, the addresses of two source registers and the address of a destination register. The interpreter here allows for ten working registers. When the interpreter runs a program it performs the function on the values stored in the source registers and stores the output value in the destination register.

Hence the next step in the process is to work forward through the active nodes to write out each step of the IC. For each node, a register is selected (from a stack of currently available registers) to store the output result in. This register address is then stored in a dictionary alongside the node index. The source registers for this node are computed using previous entries in this dictionary.

There are two circumstances in which there will be no source register in the dictionary: nodes that return inputs and nodes that have connections whose relative address is beyond the edges of the graph, in the former such connections return zero.

For functions that return a program input, the interpreter has a special instruction that can load (i.e. copy) a value from the input dataset to a destination register.

Using the counter stored in the initial parsing and by keeping count of how many times the register was read from, the converter knows when a value in a register will no longer be used. When this happens that memory location is released to be used by further operations. To release the address, the address is just pushed back into the stack of available registers.

In the work here, we consider only one output per program. This value is copied into shared memory so that it can be returned from the GPU. In future work, this can easily be expanded to copy the output to shared memory whenever an output node is computed.

4.4 The Interpreters

Three different interpreters were written. The first places the interpreter loop on the host (CPU) side, with the program instructions implemented as linear algebra-style vector operations. In this setup, the interpreter's loop runs on the host and calls a GPU function to operate on a vector of data. Each vector represents one column of data in the dataset, with the data being aligned so that all elements at a given index are from the same row. Having the interpreter loop on the CPU removes the need for the GPU to implement a branching structure. In future systems, it would also simplify the implementation of operations that operate across rows (such as "min" or "max"). The implementation of these vector operations is also convenient as the memory access does not run into any bottlenecks, as the memory access patterns are optimal for the GPU. The interpreter uses the register pattern described above to hold intermediate results, with each register being a vector in global memory. As the length of these vectors is the same as the number of rows in the data, there may be issues with memory. The GPU needs enough global memory to hold the dataset, as well as the output vectors and ten vectors to use as working registers (see Sect. 4.5).

TidePowerd does not allow for the creation of arrays that are local to a kernel. So the next interpreter method uses the fast shared memory to store the working register values. However, as mentioned previously, there is a lack of available shared memory on the GPU. This means that the number of concurrent threads (i.e. the size of the thread block) has to be relatively small. On the compute capability 1.3 devices used here, the shared memory is 16 KB. With ten working registers using 4 bytes per floating point number, there is (theoretically) a maximum of 409 threads per block. However, we used a thread block size of 384, which is the next smallest power of 2. This, in principle, should allow for full occupancy of the GPU.

As the number of threads per block is very limited and fewer than the number of processors available, there may be a risk of underutilising the GPU. In the third implementation, the registers were moved from the fast shared memory to slower global memory. This then meant that more threads could execute per block—and

Table 1 For the interpreter, program length and the number of working registers required are important parameters. These results show how they vary depending on the size of the genotype

	Program length		Peak registers used	
Genotype width	Avg.	Std. dev.	Avg.	Std. dev.
16	3.6	2.4	2.6	0.8
32	4.6	3.3	2.8	1.0
64	5.2	4.2	3.0	1.2
128	6.0	5.4	3.1	1.4
256	6.6	6.3	3.2	1.6
512	8.0	8.3	3.5	2.0
1,024	9.0	9.2	3.6	2.0
2,048	9.2	10.1	3.6	2.1
4,096	10.5	12.4	3.8	2.5
8,192	12.4	14.9	4.0	2.7
16,384	13.6	16.6	4.2	3.0
32,768	14.8	18.4	4.3	3.3
65,536	17.3	21.8	4.7	3.8
131,072	16.8	23.2	4.5	3.8
262,144	19.7	25.3	5.0	4.3

hence, more processors could be used in parallel. However, now the register memory is a slower resource.

In previous work using interpreters, each thread dealt with one fitness case at a time. In this work, we also investigated the possibility of testing multiple fitness cases per thread.

4.5 *Sizing for Interpreter Parameters*

To determine the number of registers needed and the maximum amount of storage we needed for the interpreter, we analysed the requirements of randomly generated genotypes. Although this does not provide the true requirements of evolved programs (which can grow within the bounds of the genotype), it gives an indication of the requirements.

Table 1 shows the behaviour of the program length and maximum number of working registers required for various program lengths. For the working registers (the more constrained parameter) we found that 8.5 registers should be sufficient 95 % of the time. For convenience, we chose to use ten registers (which covers 97 % of the randomly generated programs). The maximum program length that can be used with GPU.NET is more flexible. The results indicated that 50 operations should be sufficient, but as this resource is not as limited, we use a maximum length of 200 operations.

4.6 Fitness Function

For benchmarking, two fitness functions were implemented. Both are based on the KDD Cup Challenge 1999 data [3, 16]. One fitness function was a typical classification problem where the fitness of an individual was a measure of the accuracy in detecting normal or abnormal network traffic (in the original problem, the type of abnormal traffic is the classifier output). The fitness itself is also calculated using a kernel on the GPU. To calculate the score, 512 threads were launched (each to operate on $1/512$ the data). Each thread counts the number of true/false positives/negatives within a section of all the program outputs, which are then combined host side to find a confusion matrix. From this the sensitivity-specificity was calculated.

For the second fitness function, the fitness was the number of instructions in a program. Informal testing, and previous experience with GPUs, showed that longer programs are more efficient—and a higher speed-up can be achieved. Where programs are generated directly from trees, linear genetic programming, etc., then all operations in the genotype are executed. In CGP, because of redundancy, this is not the case. With the bloat-free evolution of CGP, we would also expect programs to be more compact. Therefore this second fitness function gives a better idea of the maximum capabilities of the system. The first fitness function however gives a more realistic impression of how CGP on GPU behaves with a typical problem.

4.7 Speed Results

As with previous papers on GP on GPUs, we measure the speed by counting the number of Genetic Programming Operations Per Second (GPOps/s) that can be performed. This measure includes all the overheads (e.g. transferring data, programs as well as executing the interpreter), and so it is expected to be significantly less than the theoretical Floating Point Operations Per Second (FLOPS) that is often quoted when measuring GPU speed. Unless stated otherwise here, the GPOps/s are reported just for program interpreter stage and not the fitness evaluation stage. It should also be noted that MGPOps/s and GGPOps/s are used to indicate mega-GPOps/s and giga-GPOps/s, respectively. To measure the speed, multiple evolutionary runs were performed, with each individual evaluation benchmarked. These were then re-sampled by picking a number of evaluation results at random.

The computer has an AMD 9950 (2.6 GHz) processor, running Windows 7, with a Tesla C1060 (240 CUDA cores, 4 GB, driver version 258.96).

4.8 Vector

We first present the results for the "vector" approach. Listing 1 shows an example $C\sharp$ kernel for adding two vectors together. TidePowerd's converter requires the kernels be private static written inside an internal static class. The kernel code itself is also flagged with the [kernel] attribute. A public static method (which runs host side) is then used to call this method. When the converter tool is used, it rewrites the class file inside the assembly to add in code so that the host-side method can call the GPU kernel. The methods flagged with [kernel] have their IL converted to device code to run on the GPU. As the listing shows, the kernel code is very similar to CUDA. However, the language conventions are all $C\sharp$.

In the interpreter, the main loop iterates over the instructions and calls methods (such as the Add function) on the vector data. Since this methodology is also most suited for CPU execution, we report the timing information for a CPU version to provide a comparison. The CPU version uses only one core. Although the data type is float, .Net only provides double-precision versions of the non-primitive mathematical operators.

For the CPU version, we found that the GPOps/s were independent of the length of the program and (largely) of the number of elements in the vector. On average, the CPU version was capable of 64 MGPOps/s (standard deviation 13.2, 1,000 results) with a minimum 18.3 and maximum of 139 MGPOps/s.

We were unable to successfully run the GPU version on the complete dataset as there was insufficient memory to hold the input data and the working registers (ten registers, each the size of number of rows of the input data). Therefore, these benchmarks are formed using 2,000,000 rows (half the data). GPU.NET is able to work with multiple GPUs, so it would be possible to implement the software to span the data over multiple devices.

On the reduced dataset, the GPU vector interpreter was able to perform an average of 144 MGPOps/s (std. dev. 12.8, 1,000 results sampled) and a minimum of 64 and a maximum of 199 MGPOps/s. The speed again was independent of the length of the evolved program.

4.9 Global and Shared Memory

Listing 2 shows a section of the kernel source code for interpreting an evolved program on the GPU. The arrays Ops, Src0, Src1 and Dest encode the evolved program's operations. Data is a pointer to the input data to the programs. Outputs are placed into the output array. The Regs array is the working registers and is held in global memory. The other kernel parameters specify the length of the program, dimensions of the input data, number of registers available and the number of test cases to process per kernel. Listing 3 shows a similar kernel, but with the working registers now held in the faster shared memory.

Both kernels have stability issues when working when the WorkSize (the number of test cases per thread) was increased and the program would crash. It is unclear why this occurs, as the CPU version of these kernels appears to function correctly. Unfortunately TidePowerd does not yet provide debugging tools, so we were unable to find the cause. Both types of failure are evident in Figs. 2 and 3, where the

Listing 2 Kernel for interpreting a program on the GPU, using global memory to store register results for intermediate workings

```
[Kernel]
private static void RunProgGM(int[] Ops, int[] Src0, int[] Src1,
int[] Dest, float[] Data, float[] Output, float[] Regs,
int ProgLength, int DataSize, int DataWidth, int RegCount, int WorkSize) {
    int ThreadID = BlockDimension.X * BlockIndex.X + ThreadIndex.X;
    int TotalThreads = BlockDimension.X * GridDimension.X;
    int RegOffset = ThreadIndex.X * RegCount;
    int Op = 0; float vSrc0 = 0; float  vSrc1 = 0;

    //Clear registers (omitted for space)

    for (int v = 0; v < WorkSize; v++)  {
        int RegOffset2 = (ThreadIndex.X * RegCount);

        for (int p = 0; p < ProgLength; p++) {
            Op = Ops[p];

            if (Op == -1) break;
            if (Op == 100) {   //load
                    Regs[RegOffset2 + Dest[p]] =
                            Data[(WorkSize * ThreadID * DataWidth) + v + Src0[p]];
                    continue;
                }
            vSrc0 = Regs[RegOffset2 + Src0[p]]; vSrc1 = Regs[RegOffset2 + Src1[p]];
            if (Op == 1) //add
                Regs[RegOffset2 + Dest[p]] = vSrc0 + vSrc1;
            ... //abridged for space
            else if (Op == 8) //sqrt
                Regs[RegOffset2 + Dest[p]] =
                TidePowerd.DeviceMethods.DeviceMath.Sqrt(vSrc0);
        }
        Output[(ThreadID * WorkSize) + v] = Regs[RegOffset2 + (RegCount - 1)];
    }
}
```

Listing 3 Kernel for interpreting a program on the GPU, using shared memory to store register results for intermediate workings

```
[SharedMemory(10 * 384)]  //10 working registers, 384 concurrent threads.
private static readonly float[] Regs = null;
[Kernel]
private static void RunProgSM(int[] Ops, int[] Src0, int[] Src1,
int[] Dest, float[] Data, float[] Output,
int ProgLength, int DataSize, int DataWidth, int RegCount, int WorkSize) {
    int ThreadID = BlockDimension.X * BlockIndex.X + ThreadIndex.X;
    int TotalThreads = BlockDimension.X * GridDimension.X;
    int RegOffset = ThreadIndex.X * RegCount;
    int Op = 0;  float vSrc0 = 0; float vSrc1 = 0;

    //Clear registers

    for (int v = 0; v < WorkSize; v++)  {
        int RegOffset2 = (ThreadIndex.X * RegCount);

        for (int p = 0; p < ProgLength; p++) {
            Op = Ops[p];

            if (Op == -1) break;
            if (Op == 100) { //load
                Regs[RegOffset2+ Dest[p]] =
                      Data[(WorkSize * ThreadID * DataWidth) + v + Src0[p]];
                continue;
            }

            vSrc0 = Regs[RegOffset2 + Src0[p]]; vSrc1 = Regs[RegOffset2 + Src1[p]];
            if (Op == 1) //add
                Regs[RegOffset2+ Dest[p]] = vSrc0 + vSrc1;
             ... //abridged for space
            }

        Output[(ThreadID * WorkSize) + v] =
            Regs[RegOffset2 + (RegCount - 1)]; ;
    }
}
```

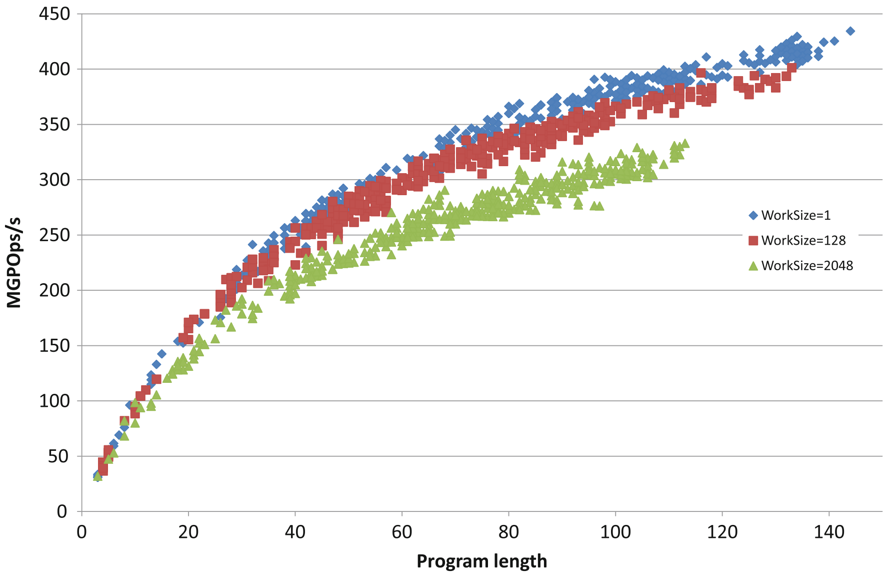


Fig. 2 Graph showing how the speed is dependent on both the length of the evolved program (in operations) and the number of test cases handled per thread (WorkSize). The results here are for the interpreter using global memory. Missing results are due to program instability

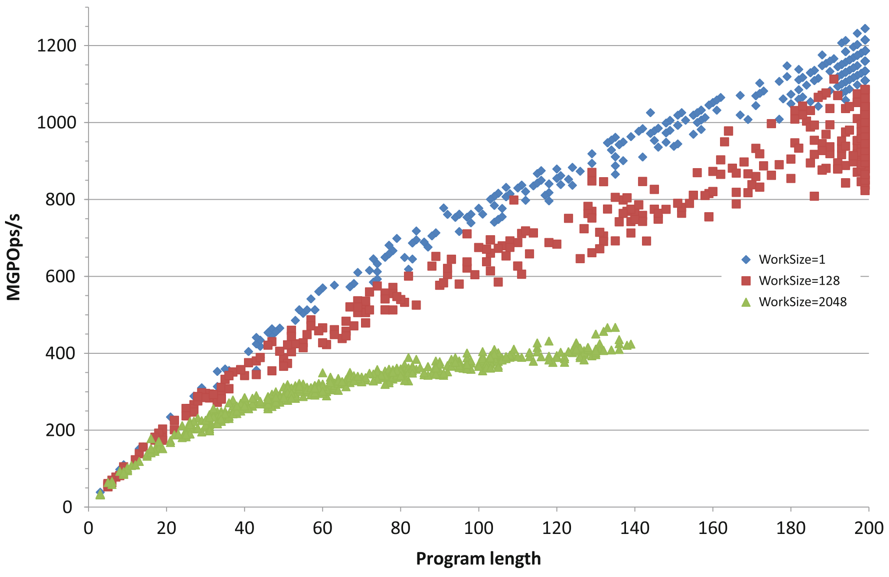


Fig. 3 Graph showing how the speed is dependent on both the length of the evolved program (in operations) and the number of test cases handled per thread (WorkSize). The results here are for the interpreter using shared memory. Using shared memory with GPU.NET is much more stable and faster than using global memory

Table 2 Speed of the two different interpreters, in MGPOps/s

Interpreter	Global memory			Shared memory		
WorkSize	1	128	2,048	1	128	2,048
Minimum	31	37	32	38	53	32
Maximum	434	401	333	1,244	1,113	467
Average	317	286	255	1,009	756	307
Std. dev.	84	84	53	249	279	85

sampled results do not have the same coverage. However, the shared memory interpreter when used with WorkSize = 1 appeared to consistently work. As there is a limited amount of shared memory, the block size was set to 384 threads. For the global memory kernel, a block size of 512 threads was used.

Table 2 shows statistical results from both interpreter types. Five hundred samples per WorkSize were used. It can be clearly seen that the shared memory version is much faster. It is interesting to see that giving each thread more test cases to work with (and hence fewer threads running) reduced performance. The most efficient approach is to have one test case per thread.

The graphs in Figs. 2 and 3 show how the interpreters' performance is dependent on both the length of the evolved program and the number of test cases each thread handles. The longer the program length, the more efficient the interpreter becomes. However, in this scenario the fitness function was to find long programs. In real-world situations, long programs may not be desirable (and may even be penalised by the fitness function) or may not occur due to the GP representation or operators used. With CGP, we do not expect programs to grow over time [22], and therefore we would not expect the same high performance as demonstrated here (in Sect. 4.11, this effect is investigated). With other forms of GP, such as basic versions of tree-based GP, we would expect the program length to increase over time. Since this performance-to-program length relationship appears in other GPU papers, it would be interesting to know how the apparent efficiency of the implementations would be effected by forcing the evolution to reduce program length.

4.10 Fitness Scores

The time required to compute the confusion matrix for the fitness scores is, in principle, only dependent upon the number of test cases. It was found that this took on average 0.04 s to compute. However, the timings showed a large variance (a standard deviation of 0.1 s, median 0.03 s). It is unclear why this should occur, and perhaps when better debugging tools are available, the reason will become apparent. Fitness evaluation time was also hampered by the need to do large memory transfers to move the predicted and expected outputs to the GPU. This is a current limitation with GPU.NET where the memory management cannot be hand-optimised.

4.11 CGP Classification Results

Both to test CGP as a classifier and to investigate the GPU implementation under a more real-world environment, longer experiments (a maximum of 10,000 evaluations) were run using the shared memory approach, with WorkSize = 1. Fitness here was the sensitivity-specificity metric discussed previously.

Looking at performance, we found that the average speed was 210 MGPOps/s (std. dev. 126, maximum 701) for executing the evolved programs. When the fitness evaluation itself was taken into consideration, the average performance was 192 MGPOps/s (std. dev. 118, maximum 657). As discussed previously, the average CPU speed was 64 MGPOps/s, which means that with a real fitness function, this method produces, approximately, a three times speed-up on average and a peak of just over ten times speed-up. The relatively limited speed-up is largely due to the short programs found by CGP. The average program contained only 20 operations (std. dev. 14, min. 3 and maximum 84). These are therefore on the borderline of the area where the GPU produces an advantage. For this task, it may have been more efficient to use the multicore CPU to perform the evaluations. It would have also been possible to implement this application to use multiple GPUs, and this would have also led to a performance increase. As these efficiencies are based on these particulars of this classification task, observed speed-ups in other applications will be different.

Although not the focus of this chapter, it is worth noting that the classification results from CGP appear to be very good. In over 50 experiments, it was found that the average fitness (sensitivity-specificity) was 0.955 (std. dev. 0.034). The maximum classification rate found was 99.6 %; however without a validation test, we cannot exclude overfitting. Modifying the fitness function to collect both training and validation fitness scores would be a straightforward extension.

5 Conclusions

CGP was one of the first Genetic Programming techniques implemented on the GPU and has been successfully used to solve many different problem types using GPUs. Although this chapter focused on TidePowerd's implementation, CGP has successfully been developed using technologies such as CUDA and MS Accelerator. In all circumstances, significant speed-ups have been reported.

As GPU technology improves, both in terms of hardware features and software development, it is likely that more advanced CGP approaches such as SMCGP and MT-CGP [14] will be implemented. SMCGP requires a very flexible environment to work in, as programs can change dramatically during their run time. MT-CGP operates on multiple data types, and this presents some interesting challenges when developing a high-performance system. However, as GPGPU becomes ever more flexible, we expect that CGP will be able to take advantage of the new capabilities.

Acknowledgements SH acknowledges the funding from Atlantic Canada's HPC network ACENET and by NSERC under the Discovery Grant Program RGPIN 283304-07. SH would like to thank Innervision Medical Ltd. and TidePowerd for their technical assistance.

References

1. Chitty, D.M.: A data parallel approach to genetic programming using programmable graphics hardware. In: Thierens, D., Beyer, H.G., et al. (eds.) GECCO '07: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, vol. 2, pp. 1566–1573. ACM Press, London (2007)
2. Coates, P.: Using Genetic Programming and L-systems to explore 3D design worlds. In: CAADFutures'97. Kluwer Academic, Dordecht (2008)
3. Elkan, C.: Results of the KDD'99 classifier learning contest. http://cseweb.ucsd.edu/~elkan/clresults.html (1999)
4. Harding, S.: Evolution of image filters on graphics processor units using Cartesian genetic programming. In: Wang, J. (ed.) 2008 IEEE World Congress on Computational Intelligence. IEEE Computational Intelligence Society, IEEE Press, Hong Kong. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4631051 (2008)
5. Harding, S., Banzhaf, W.: Fast genetic programming on GPUs. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) Proceedings of the 10th European Conference on Genetic Programming. Lecture Notes in Computer Science, vol. 4445, pp. 90–101. Springer, Valencia (2007). doi:10.1007/978-3-540-71605-1_9. http://www.springerlink.com/index/w57468k30j124410.pdf
6. Harding, S.L., Banzhaf, W.: Fast genetic programming and artificial developmental systems on GPUs. In: 21st International Symposium on High Performance Computing Systems and Applications (HPCS'07), p. 2. IEEE Computer Society, Canada (2007). doi:10.1109/HPCS.2007.17. http://doi.ieeecomputersociety.org/10.1109/HPCS.2007.17
7. Harding, S., Banzhaf, W.: Genetic programming on GPUs for image processing. In: Lanchares, J., Fernandez, F., Risco-Martin, J. (eds.) Proceedings of the First International Workshop on Parallel and Bioinspired Algorithms (WPABA-2008), Toronto, Canada, 2008, pp. 65–72. Complutense University of Madrid Press, Madrid. http://www.inderscience.com/search/index.php?action=record&rec_id=24207&prevQuery=&ps=10&m=or (2008)
8. Harding, S., Banzhaf, W.: Genetic programming on GPUs for image processing. Int. J. High Perform. Syst. Archit. **1**(4), 231–240 (2008). doi:10.1504/IJHPSA.2008.024207. http://www.inderscience.comkern-1pt/search/index.php?action=record&rec_id=24207&prevQuery=&ps=10&m=or
9. Harding, S.L., Banzhaf, W.: Distributed genetic programming on GPUs using CUDA. In: Hidalgo, I., Fernandez, F., Lanchares, J. (eds.) Workshop on Parallel Architectures and Bioinspired Algorithms. Raleigh, USA. http://www.evolutioninmaterio.com/preprints/CudaParallelCompilePP.pdf (2009)
10. Harding, S., Banzhaf, W.: Optimizing shape design with distributed parallel genetic programming on GPUs. In: Fernández de Vega, F., Hidalgo Pérez, J.I., Lanchares, J. (eds.) Parallel Architectures and Bioinspired Algorithms. Studies in Computational Intelligence, vol. 415, pp. 51–75. Springer, Berlin (2012)
11. Harding, S.L., Banzhaf, W.: Distributed genetic programming on GPUs using CUDA. http://www.evolutioninmaterio.com/preprints/Technicalreport (submitted)
12. Harding, S., Miller, J.F., Banzhaf, W.: Developments in Cartesian genetic programming: self-modifying CGP. Genet. Program. Evolvable Mach. **11**(3–4), 397–439 (2010)
13. Harding, S., Miller, J.F., Banzhaf, W.: A survey of self modifying CGP. Genetic Programming Theory and Practice, 2010. http://www.evolutioninmaterio.com/preprints/ (2010)

14. Harding, S., Graziano, V., Leitner, J., Schmidhuber, J.: MT-CGP: Mixed type cartesian genetic programming. In: Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference, pp. 751–758. ACM, New York (2012)
15. Hotz, P.E.: Evolving morphologies of simulated 3D organisms based on differential gene expression. In: Proceedings of the Fourth European Conference on Artificial Life, pp. 205–213. Elsevier Academic, London (1997)
16. KDD Cup 1999 Data: Third international knowledge discovery and data mining tools competition. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html (1999)
17. Langdon, W.B.: A many threaded CUDA interpreter for genetic programming. In: Esparcia-Alcázar, A.I., Ekárt, A., et al. (eds.) Genetic Programming. Lecture Notes in Computer Science, vol. 6021, pp. 146–158. Springer, Berlin (2010)
18. Lewis, T.E., Magoulas, G.D.: Strategies to minimise the total run time of cyclic graph based genetic programming with GPUs. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, GECCO '09, pp. 1379–1386. ACM, New York (2009). doi:10.1145/1569901.1570086. http://doi.acm.org/10.1145/1569901.1570086
19. Lohn, J.D., Hornby, G., Linden, D.S.: Human-competitive evolved antennas. AI EDAM **22**(3), 235–247 (2008)
20. Miller, J.F.: An empirical study of the efficiency of learning Boolean functions using a Cartesian genetic programming approach. In: Proceedings of Genetic and Evolutionary Computation Conference, pp. 1135–1142. Morgan Kaufmann, Los Altos (1999)
21. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Proceedings of European Conference on Genetic Programming. Lecture Notes in Computer Science, vol. 1802, pp. 121–132. Springer, Berlin (2000)
22. Miller, J.: What bloat? Cartesian genetic programming on Boolean problems. In: Goodman, E.D. (ed.) 2001 Genetic and Evolutionary Computation Conference Late Breaking Papers, pp. 295–302. Morgan Kaufmann (2001)
23. Miller, J.F.: Evolving a self-repairing, self-regulating, French flag organism. In: Deb, K., Poli, R., Banzhaf, W., et al. (eds.) GECCO (1). Lecture Notes in Computer Science, vol. 3102, pp. 129–139. Springer, Berlin (2004)
24. Miller, J.F. (ed.): Cartesian Genetic Programming. Natural Computing Series. Springer, Berlin (2011)
25. Robilliard, D., Marion-Poty, V., Fonlupt, C.: Genetic programming on graphics processing units. Genet. Program. Evolvable Mach. **10**(4), 447–471 (2009)
26. Tarditi, D., Puri, S., Oglesby, J.: Accelerator: using data parallelism to program GPUs for general-purpose uses. In: ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 325–335. ACM, New York (2006). http://doi.acm.org/10.1145/1168857.1168898
27. Wilson, G.C., Banzhaf, W.: Deployment of parallel linear genetic programming using GPUs on PC and video game console platforms. Genet. Program. Evolvable Mach. **11**(2), 147–184 (2010)

Implementation Techniques for Massively Parallel Multi-objective Optimization

Deepak Sharma and Pierre Collet

Abstract Most real-world search and optimization problems involve more than one goal. The task of finding multiple optimal solutions for such problems is known as multi-objective optimization (MOO). It has been a challenge for researchers and practitioners to find solutions for MOO problems. Many techniques have been developed in operations research and other related disciplines, but the complexity of MOO problems such as large search spaces, uncertainty, noise, and disjoint Pareto curves may prevent the use of these techniques. At this stage, evolutionary algorithms (EAs) are preferred and have been extensively used for the last two decades. The main reason is that EAs are population-based techniques which can evolve a Pareto front in one run. But EAs require a relatively large computation time. A remedy is to perform function evaluations in parallel. But another bottleneck is Pareto ranking in multi-objective evolutionary algorithms. In this chapter, a brief introduction to MOO and techniques used to solve MOO problems is given. Thereafter, a GPGPU-based archive stochastic ranking evolutionary algorithm is discussed that can perform function evaluations and ranking on a massively parallel GPU card. Results are compared with CPU and GPU versions of NSGA-II.

D. Sharma (✉)
Department of Mechanical Engineering, Indian Institute of Technology Guwahati, Guwahati-781039, Assam, India
e-mail: dsharma@iitg.ernet.in

P. Collet
Université de Strasbourg, ICUBE - Illkirch, France
e-mail: pierre.collet@unistra.fr

1 Introduction

In the last two decades, the field of multi-objective optimization (MOO) has attracted researchers and practitioners who solve real-world optimization problems. Often, it has been observed that real-world problems deal with more than one objective. Solving such MOO problems generates a set of trade-off solutions which are known as Pareto-optimal solutions. Many optimization algorithms exist in the literature for MOO problems, but heuristic population-based algorithms (also known as multi-objective evolutionary algorithms (MOEAs)) are the most suitable for evolving trade-off solutions in one run [2, 5].

Though many MOEAs have been developed, there are only a few dominance-ranking-based algorithms that are really effective for solving MOO problems. Mostly, these MOEAs differ in their ranking methods which help to select and propagate good individuals to the next iteration. According to [2], dominance ranking methods can be categorized by dominance rank, dominance count, and dominance depth. Multi-objective genetic algorithm (MOGA) [9] and Niched Pareto Genetic Algorithm (NPGA) [15] use a dominance rank method by checking the number of solutions that dominate an individual. The elitist Non-dominated Sorting Genetic Algorithm II (NSGA-II) [7] sorts individuals according to dominance depth, using the concept of non-dominated sorting [13]. Strength Pareto Evolutionary Algorithm 2 (SPEA2) [28] assigns rank, based on dominance depth and dominance count, where the count of individuals dominated by an individual is used. Pareto Envelope-based Selection Algorithm II (PESA-II) [3] is based on dominance count and uses an archive of non-dominated solutions. All these dominance-based methods evaluate rank serially in a deterministic way (except NPGA), with a quadratic ranking complexity on the population size ($O(mn^2)$ for NSGA-II, where m is the number of objectives and n the population size).

1.1 Related Work

Often a large population is required when solving many objective problems [2, 5] or Genetic Programming (GP) with more than one objective (error minimization and parsimony [1]). In this case, MOEAs become computationally expensive. Parallel function evaluations on GPGPUs can reduce the computation time of algorithms. However, the ranking of a very large population can be a further bottleneck for MOEAs. The first appreciable effort to parallelize MOEAs on GPUs is shown in [26]. In this study, the dominance comparison of NSGA-II was implemented on a GPU, but the sorting of individuals in different fronts is still done on CPU. Recently, an archived-based stochastic ranking evolutionary algorithm (ASREA) [23] has been developed for MOO with an $O(man)$ ranking complexity (where a is the size of an archive that depends on the number of objectives m) which breaks the $O(mn^2)$ complexity while yielding improved results over NSGA-II. Further, ASREA has been modified so that the ranking and function evaluations can be done in parallel on GPGPUs [24]. The algorithm is known as G-ASREA and is discussed later in this chapter. In the following section, an overview of multi-objective optimization is given followed by the concept of dominance and Pareto optimality. The details of ASREA are discussed in Sect. 3 in which its implementation on a GPU is discussed. In Sect. 4, ASREA is compared with benchmarked MOEAs such as NSGA-II and SPEA2 based on statistical indicator analysis. Further, computation time of ASREA on a GPU versus CPU versions of ASREA and NSGA-II is compared. A few salient features of ASREA are discussed in Sect. 5. The chapter is concluded in Sect. 6.

2 Overview of Multi-objective Optimization

A multi-objective optimization problem (MOOP) has more than one objective function to be maximized or minimized. A MOOP may have constraints to satisfy, like single-objective optimization. The general form of a MOOP is given in (1):

$$\begin{aligned} \text{Minimize/maximize: } f_m(x), &\quad m = 1, 2, \ldots, M; \\ \text{Subject to: } g_j(x) \geq 0, &\quad j = 1, 2, \ldots, J; \\ h_k(x) = 0, &\quad k = 1, 2, \ldots, K; \\ x_i^{(L)} \leqslant x_i \leqslant x_i^{(U)}, &\quad i = 1, 2, \ldots, n. \end{aligned} \tag{1}$$

In the above formulation, there are M objective functions $f(x) = (f_1(x), f_2(x), \ldots, f_M(x))^T$. Each objective function can be either minimized or maximized. A solution x is a vector of n decision variables: $x = (x_1, x_2, \ldots, x_n)^T$. The value of each decision variable x_i is restricted to its lower $X_i^{(L)}$ and upper $X_i^{(U)}$ bounds as shown in the last set of constraints. These variable bounds constitute a decision variable space. The MOOP is also subject to J inequality and K equality constraints.

2.1 *Concept of Dominance and Pareto Optimality*

As mentioned earlier, most effective multi-objective optimization algorithms use dominance ranking methods. These methods are based on the concept of dominance where two solutions are compared as given below:

Definition 1. A solution $x^{(1)}$ is said to dominate another solution $x^{(2)}$ if both conditions 1 and 2 are true:

1. The solution $x^{(1)}$ is no worse than $x^{(2)}$ in all objectives.
2. The solution $x^{(1)}$ is strictly better than $x^{(2)}$ in at least one objective.

Based on the above concept, solutions can be categorized as:

1. $x^{(1)}$ dominates $x^{(2)}$.
2. $x^{(1)}$ is dominated by $x^{(2)}$.
3. $x^{(1)}$ and $x^{(2)}$ are non-dominated.

Suppose among a set of solutions P, the non-dominated set of solutions P' are those that are not dominated by any member of the set P. When the set P is the entire search space, the resulting non-dominated set P' is called the *Pareto-optimal set*.

3 Archived-Based Stochastic Ranking Evolutionary Algorithm (ASREA)

ASREA is an attempt to create a more homogeneous MOEA by using a stochastic ranking operator that reduces the usual $O(mn^2)$ complexity to $O(man)$ and also performs this Pareto ranking on GPU cards. Here, m is the number of objectives, a is the size of the archive, and n is the population size. Finding an appropriate size of archive is a difficult task [17] because it depends on many factors such as number of objectives and population size. An archive size can be considered good and optimum if it can accommodate evenly spread non-dominated solutions that can help in the convergence of algorithms. Preliminary tests on various test functions of $m = 2$ and $m = 3$ objectives suggest that an archive size of $(a =)\ 20 \times m$ allows ASREA to obtain good results. However, an appropriate size for an archive in ASREA is a subject for future work.

3.1 Description of ASREA

ASREA starts by evaluating a random initial population (**ini_pop**). Based on their fitness value, a Pareto ranking is assigned to every individual using NSGA-II's ranking operator. An archive is maintained in ASREA and updated according to Algorithm 1 where the distinct non-dominated individuals of **ini_pop** are copied. If the number of non-dominated individuals is larger than the size of the archive, then the diverse non-dominated individuals (according to some clustering techniques) are copied into the archive until it is full. Currently, NSGA-II's crowding distance (CD) operator [7] is used to select the diverse non-dominated individuals, but another clustering technique could be used.

As shown in Fig. 1, **ini_pop** is copied into **parent_pop** and a relatively standard EA loop starts, by checking whether a stopping criterion is met. If this is not the case, a mating pool for crossover is created by repeatedly *randomly* selecting two parents from **parent_pop**. Using the simulated binary crossover (SBX) operator [6], two new individuals are created from randomly selected parents. Polynomial mutation

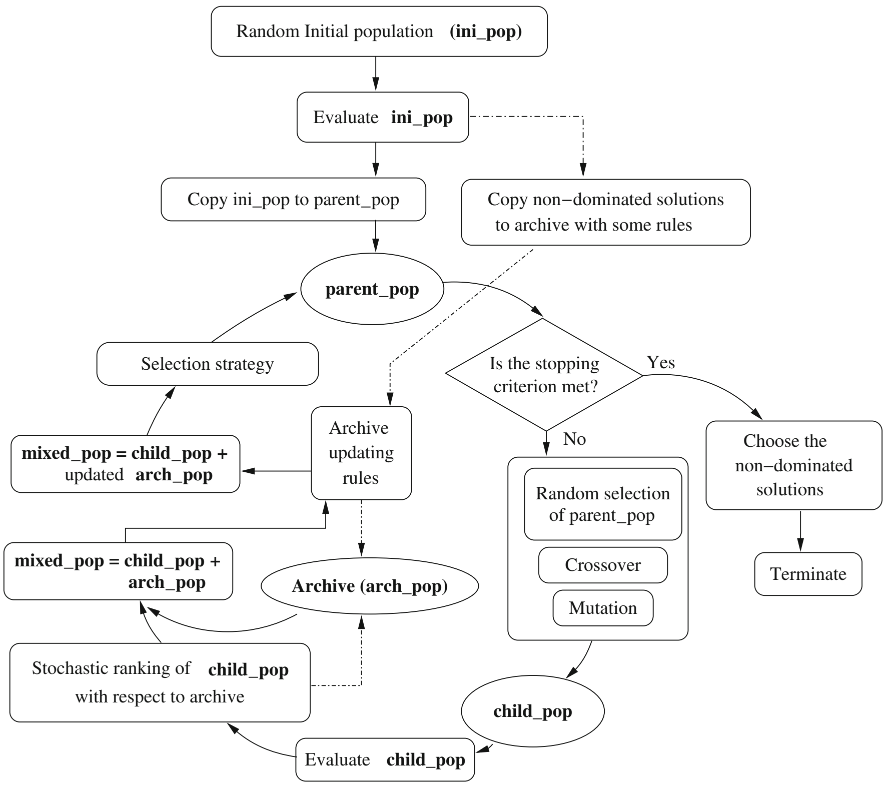

Random Initial population (ini_pop)
Evaluate ini_pop
Copy ini_pop to parent_pop
Copy non−dominated solutions to archive with some rules
parent_pop
Selection strategy
Is the stopping criterion met?
Yes
No
Archive updating rules
Choose the non−dominated solutions
mixed_pop = child_pop + updated arch_pop
Random selection of parent_pop
Crossover
Mutation
Terminate
mixed_pop = child_pop + arch_pop
Archive (arch_pop)
Stochastic ranking of child_pop with respect to archive
child_pop
Evaluate child_pop

Fig. 1 ASREA flowchart

Algorithm 1 Rules for archive update

```
1: if Number of non-dominated solutions ≤ archive size then
2:     Copy distinct non-dominated solutions to archive
3: else
4:     Copy the extreme solutions of non-dominated front to archive and evaluate crowding
       distance (CD [7]) of rest of the non-dominated solutions. Fill the remaining archive with the
       sorted CD-wise individuals in descending order
5: end if
```

is then performed on newly created individuals to create **child_pop**. The next task is to evaluate **child_pop** on the GPU, which is described in the following section.

3.1.1 Function Evaluation on the GPU

Function evaluation (FE) is one of the most time-consuming parts of MOEAs. As FE for every individual can be done independently, it can easily be parallelized on

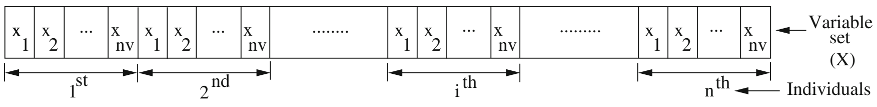


Fig. 2 Single array representation for GPU

GPUs. In order to perform FE, a single float array X of size $v * m$ is allocated in the global memory of the GPU, where v is the number of variables and m is the number of objectives. The variable set of **child_pop** is then copied to X, as shown in Fig. 2. At the same time, a single float array OBJ of size $m * n$ (where n is the child population size) is also allocated in the global memory of the GPU, in order to store the objective function values of **child_pop**. Using a single instruction multiple data (SIMD) GPU kernel, objective functions for all individuals are evaluated on GPU threads.

Once FE for every individual is over, rank is assigned to **child_pop** so that good individuals can propagate to the next generation. So far existing MOEAs such as NSGA-II and SPEA2 have a deterministic Pareto-ranking operator which executes the ranking calculation serially. In this chapter, a stochastic ranking operator is described which is inspired by the EP-tournament operator [8] and close to the working principle of Niched Pareto Genetic Algorithm (NPGA [15]). In both studies, the rank of an individual is assigned by comparing it with only a few individuals randomly selected from the population.

In ASREA, the same concept is used, but the comparison for the rank evaluation of **child_pop** is done with respect to an archive of distinct non-dominated solutions. This feature of ASREA not only reduces the ranking complexity of the algorithm but allows it to stochastically propagate good individuals to the next generation. In this algorithm, the rank of individual (A) of **child_pop** is calculated on a dominance criterion which is given below:

$$\begin{aligned}\mathrm{rank}(A) = 1 + {} & \textit{number of } \boldsymbol{arch\_pop} \textit{ members} \\ & \textit{that dominate } A \end{aligned} \tag{2}$$

Note that in ASREA, a lower rank is better, with best rank = 1. The ranking procedure discussed above is one of the differences in the working principle of ASREA from other MOEAs and specially the archived-based ones. In the following section, an overview of archived-based MOEAs is given.

3.1.2 Overview of Archived-Based MOEAs

The idea of using an off-line population, or so-called *"archive"*, to store good individuals or non-dominated solutions has been explored earlier [22]. This introduces elitism into MOEAs for better convergence. There are some notable archived-based

MOEAs such as PESA [4] and PESA2 [3] in which the archive is filled from the current solution by checking the dominance. If the archive is full, then the selection is done on a squeeze factor of a hyperbox filled with individuals in PESA or region-based hyperbox selection in PESA2. Thereafter, the archive is used for making the mating pool for crossover using a binary tournament selection operator. In SPEA2 [28], the archive is updated by selecting the good individuals from the current child population and previous archive, based on the rank assignment scheme and kth nearest neighbor clustering method. The archive then propagates to the next iteration for mating pool and other genetic operators. The ϵ-dominance and ϵ-Pareto optimality-based archive selection strategy also exist in the literature [19]. Overall, the archive or off-line population discussed in the literature is used for storing the good individuals and propagating them to the next iteration, whereas in ASREA, we use the archive to evaluate the rank of the child population and partially fill the parent population for the next iteration (discussed in the following paragraphs).

3.1.3 Parallel SIMD Ranking on a GPU

The parallel stochastic ranking operator of ASREA [24] suggests that *CR*, another integer array of size n, is used for storing the ranks of **child_pop**. Allocation of *CR* in the global GPU memory can be done at the same time as the *OBJ* array. Suppose the thread processor *idx* computes the rank of a **child_pop** individual using Eq. (2), and then it stores the calculated rank at position *idx* of *CR*.

During the ranking on the GPU, each thread processor also independently keeps track of the domination count of **arch_pop**. For this, another single integer array *AR* of size $a \times n$ is allocated in the global GPU memory before execution. Note that the array is initialized to value 1 because all members of **arch_pop** are rank 1 solutions. When an individual of thread processor *idx* dominates the kth member of **arch_pop**, then the thread processor increments $AR[(a \times idx) + k]$ by 1. This dominance check information is used later, to update ranks in the archive.

After parallel stochastic ranking is finished on the GPU, the objective function values and ranks of **child_pop** are updated by copying *OBJ* and *CR* back to the CPU. The rank of the archive is also modified using array *AR* in the following manner: suppose that the modified rank of the kth member of the archive is evaluated. Then, for $i = 0$ to $n - 1$, the integer value of every $AR[(i \times a) + k]$ is added and finally n is subtracted. If the kth member is still non-dominated, then its modified rank is 1. Otherwise, the rank of the kth member depends on the number of **child_pop** individuals who dominate it.

3.1.4 Replacement Strategy

The next step of ASREA is to update the archive and propagate good individuals to the next generation. First, the ranked **child_pop** and **arch_pop** with modified ranks

Algorithm 2 Selection strategy to fill **parent_pop**

```
1: Copy the extreme solutions of updated arch_pop to the parent population.
2: Fill 20 % of parent_pop from the updated arch_pop in the following way:
3: while 20 % of parent_pop is not filled do
4:     Pick randomly two individuals of updated arch_pop
5:     if Crowding distances are different then
6:         Copy the individual with larger crowding distance into parent_pop
7:     else
8:         Copy any individual randomly
9:     end if
10: end while
11: Fill rest of parent_pop from child_pop in the following way:
12: while rest of parent_pop is not filled do
13:     Pick two individuals randomly from child_pop without replacing them
14:     if Ranks are different then
15:         Copy the individual with smaller rank into parent_pop
16:     else
17:         Copy the individual with larger crowding distance into parent_pop
18:     end if
19: end while
```

are mixed together to form **mixed_pop**. Now, the archive is updated from the set of non-dominated solutions ($rank = 1$) of **mixed_pop** as given in Algorithm 2.

The next generation (new parent population) is also selected from **mixed_pop** according to the strategy discussed in Algorithm 2. Here, 20 % of the new parent population is filled from the updated archive with randomly picked members. The rest of **parent_pop** is filled from **child_pop** individuals using binary tournament selection. The EA loop is then completed and can start again, by checking whether a termination criterion is met (e.g., number of generations) as in Fig. 1.

4 Results and Discussion

4.1 Test Suites and Performance Assessment Tools of MOEAs

Three Zitzler-Deb-Thiele functions (ZDT3, ZDT4 and ZDT6) were chosen to assess the performance of ASREA against two benchmarking MOEAs: NSGA-II and SPEA2. These two algorithms use a deterministic ranking procedure but in different ways: NSGA-II[1] sorts the non-dominated fronts from the child and parent populations, whereas SPEA2 uses the fitness assignment scheme to rank the combined population of archive and child populations.

[1]Source codes for NSGA-II and SPEA2 are available at [16] and [25].

4.2 Performance Assessment Tools

In the field of multi-objective optimization (MOO), the strengths and weaknesses of any algorithm are based on the quality of the evolved solution such as proximity to the reference set and spread and evenness of the non-dominated solutions in the objective space. Several tools are available to assess the performance of MOO algorithms which independently explore different features of the algorithm [10, 18, 30]. We choose two indicators and attainment surface plots for the assessment, whose source codes are available at [25].

R-indicator (I$_{R2}$): [14]

$$I_{R2} = \frac{\sum_{\lambda \in \Lambda} u^{\star}(\lambda, A) - u^{\star}(\lambda, R)}{|\Lambda|} \tag{3}$$

where R is a reference set and $u^{\star}$ is the maximum value reached by the utility function u with weight vector λ on an approximate set A, i.e., $u^{\star} = \max_{z \in A} u_{\lambda}(z)$. The augmented Tchebycheff function is used as the utility function. The second-order R-indicator gives the idea of proximity with respect to the reference set A [18, 30].

Hypervolume indicator (I$_{\bar{H}}$): [29]

The hypervolume indicator I_H measures the hypervolume of that portion of the objective space that is weakly dominated by an approximate set A. This indicator gives the idea of spread quality and has to be maximized. As recommended in the study [18], the difference in values of hypervolume indicator between the approximate set A and the reference set R is calculated in this paper, i.e., $I_{\bar{H}} = I_H(R) - I_H(A)$. A smaller value suggests a good spread [18, 30].

Attainment surface: [11, 12]

An approximate set A is called the $k\,\% - $ *approximate set* of the empirical attainment function (EAD) $\alpha_r(z)$ if it weakly dominates exactly those objective vectors that have been attained in at least k percent of the r runs. Formally, $\forall z \in Z : \alpha_r(z) >= k/100 \leftrightarrow A \preceq \{z\}$ where $\alpha_r(z) = \frac{1}{r}\sum_{i=1}^{r} I(A^i \preceq \{z\})$. A^i is the ith approximation set (run) of the optimizer and $I(.)$ is the indicator function, which evaluates to one if its argument is true and zero if its argument is false.

An attainment surface of a given approximate set A is the union of all tightest goals that are known to be attainable as a result of A. Formally, this is the set $\{z \in \mathfrak{R}^n : A \preceq z \wedge \neg A \prec\prec z\}$ [18].

4.3 Parameters

Performance assessment of ASREA against NSGA-II and SPEA2 is done by performing 25 independent random runs. All three MOEAs are run using identical parameters on the ZDT functions which are given below:

Table 1 Objective function bounds for normalization

Functions→	ZDT3		ZDT4		ZDT6	
Bounds↓	f_1	f_2	f_1	f_2	f_1	f_2
Lower	0.0	−0.773	0.0	0.0	0.281	0.0
Upper	0.873	5.915	1.0	149.672	1.0	9.141

Population	100
Number of generations	500
Crossover probability of individual	0.9
Crossover probability of variable	0.5
Mutation probability of individual	1.0
Mutation probability of variable	1/(no. of variables)
Distribution crossover index	15
Distribution mutation index	20

The performance of evolved non-dominated fronts of the three MOEAs is evaluated at 10^3, 10^4, and 5×10^4 function evaluations (EVAL).

4.4 Performance Assessment of MOEAs

As recommended in [18], minimum and maximum limits on each objective (cf. Table 1) are used to normalize and further scale the approximate sets of MOEAs and the reference set of the given ZDT functions between 1 and 2. On this basis, the reference sets (z^*) of (1, 1) and (2, 2) are used to evaluate I_{R2} values, whereas (2.1, 2.1) is assigned as the reference point (z^+) for calculating the $\mathrm{I}_{\bar{H}}$ values. The suggested range of I_{R2} and $\mathrm{I}_{\bar{H}}$ lies between -1 and $+1$, where -1 is the best and $+1$ is the worst. Interested readers may refer to [18,30] for more details on performance assessment of MOEAs.

Table 2 shows the statistical I_{R2} values (which show the proximity w.r.t. the reference set) of each MOEA over 25 runs (the best values are in bold face). At 5×10^4 EVAL, ASREA shows the best results in the mean, median, best, and worst values of I_{R2} (proximity with the Pareto-optimal (P-O) front). ASREA also gets the best statistical I_{R2} values for ZDT3, ZDT4, and ZDT6 at 10^4 EVAL and for ZDT3 and ZDT6 at 10^3 EVAL.

This suggests that ASREA obtains better results than NSGA-II and SPEA2 on more problems at three stages of EVAL and also ASREA approaches the P-O front faster, before showing the best convergence on the P-O front overall.

One interesting thing to ponder here is that negative values are observed for I_{R2} for the ZDT6 function. This suggests that the evolved non-dominated solutions of the three MOEAs explore regions of the P-O front which are not represented by the limited points of the reference set.

Table 2 R-indicator values of ASREA, NSGA-II, and SPEA2 on ZDT functions (the best values are in bold face)

Functions→		ZDT3			ZDT4			ZDT6		
MOEAs→		ASREA	NSGA-II	SPEA2	ASREA	NSGA-II	SPEA2	ASREA	NSGA-II	SPEA2
10^3	Mean	**0.0719**	0.0886	0.1381	0.0621	0.0763	**0.0394**	**0.1963**	0.2014	0.2248
EVAL	Median	**0.0705**	0.0879	0.1397	0.0601	0.0759	**0.0390**	**0.1982**	0.2026	0.2264
	S.D.	0.0114	**0.0081**	0.0112	0.0162	**0.0095**	0.0152	0.0104	**0.0067**	0.0070
	Best	**0.0566**	0.0718	0.1128	0.0326	0.0450	**0.0150**	**0.1745**	0.1802	0.2098
	Worst	**0.0969**	0.1060	0.1587	0.1084	0.0981	**0.0771**	0.2155	**0.2116**	0.2368
10^4	Mean	**0.0017**	0.0030	0.0944	**0.0011**	0.0018	0.0052	**0.0052**	0.0173	0.0534
EVAL	Median	**0.0008**	0.0016	0.0938	**0.0011**	0.0017	0.0052	**0.0049**	0.0174	0.0525
	S.D.	**0.0035**	0.0048	0.0107	**0.0004**	0.0009	0.0022	**0.0012**	0.0030	0.0121
	Best	**0.0007**	0.0010	0.0747	**0.0003**	0.0005	0.0010	**0.0033**	0.0105	0.0370
	Worst	**0.0187**	0.0192	0.1174	**0.0021**	0.0042	0.0119	**0.0077**	0.0230	0.0835
$5*10^4$	Mean	**0.0006**	**0.0006**	0.0074	**0.0000**	**0.0000**	0.0040	**−0.0008**	**−0.0008**	−0.0006
EVAL	Median	**0.0000**	**0.0000**	0.0053	**0.0000**	**0.0000**	0.0036	**−0.0008**	**−0.0008**	−0.0006
	S.D.	**0.0034**	**0.0034**	0.0066	**0.0000**	**0.0000**	0.0017	**0.0000**	**0.0000**	**0.0000**
	Best	**0.0000**	**0.0000**	0.0010	**0.0000**	**0.0000**	0.0008	**−0.0008**	**−0.0008**	−0.0007
	Worst	**0.0173**	**0.0173**	0.0214	**0.0000**	**0.0000**	0.0099	**−0.0008**	−0.0007	−0.0005

Table 3 shows the statistical $I_{\bar{H}}$ values (which represent the spread of solutions in the objective space) of each MOEA over 25 runs. ASREA displays the best mean, median, best, and worst $I_{\bar{H}}$ values for ZDT4 at 10^4 and for ZDT3 and ZDT6 at 10^3 EVAL.

For the rest of the EVAL stages, ASREA's statistical $I_{\bar{H}}$ values are close to NSGA-II but better than SPEA2 for all ZDT functions except for ZDT3 and ZDT4 at 10^3 EVAL and ZDT6 at 5×10^4 EVAL where SPEA2 shows the best spread of all.

For all ZDT functions, Fig. 3 shows the 0 % attainment surface plots in which:

1. The theoretical Pareto front is in black.
2. The obtained P-O front for NSGA-II is in dark red (dark grey).
3. The obtained P-O front for SPEA2 is in blue (grey).
4. The obtained P-O front for ASREA is in green (light grey).

The plots are overlapped from black to light grey (ASREA) so that the results for ASREA are always visible over SPEA2, NSGA-II, and also over the theoretical front (in black), meaning that ASREA always appears, and the other curves appear where they differ from ASREA.

In the initial stage (10^3 EVAL), ASREA shows the closest proximity with the P-O front and maintains a good spread for all functions, except ZDT4 where SPEA2 has an edge.

In the intermediate stage (10^4 EVAL), ASREA shows the closest proximity for all functions over NSGA-II and SPEA2 and equivalent spread for all functions. In the final stage (5×10^4 EVAL), all three algorithms approximate the P-O front but for SPEA2 on the ZDT4 function (where it was best in the initial stage).

4.5 GPU Simulations Results

The same three ZDT functions are chosen to effectively compare ASREA on a GPU (G-ASREA) with the CPU version of ASREA and NSGA-II on different population sizes. Every MOEA is run 25 times using different seeds, and the average computation time of ranking and function evaluations are shown in Tables 6 and 7. Note that G-ASREA's computation time includes the copying of data from the CPU to the GPU, computation on the GPU, and copying back the data from the GPU to the CPU. All ZDT functions are executed with identical MOEA parameters (cf. Table 4) on an Intel(R) Core(TM)2 Quad CPU Q8200 @ 2.33GHz computer with one of the two GPUs of a GTX295 card.

The algorithms' ranking complexity (such as $O(mn^2)$ or $O(man)$) correspond to the *worst* case, which may not occur in real life. Beyond their theoretical complexity, it is therefore important to assess the complexity of algorithms on real-world problems or at least benchmarks that are supposed to exhibit behaviors encountered in real-world problems.

Table 5 shows the average number of ranking comparisons for NSGA-II, ASREA, and G-ASREA on ZDT functions over different population sizes (100 to 100,000). Depending on the problems and their Pareto fronts, different comparison

Table 3 $I_{\bar{H}}$ indicator values of ASREA, NSGA-II, and SPEA2 on ZDT functions (the best values are in bold face)

Functions→		**ZDT3**			**ZDT4**			**ZDT6**		
MOEAs→		**ASREA**	**NSGA-II**	**SPEA2**	**ASREA**	**NSGA-II**	**SPEA2**	**ASREA**	**NSGA-II**	**SPEA2**
***10^3**	Mean	**0.2561**	0.3149	0.4758	0.1960	0.2498	**0.1484**	**0.6192**	0.6499	0.8009
***EVAL**	Median	**0.2519**	0.3180	0.4758	0.1929	0.2446	**0.1484**	**0.6205**	0.6513	0.7956
	S.D.	0.0234	0.0238	**0.0206**	0.0504	**0.0333**	0.0455	0.0432	**0.0342**	0.0397
	Best	**0.2178**	0.2597	0.4333	0.0972	0.2077	**0.0602**	**0.5200**	0.5384	0.7217
	Worst	**0.3114**	0.3731	0.5307	0.3344	0.3363	**0.2488**	**0.7082**	0.7094	0.8531
***10^4**	Mean	**0.0070**	0.0072	0.2945	**0.0036**	0.0059	0.0162	**0.0137**	0.0416	0.1471
***EVAL**	Median	**0.0062**	0.0071	0.2906	**0.0036**	0.0056	0.0161	**0.0138**	0.0409	0.1408
	S.D.	**0.0020**	**0.0020**	0.0254	**0.0015**	0.0030	0.0064	**0.0023**	0.0063	0.0385
	Best	0.0052	**0.0045**	0.2412	**0.0011**	0.0017	0.0030	**0.0094**	0.0282	0.0959
	Worst	0.0135	**0.0133**	0.3564	**0.0070**	0.0130	0.0355	**0.0197**	0.0547	0.2117
***5*10^4**	Mean	0.0026	**0.0006**	0.0089	0.0002	**0.0000**	0.0132	−0.0012	**−0.0032**	−0.0029
***EVAL**	Median	0.0023	**0.0003**	0.0084	0.0001	**0.0000**	0.0119	−0.0013	**−0.0032**	−0.0029
	S.D.	**0.0012**	0.0013	0.003	**0.0000**	**0.0000**	0.0054	0.0004	**0.0000**	**0.0000**
	Best	0.0016	**0.0003**	0.0045	0.0001	**0.0000**	0.0027	−0.0017	**−0.0032**	−0.0030
	Worst	0.0083	**0.0071**	0.0161	0.0004	**0.00004**	0.0320	0.0000	**−0.0031**	−0.0027

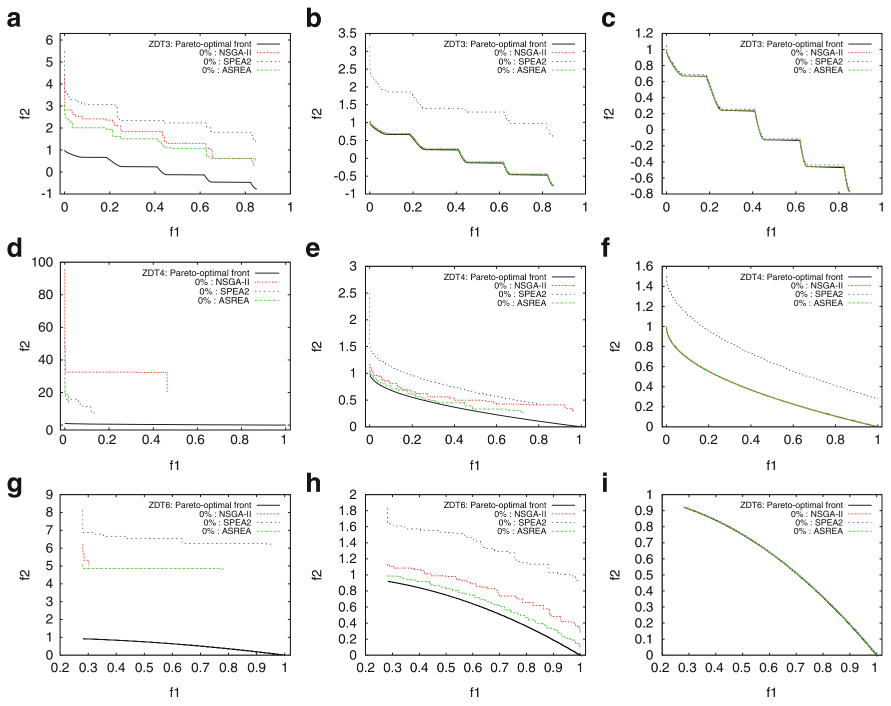


Fig. 3 Attainment surface plots of ASREA, NSGA-II, and SPEA2 for ZDT functions. (**a**) ZDT3 at 1,000 EVAL, (**b**) ZDT3 at 10,000 EVAL, (**c**) ZDT3 at 50,000 EVAL, (**d**) ZDT4 at 1,000 EVAL, (**e**) ZDT4 at 10,000 EVAL, (**f**) ZDT4 at 50,000 EVAL, (**g**) ZDT6 at 1,000 EVAL, (**h**) ZDT6 at 10,000 EVAL, (**i**) ZDT6 at 50,000 EVAL

Table 4 Common parameters for all tested algorithms

No. of generations	100		
Individual Xover prob	0.9	Variable Xover prob	0.5
Individual Mut prob	1.0	Variable Mut prob	1/(no. of var)
Distrib crossover index	15	Distrib mutation index	20

counts are observed for the same algorithm (522.10^6 comparisons for NSGA-II with 10,000 individuals on ZDT3, compared to $2,395.10^6$ comparisons for the same algorithm and the same population size on ZDT6).

Significant differences can be seen with ASREA and G-ASREA over NSGA-II, which increase drastically with larger populations, confirming the difference shown by their respective $O(man)$ and $O(mn^2)$ complexities (small differences are observed between ASREA and G-ASREA because the algorithms were run with different seeds).

ASREA and G-ASREA were further tested for $100,000$ individuals (a typical genetic programming population size) where they show a comparable number

Table 5 Average number of ranking comparisons for different population sizes

Problems		ZDT3			ZDT4			ZDT6		
Pop↓	MOEAs→	NSGA-II	ASREA	G-ASREA	NSGA-II	ASREA	G-ASREA	NSGA-II	ASREA	G-ASREA
100		1.19 E6	198,073	197,940	729,419	181,269	179,710	753,581	191,993	189,120
1,000		79.8 E6	2.15 E6	2.15 E6	43.8 E6	1.98 E6	1.98 E6	45.3 E6	2.06 E6	2.05 E6
10,000		522 E6	25.5 E6	25.5 E6	2382 E6	22.9 E6	22.9 E6	2395 E6	23.5 E6	23.1 E6
100,000		–	392 E6	391 E6	–	330 E6	330 E6	–	372 E6	337 E6

Table 6 Ranking comparison time (in seconds) and speedup ratio of MOEAs over wide range of populations

Problem		ZDT3			Speedup ratio		
Pop↓	MOEAs→	NSGA-II (N)	ASREA (A)	G-ASREA (G)	N/A ratio	N/G ratio	A/G ratio
100		0.000574	0.000103	0.000121	5.5728	4.7438	0.8512
1,000		0.038939	0.001014	0.000162	38.4013	240.3641	6.2592
10,000		4.538463	0.009968	0.000834	455.3032	5441.8021	11.9520
100,000		–	0.098616	0.007113	–	–	13.8641
1,000,000		ZDT4	0.992453	0.067140	–	–	14.7818
100		0.000374	0.000094	0.000115	3.9787	3.2521	0.8173
1,000		0.020379	0.000952	0.000160	21.4065	127.3687	5.9500
10000		1.159553	0.009277	0.000866	124.9922	1338.9757	10.7124
100,000		–	0.092044	0.006947	–	–	13.2492
1,000,000		ZDT6	0.920826	0.066112	–	–	13.9283
100		0.000372	0.000105	0.000120	3.5428	3.1000	0.8750
1,000		0.021340	0.001071	0.000161	19.9253	132.5465	6.6521
10,000		1.161053	0.009821	0.000829	118.2214	1400.5464	11.8468
100,000		–	0.095291	0.006969	–	–	13.6735
1,000,000		–	0.947676	0.066241	–	–	14.3064

Table 7 Function evaluation time (in seconds) of serial and GPU ASREA

Problems		ZDT6		
Pop↓	MOEAs→	ASREA	G-ASREA	Speedup ratio
100		0.00053	0.000119	0.4453
1000		0.00521	0.000161	3.2360
10000		0.004798	0.000816	5.8821
100000		0.04728	0.006921	6.8319
1000000		0.4785	0.06526	7.3322

of ranking comparisons than NSGA-II for $10,000$. NSGA-II was not tested for $100,000$ individuals over 25 different runs because it took around 20 h for just one single run on a ZDT3 function.

ASREA algorithms are not only frugal on comparisons, but G-ASREA can parallelize the ranking (and evaluation) over the GPU processors, allowing for the speedups shown in Table 6, in which the average time of ranking comparison per generation of NSGA-II, ASREA, and G-ASREA is given. The N/A and N/G ratios represent speedup of ASREA and G-ASREA over NSGA-II. N/A speedup ranges from 3.54 to 455.3, whereas N/G speedup ranges between 3.1 and $5,441$ for 10,000 individuals.

For a population of $10,000$ to one million[2], G-ASREA shows a speedup between 10 and 15 over CPU ASREA (cf. Table 6). The study [26] also showed the same range of speedup in ranking comparison for dominance check on ZDT functions. However, the speedup is limited to this range for two-objective problems because the dominance check of G-ASREA concerns only parallelization of ranking comparisons on the GPU. Nevertheless, if the number of comparisons increases for many-objective optimization, then the speedup for dominance check can increase further.

G-ASREA's ranking operator offers a twofold advantage: first, the speedup comes from the smaller ranking complexity over NSGA-II. Then, another advantage comes from being able to perform ranking comparisons on the GPU. Function evaluation time can also benefit from being run on the GPU. However, this aspect was already studied in several papers [20, 26], so only the computation time for ZDT6 is shown in Table 7. The speedup is not impressive because the ZDT6 function is not computationally intensive (speedups of up to $\times 250$ have been obtained on GP function evaluations in [21]).

The study presented in this chapter is not so much concerned with solving the ZDT benchmarks (this can be done with hundreds of individuals only [23]) as to study and minimize the bottleneck induced by the ranking operator that may limit the usage of very large populations in multi-objective optimization problems.

[2]Note that only one run is performed for 1 million individuals on each ZDT function.

5 Salient Features

The first important feature of G-ASREA comes from its reduced ranking computational complexity $O(man)$ and its implementation on GPGPU for assigning rank to the population by dominance check.

A second important feature of G-ASREA is its inherent ability to preserve diversity in the population. This comes when the not-so-good individuals of **child_pop** may nevertheless obtain a good rank (including rank = 1) because they are only compared with an archive of limited size. Had the ranking method been deterministic, then these not-so-good individuals would not have made it into the next generation and it may have been necessary to implement a diversity-preserving scheme in order to avoid premature convergence. This is why ASREA's ranking procedure is *stochastic*.

Another feature of G-ASREA is the drastic but subtle selection strategy that is used to propagate good individuals to the next iteration. Finally, G-ASREA also uses the GPU for function evaluation.

6 Conclusions

The advent of massively parallel GPGPU cards allows one to parallelize the evaluation of very large populations in order to solve complex optimization problems. In MOEAs, the bottleneck then becomes the multi-objective ranking stage. In this chapter, a GPGPU-compatible stochastic ranking operator is discussed that not only requires fewer comparisons for dominance check but that can parallelize on the GPU card to assign the population ranking. G-ASREA's benefit in ranking complexity is verified as speedups of $\approx \times 5000$ are observed over dominance sorting of NSGA-II on a CPU on a population of 10,000 individuals. However, the speedup was limited to $\approx \times 15$ over ASREA for a large population on two-objective problems because the dominance check of G-ASREA concerns only parallelization of ranking comparisons on the GPU.

Future research work will address G-ASREA's clustering method, in order to make it GPGPU compatible so as to obtain further speedups over ASREA and also to solve many-objective problems for which the current crowding distance clustering operator is not very effective. Further investigation is required on the size of archive for many-objective problems.

Appendix: Test Function

Problem definition for ZDT functions [27] (Table 8):
Minimize $\Gamma(\mathbf{x}) = (f_1(x_1), f_2(\mathbf{x}))$
subject to $f_2(\mathbf{x}) = g(x_2, \ldots., x_m)h(f_1(x_1), g(x_2, \ldots., x_m))$
where $\mathbf{x} = (x_1, \ldots, x_m)$

Table 8 ZDT functions

Functions	Properties
ZDT3: $f_1(x_1) = x_1$, $g(x_2, \ldots, x_m) = 1 + 9.\sum_{i=2}^{m} x_i/(m-1)$, $h(f_1, g) = 1 - \sqrt{f_1/g} - (f_1/g)\sin(10\pi f_1)$, where $m = 30$, and $x_i \in [0, 1]$	Discontinuous convex P-O front
ZDT4: $f_1(x_1) = x_1$, $g(x_2, \ldots, x_m) = 1 + 10(m-1) + \sum_{i=2}^{m}(x_i^2 - 10\cos(4\pi x_i))$, $h(f_1, g) = 1 - \sqrt{f_1/g}$, where $m = 10$, $x_1 \in [0, 1]$, and $x_2, \ldots, x_m \in [-5, 5]$	21^9 local optimal P-O fronts
ZDT6: $f_1(x_1) = 1 - \exp(-4x_1)\sin^6(6\pi x_1)$, $g(x_2, \ldots, x_m) = 1 + 9.(\sum_{i=2}^{m} x_i/(m-1))^{0.25}$, $h(f_1, g) = 1 - (f_1/g)^2$, where $m = 10$, and $x_i \in [0, 1]$	1. Nonuniform distribution of P-O solutions. 2. Density of solutions is lower near the P-O front and highest away from the front

References

1. Baumes, L., Blansch, A., Serna, P., Tchougang, A., Lachiche, N., Collet, P. Corma, A.: Using genetic programming for an advanced performance assessment of industrially relevant heterogeneous catalysts. Mater. Manuf. Process. **24**(3), (March 2009)
2. Coello, C.A.C., Lamont, G.B., Veldhuizen, D.A.V.: Evolutionary Algorithms for Solving Multi-objective Problems. Springer, New York (2007)
3. Corne, D.W., Jerram, N.R., Knowles, J.D., Oates, M.J.: PESA-II: Region-based selection in evolutionary multiobjective optimization. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO2001), pp. 283–290. Morgan Kaufmann Publishers, San Francisco, CA (2001)
4. Corne, D.W., Knowles, J.D., Oates, M.J.: The Pareto envelope-based selection algorithm for multiobjective optimization. In Proceedings of the Parallel Problem Solving from Nature VI Conference, pp. 839–848. Springer, Paris (2000)
5. Deb, K.: Multi-objective Optimization Using Evolutionary Algorithms, Wiley, Chichester, UK (2001)
6. Deb, K., Agrawal, R.B.: Simulated binary crossover for continuous search space. Complex Systems. **9**(2), 115–148 (1995)
7. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182 –197 (April 2002)
8. Fogel, L.J., Owens, A.J., Walsh, M.J.: Artificial Intelligence Through Simulated Evolution. Wiley, New York (1966)
9. Fonseca, C.M., Fleming, P.J.: Genetic algorithms for multi-objective optimization: Formulation, discussion, and generalization. In: Proceedings of the Fifth International Conference on Genetic Algorithms, pp. 416–423. Morgan Kaufmann, San Mateo, CA (1993)
10. Fonseca, C.M., Fleming, P.J.: On the performance assessment and comparison of stochastic multiobjective optimizers. In: Proceedings of Parallel Problem Solving from Nature IV (PPSN-IV), pp. 584–593. Springer, Berlin (1996)
11. Fonseca, C.M., Fonseca, V.G., Paquete, L.: Exploring the performance of stochastic multiobjective optimizers with the second-order attainment functions. In: Proceedings of the Third Evolutionary Multi-criterion Optimization (EMO-05) Conference, pp. 250–264. Springer, Berlin (2005)
12. Fonseca, V.G., Fonseca, C.M., Hall, A.O.: Inferential performance assessment of stochastic optimizers and the attainment function. In: Proceedings of the First Evolutionary Multi-criterion Optimization (EMO-01) Conference, pp. 213–225. Springer, Berlin (2001)

13. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, New York (1989)
14. Hansen, M.P., Jaszkiewicz, A.: Evaluating the Quality of Approximations to the Non-dominated Set. Imm-rep-1998-7, Technical University of Denmark, 1998
15. Horn, J., Nafploitis, N., Goldberg, D.E.: A niched Pareto genetic algorithm for multi-objective optimization. In: Proceedings of the First IEEE Conference on Evolutionary Computation, Genetic Algorithms Laboratory, Illinois University, Urbana, IL, USA pp. 82–87, 1994
16. KanGAL. NSGA-II in C with gnuplot (real + binary + constraint handling): Revision 1.1. http://www.iitk.ac.in/kangal/codes.shtml, July 26 2013
17. Knowles, J., Corne, D.: Properties of an adaptive archiving algorithm for storing nondominated vectors. IEEE Trans. Evol. Comput. **7**(2), 100–116 (2003)
18. Knowles, J., Thiele, L. Zitzler, E.: A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers. TIK Report 214, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, February 2006
19. Laumanns, M., Thiele, L., Deb, K., Zitzler, E.: Archiving with Guaranteed Convergence and Diversity in Multi-objective Optimization. In: Genetic and Evolutionary Computation Conference (GECCO 2002), pp. 439–447, Morgan Kaufmann Publishers, New York, NY, USA (July 2002)
20. Maitre, O., Baumes, L.A., Lachiche, N., Corma, A., Collet, P.: Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA. In: GECCO '09: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, pp. 1403–1410, ACM, New York, NY, USA, 2009
21. Maitre, O., Querry, S., Lachiche, N., Collet, P.: EASEA parallelization of tree-based genetic programming. In: IEEE Congress on Evolutionary Computation (CEC 2010), University of Strasbourg, Illkirch, France, 2010
22. Parks, G.T., Miller, I.: Selective breeding in a multiobjective genetic algorithm. In: Eiben, A.E., Schoenauer, M., Schwefel, H.P. (eds.) Proceedings of the Parallel Problem Solving from Nature V (PPSN-V), pp. 250–259. Springer, Amsterdam, Netherlands (1998)
23. Sharma, D., Collet, P.: An archived-based stochastic ranking evolutionary algorithm (ASREA) for multi-objective optimization. In: GECCO '10: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, pp. 479–486. ACM, New York, NY, USA, 2010
24. Sharma, D., Collet, P.: GPGPU-Compatible archive based stochastic ranking evolutionary algorithm (G-ASREA) for multi-objective optimization. In: Schaefer, R., Cotta, C., Kolodziej, J., Rudolph, G. (eds.) PPSN (2). Lecture Notes in Computer Science, vol. 6239, pp. 111–120. Springer, Berlin (2010)
25. TIK. A platform and programming language independent interface for search algorithms. http://www.tik.ee.ethz.ch/pisa/. July 26 2013
26. Wong, M.L.: Parallel multi-objective evolutionary algorithms on graphics processing units. In: GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference, pp. 2515–2522. ACM, New York, NY, USA, 2009
27. Zitzler, E., Deb, K., Thiele, L.: Comparison of multiobjective evolutionary algorithms: Empirical results. Evol. Comput. **8**(2), 125–148 (2000)
28. Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the strength Pareto evolutionary algorithm for multiobjective optimization. In: Giannakoglou, K. et al. (eds.) Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems (EUROGEN 2001), pp. 95–100. International Center for Numerical Methods in Engineering (CIMNE), 2002
29. Zitzler, E., Thiele, L.: Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. IEEE Trans. Evol. Comput. **3**(4), 257–271 (1999)
30. Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C.M., Grunert da Fonseca, V.: Performance assessment of multiobjective optimizers: an analysis and review. IEEE Trans. Evol. Comput. **7**(2), 117–132 (2003)

Data Mining Using Parallel Multi-objective Evolutionary Algorithms on Graphics Processing Units

Man Leung Wong and Geng Cui

Abstract An important and challenging data mining application in marketing is to learn models for predicting potential customers who contribute large profits to a company under resource constraints. In this chapter, we first formulate this learning problem as a constrained optimization problem and then convert it to an unconstrained multi-objective optimization problem (MOP), which can be handled by some multi-objective evolutionary algorithms (MOEAs). However, MOEAs may execute for a long time for the MOP, because several evaluations must be performed. A promising approach to overcome this limitation is to parallelize these algorithms. Thus we propose a parallel MOEA on consumer-level graphics processing units (GPU) to tackle the MOP. We perform experiments on a real-life direct marketing problem to compare the proposed method with the parallel hybrid genetic algorithm, the DMAX approach, and a sequential MOEA. It is observed that the proposed method is much more effective and efficient than the other approaches.

1 Introduction

How to improve marketing productivity or the return on marketing investment under resource constraints is one of the most challenging issues facing marketing professionals and researchers. The issue seems to be more pressing in hard economic times and given the increasing emphasis on customer relationship management—containing cost and channeling precious marketing resources

M.L. Wong (✉)
Department of Computing and Decision Sciences, Lingnan University, Tuen Mun, Hong Kong
e-mail: mlwong@ln.edu.hk

G. Cui
Department of Marketing and International Business, Lingnan University, Tuen Mun, Hong Kong
e-mail: gcui@ln.edu.hk

to the high-value customers who contribute greater profit to a company. Such situations include (1) upgrading customers—how to provide sizable incentives to the customers who are the most likely to upgrade and contribute greater profit over the long run? (2) modeling customer churn or retention—how to identify and prevent the most valuable customers from switching to a competitor? and (3) predicting loan default—how to predict the small minority who default on their large loans or credit card bills? This problem is particularly acute in direct marketing operations that typically have a fixed budget to target, from the vast list of customers in a company's database, those customers who are the most likely to respond to a marketing campaign and purchase greater amounts.

Most marketing activities, as espoused by marketing scholars and practitioners, are targeted marketing in nature—to reach customers who are the most responsive to marketing activities. Until recently, statistical methods such as regression and discriminant analysis have dominated the modeling of consumer responses to marketing activities. These methods, however, suffer from two shortcomings. First, methods based on OLS regression (i.e., mean regression) survey the entire population and focus on the average customer in estimating parameters. Segmentation methods, such as discriminant analyses, are not informative of their marketing responses. Thus, these methods by design are not entirely compatible with the objectives of targeted marketing. Second, researchers have so far focused on modeling either consumer responses or purchase quantity. Few models jointly consider consumers' responses and the revenue/profit that they generate.

These problems are particularly acute in modeling consumer responses to direct marketing and result in suboptimal performance of marketing campaigns. Conventional methods generate the predicted purchase probabilities for the entire population and do not focus on the top portion of the population, e.g., the top 20 % most attractive customers. This constraint is crucial as most firms have a limited marketing budget and can only target the most attractive customers. Thus, improving the accuracy of predicting purchase and potential sales or profit for these customers is crucial for augmenting the performance of targeted marketing. This is an urgent research problem given the increasing emphasis on customer relationship management and differentiating customers based on their profitability. Predicting loan default, customer churning, and service intervention represent other situations where resources are limited, budget constraints need to be considered, and targeted efforts are required.

To improve the accuracy of customer selection for targeted marketing, we formulate this problem as a constrained optimization problem. Recently, several researchers suggested using multi-objective evolutionary algorithms (MOEAs) to solve constrained optimization problems [24, 26].

However, MOEAs may execute for a long time for some difficult problems, because several objective value evaluations must be performed on huge datasets containing information about customers. Moreover, the non-dominance-checking and the non-dominated-selection procedures are also time-consuming. A promising approach to overcome this limitation is to parallelize these algorithms. In this chapter, we propose implementing a parallel MOEA for constrained optimization within the CUDA (Compute Unified Device Architecture) environment on an nVidia graphics processing unit (GPU). We perform experiments on a real-life direct marketing problem to compare our parallel MOEA with a parallel hybrid genetic algorithm (HGA) [29], the DMAX approach [1], and a sequential MOEA. It is observed that the parallel MOEA is much more effective and efficient than the other approaches. Since consumer-level GPUs are available in omnipresent personal computers and these computers are easy to use and manage, more people will be able to use our parallel MOEA to handle different real-life targeted marketing problems.

In the rest of the chapter, we first give an overview of constrained optimization for direct marketing, MOEAs, and GPU. In Sect. 3, our parallel MOEA for constrained optimization on GPU is discussed. The experiments and the results are reported in Sect. 4. In Sect. 5, conclusions and possible future research are discussed.

2 Overview

2.1 Constrained Optimization for Direct Marketing

Recent emphasis on customer relationship management require marketers to focus on the high-profit customers as in the 20/80 principle: 20 % of the customers account for 80 % profit of a firm. Thus, another purpose of direct marketing models is to predict the amount of purchase or profit from the buyers. However, the distribution of customer sales and profit data is also highly skewed with a very long tail, indicating a concentration of profit among a small group of customers [20]. In empirical studies of profit forecasting, the skewed distribution of profit data also creates problem for researchers. Given the limited and often a fixed marketing budget, the profit maximization approach to customer selection, which include only those customers with an expected marginal profit, is often not realistic [2]. Thus, the ultimate goal of target customer selection is to identify those customers who are the most likely to respond as well as contribute a larger amount of revenue or profit. Overall, unbalanced class distribution and skewed profit data, i.e., the small number of buyers and that of high-profit customers remain significant challenges in direct marketing forecasting [6]. Even a small percentage of improvement in the predictive accuracy in terms of customer purchase probability and profit can mean tremendous cost-savings and augment profit for direct marketers.

To date, only a few researchers have considered treating direct marketing forecasting as a problem of constrained optimization. Bhattacharyya [1] applied a genetic algorithm (GA) to a linear model that maximizes profitability at a given depth of file using the frontier analysis method. The DMAX model was built for a 10 %-of-file mailing. The decile analysis indicates the model has good performance as well as a very good representation of the data as evidenced by the total profit at the top decile. However, a closer look at the decile analysis reveals the model may not be as good as initially believed. The total profit shows unstable performance through the deciles, i.e., profit values do not decrease steadily through the deciles. This unstable performance, which is characterized by major "jumps" in several deciles, indicates the model is inadequately representing the data distribution and may not be reliable for decision support. The probable cause for this "lack-of-fit" is the violation of the assumption of normal distribution in the dependent variable.

Optimization of the classifier does not necessarily lead to maximization of return on investment (ROI), since maximization of the true-positive rate is often different from the maximization of sales or profit, which determines the ROI under a fixed budget constraint. To solve this problem, Yan and Baldasare [30] proposed an algorithm that uses gradient descent and the sigmoid function to maximize the monetary value under the budget constraint in an attempt to maximize the ROI. By comparison with several classification, regression, and ranking algorithms, they find that their algorithm may result in substantial improvement of the ROI.

2.2 Multi-objective Evolutionary Algorithms

We focus on, without loss of generality, minimization multi-objective problems in this chapter. However, either by using the *duality* principle [7] or by simple modifications to the domination definitions, these definitions and algorithms can be used to handle maximization or combined minimization and maximization problems.

For a multi-objective function Γ from $X(\subseteq \Re^N)$ to a finite set $Y(\subset \Re^m, m \geq 2)$, a decision vector $\mathbf{x}^* = [x^*(1), x^*(2), \cdots, x^*(N)]^T$ is *Pareto optimal* if and only if for any other decision vector $\mathbf{x} \in X$, their objective vectors $\mathbf{y}^* = \Gamma(\mathbf{x}^*) = [y^*(1), y^*(2), \cdots, y^*(m)]^T$ and $\mathbf{y} = \Gamma(\mathbf{x})$ holds either

$$y^*(i) \leq y(i) \text{ for any objective } i \ (1 \leq i \leq m)$$

or there exist two different objectives i, j such that

$$\left(y^*(i) < y(i)\right) \wedge \left(y(j)^* > y(j)\right).$$

Thus, for a Pareto optimal decision vector $\mathbf{x}^*$, there exists no decision vector $\mathbf{x}$ which would decrease some objective values without causing a simultaneous increase in at least one other objective. These Pareto optimal decision vectors are good trade-offs for the multi-objective optimization problem (MOP). For finding these vectors, dominance in the objective space plays an important role. An objective vector $\mathbf{y}_1 = \Gamma(\mathbf{x}_1) = [y_1(1), y_1(2), \cdots, y_1(m)]^T$ *dominates* another objective vector $\mathbf{y}_2 = \Gamma(\mathbf{x}_2)$ if and only if the former is partially less than the latter in each objective, i.e.,

$$\begin{cases} y_1(i) \leq y_2(i), & \forall i \in \{1, \cdots, m\} \\ y_1(j) < y_2(j), & \exists j \in \{1, \cdots, m\}. \end{cases} \tag{1}$$

It is denoted as $\mathbf{y}_1 \prec \mathbf{y}_2$. For notational convenience, $\mathbf{y}_1$ is defined to be *incomparable* with $\mathbf{y}_2$ if $\neg(\mathbf{y}_1 \prec \mathbf{y}_2 \vee \mathbf{y}_2 \prec \mathbf{y}_1 \vee \mathbf{y}_1 = \mathbf{y}_2)$. It is denoted as $\mathbf{y}_1 \sim \mathbf{y}_2$. We also denote $\neg\,(\mathbf{y}_1 \prec \mathbf{y}_2)$ as $\mathbf{y}_1 \not\prec \mathbf{y}_2$. That means $(\mathbf{y}_1 = \mathbf{y}_2 \vee \mathbf{y}_2 \prec \mathbf{y}_1 \vee \mathbf{y}_1 \sim \mathbf{y}_2)$.

Given the set of objective vectors Y, its *Pareto front* Y^* contains all vectors $\mathbf{y}^* \in Y$ that are not dominated by any other vector $\mathbf{y} \in Y$. That is, $Y^* = \{\mathbf{y}^* \in Y | \nexists \mathbf{y} \in Y, \mathbf{y} \prec \mathbf{y}^*\}$. We call its subset a *Pareto optimal set*. Each $\mathbf{y}^* \in Y^*$ is *Pareto optimal* or *non-dominated*. A Pareto optimal solution reaches a good trade-off among these conflicting objectives: one objective cannot be improved without worsening any other objective.

In the general case, it is impossible to find an analytic expression of the Pareto front. The normal procedure to find the Pareto front is to compute the objective values of decision vectors sufficiently enough and then determine the Pareto optimal vectors to form the Pareto front [4]. However, for many MOPs, the Pareto front Y^* is of substantial size, and the determination of Y^* is computationally prohibitive. Thus, the whole Pareto front Y^* is usually difficult to get and maintain. Furthermore, it is questionable to regard the whole Pareto front as an ideal answer [9, 19]. The value of presenting such a large set of solutions to a decision maker is also doubtful in the context of decision support. Usually, a set of representative Pareto optimal solutions are expected. Finally, in a solution set of bounded size, preference information could be used to steer the process to certain parts of the search space. Therefore, all practical implementations of MOEAs have maintained a bounded archive of best (non-dominated) solutions found so far [17].

A number of elitist MOEAs have been developed to address diversity of the archived solutions. The diversity exploitation mechanisms include mating restriction, fitness sharing (NPGA [13]), clustering (SPEA [35], SPEA2 [34]), nearest neighbor distance or crowding distance (NSGA-II [8]), crowding count (PAES [16], PESA-II [5], DMOEA [32]), or some preselection operators [7]. Most of them are quite successful, but they cannot ensure convergence to Pareto optimal sets.

2.3 Graphics Processing Units

The demand from the multimedia and games industries for accelerating 3D rendering has driven several graphics hardware companies devoted to the development of high-performance parallel graphics accelerator. This resulted in the birth of GPU, which handles the rendering requests using 3D graphics application programming interface (API). The whole pipeline consists of the transformation, texturing, illumination, and rasterization to the framebuffer. The need for cinematic rendering from the games industry further raised the need for programmability of the rendering process. Starting from the recent generation of GPUs launched in 2001 (including nVidia GeforceFX series and ATI Radeon 9800 and above), developers can write their own C-like programs, which are called *shaders*, on GPU by using a shading language. Due to the wide availability, programmability, and high-performance of these consumer-level GPUs, they are also cost-effective for many general purpose computations.

The shading languages are high-level programming languages and closely resemble to C. Most mathematical functions available in C are supported by the shading languages. Moreover, high-precision floating-point computation is supported on some GPUs. Hence, GPU can be utilized for speeding up the time-consuming computation in evolutionary algorithms (EAs). Since the shading languages were originally designed for applications in computer graphics, researchers should have knowledge about computer graphics, in order to use the languages effectively to develop different EAs.

Recently, the CUDA technology is developed [21]. It allows researchers to implement their GPU-based applications more easily. In CUDA, multiple threads can execute the same kernel program in parallel. Threads can access data from multiple memory spaces including the local, shared, global, constant, and texture memory. Because of the Single Instruction Multiple Thread (SIMT) architecture of GPU, certain limitations are imposed. Data-dependent for loop is not efficient because each thread may perform different number of iterations. Moreover, if-then-else construct is also inefficient, as the GPU will execute both true and false statements in order to comply with the SIMT design.

A number of GPU-based evolutionary programming (EP) [10,28], GAs [29], and genetic programming (GP) [3, 12, 18, 27] have been proposed by researchers.

3 Parallel MOEA for Constrained Optimization on Graphics Processing Units

We propose a learning algorithm to handle the constrained optimization and cost-sensitive problems. Let $E = \{e_1, e_2, \cdots, e_K\}$ be the set of K potential customers and $c(e_i)$, $1 \leq i \leq K$, be the amount of money spent by the customer e_i. Assume that $r\,\%$ of the customers will be solicited. If we can learn a regression function to predict their expected profits or induce a ranking function to arrange the cases in descending order according to their expected profits, we can solicit the first $\lceil K * r\,\% \rceil$ cases to achieve the goal of maximizing the total expected profits of the solicited cases. However, Yan and Baldasare [30] pointed out that this approach tackles an unnecessarily difficult problem and often results in poor performance.

On the other hand, we can learn a scoring function f that divides the K cases into two classes: U and L. The sizes of U and L should be $|U|$ and $|L|$, respectively. Consider a case e_i in U; its $f(e_i)$ must be greater than the scoring values of all cases in L. Moreover, the total of the expected profits of all cases e_i in U is maximized. In other words, we can formulate the learning problem as the following constrained optimization problem:

Find a scoring function f that

$$\max \left\{ \sum_{e_i \in U} c(e_i) \right\}, U = \left\{ e_i \in E \mid \nexists e_j \in L[f(e_i) \leq f(e_j)] \right\} \tag{2}$$

subject to

$$\begin{cases} |U| = \lceil K * r\,\% \rceil \\ |L| = K - \lceil K * r\,\% \rceil. \end{cases} \tag{3}$$

Since the orderings of all cases in U and all cases in L are insignificant to our objective, it would be easier to learn the scoring function that achieves an optimal partial ranking (ordering) of cases. Although the problem of learning the scoring function is easier, the procedure of finding U and L is still time-consuming because it is necessary to find the $(100 - r)$ percentile of E. Thus, we simplify the above constrained optimization problem to the following constrained optimization problem:

Find a scoring function f and a threshold τ that

$$\max \left\{ \sum_{e_i \in U} c(e_i) \right\}, U = \left\{ e_i \in E | f(e_i) > \tau \right\} \tag{4}$$

subject to

$$|U| = \lceil K * r\,\% \rceil. \tag{5}$$

We can convert the constrained optimization problem to an unconstrained MOP with two objectives [26],

$$\max\{\sum_{e_i \in U} c(e_i)\}, U = \{e_i \in E | f(e_i) > \tau\}, \tag{6}$$

$$\min\{\text{maximum}(0, |U| - \lceil K * r\,\% \rceil)\}. \tag{7}$$

By limiting f to be a linear function, a MOEA can be used to find the parameters of the scoring function f and the value of τ. We apply a parallel MOEA on GPU that finds a set of non-dominated solutions for a multi-objective function Γ that takes a vector $\mathbf{x}$ containing the parameters of the scoring function f as well as the value of τ and returns an objective vector $\mathbf{y}$, where $\mathbf{x} = [x(1), x(2), \cdots, x(N)]^T$, $\mathbf{y} = [y(1), y(2), \cdots, y(m)]^T$, N is 1 plus the number of the parameters of f, and m is the number of objectives. The algorithm is given in Fig. 1.

In the algorithm given in Fig. 1, $\mathbf{x_i}$ is a vector of variables evolving and $\boldsymbol{\eta}_\mathbf{i}$ controls the vigorousness of mutation of $\mathbf{x_i}$. Firstly, an initial population is generated and the objective values of the individuals in the initial population are calculated by using the multi-objective function Γ.

Next, the rankings and the crowding distances of the individuals are found. All non-dominated individuals will be assigned a ranking of 0. The crowding distance of a non-dominated individual is the size of the largest cuboid enclosing it without including any other non-dominated individuals. In order to find the rankings

```
1. Set $t$, the generation count, to 0.
2. Generate the initial population $P(t)$ of $\mu$ individuals, each of which can be represented as a set of real vectors, $(\mathbf{x_i}, \boldsymbol{\eta_i}), i = 1, \ldots, \mu$. Both $\mathbf{x_i}$ and $\boldsymbol{\eta_i}$ contain $N$ independent variables,
      $\mathbf{x_i} = \{x_i(1), \cdots, x_i(N)\}$,
      $\boldsymbol{\eta_i} = \{\eta_i(1), \cdots, \eta_i(N)\}$.
3. Evaluate the objective vectors of all individuals in $P(t)$ by using the multi-objective function.
4. Calculate the rankings and crowding distances of all individuals
   a. Execute **Dominance-checking**($P(t)$, $C$, $S$).
   b. Execute **Non-dominated-selection**($P(t)$, $C$, $S$, $V$, $\mu$).
5. While the termination condition is not satisfied
   a. For $i$ from 1 to $\mu/2$, select two parents $P^1_{parent_i 1}$ and $P^1_{parent_i 2}$ from $P(t)$ using the tournament selection method.
   b. For $i$ from 1 to $\mu/2$, recombine $P^1_{parent_i 1}$ and $P^1_{parent_i 2}$ using single point crossover to produce two offspring that are stored in the temporary population $P^2$. The population $P^2$ contains $\mu$ individuals.
   c. Mutate individuals in $P^2$ to generate modified individuals that are stored in the temporary population $P^3$. For an individual $P^2_i = (\mathbf{x_i}, \boldsymbol{\eta_i})$, where $i = 1, \ldots, \mu$, create a new individual $P^3_i = (\mathbf{x_i}', \boldsymbol{\eta_i}')$ as follows:
      for $j = 1, \ldots, N$
         $x'_i(j) = x_i(j) + \eta_i(j)R(0,1)$,
         $\eta'_i(j) = \eta_i(j)\exp(\frac{1}{\sqrt{2N}}R(0,1) + \frac{1}{\sqrt{2\sqrt{N}}}R_j(0,1))$
      where $x_i(j), \eta_i(j), x'_i(j)$, and $\eta'_i(j)$ denote the $j^{th}$ component of $\mathbf{x_i}, \boldsymbol{\eta_i}, \mathbf{x_i}'$, and $\boldsymbol{\eta_i}'$ respectively. $R(0,1)$ denotes a normally distributed 1D random number with zero mean and standard deviation of one. $R_j$(0,1) indicates a new random value for each value of $j$.
   d. Evaluate the objective vectors of all individuals in $P^3$.
   e. Combine the parent population $P(t)$ with $P^3$ to generate a population $P^4$ containing $2\mu$ individuals.
   f. Check the dominance of all individuals in $P^4$ by executing **Dominance-checking**($P^4$, $C$, $S$).
   g. Select $\mu$ individuals from $P^4$ and store them in the next population $P(t+1)$. The individuals are selected by executing **Non-dominated-selection**($P^4$, $C$, $S$, $V$, $\mu$).
   h. $t = t + 1$.
6. Return the non-dominated individual with the smallest value for the second objective in the last population.
```

Fig. 1 The MOEA algorithm

```
Procedure Dominance-checking($Pop$, $C$, $S$)

• $Pop$ be the input population.
• $C$ be an array of integers.
• $S$ be an array of sets of integers.

1. Set $k$ be the number of individuals in the population $Pop$.
2. For $i$ from 1 to $k$
   • Set $P_i$ be the $i^{th}$ individual of $Pop$.
   • Set $C_i$, the $i^{th}$ element of $C$, be 0.
   • Set $S_i$, the $i^{th}$ element of $S$, be an empty set.
   • For $j$ from 1 to $k$
     – If $i$ is not equal to $j$, then
           If $P_i$ is dominated by $P_j$, then
               increase $C_i$ by 1
           else if $P_j$ is dominated by $P_j$, then
               $S_i = S_i \cup \{j\}$.
```

Fig. 2 The **dominance-checking** algorithm

and the crowding distances of other individuals, the non-dominated individuals are assumed to be removed from the population and thus another set of non-dominated individuals can be obtained. The rankings of these individuals should be larger than those of the previous non-dominated individuals. The crowding distances of the individuals can also be found. Similarly, the same approach can be applied to find the rankings and the crowding distances of all other individuals. The procedures **dominance checking** and **non-dominated selection** are used to find these values. Their algorithms are given in Figs. 2 and 3, respectively.

Then, $\mu/2$ pairs of parents will be selected from the population. Two offspring will be generated for each pair of parents by using crossover and mutation. In other words, there will be μ offspring. The objective vectors of all offspring will be obtained, and the parent population will be combined with the μ offspring to generate a selection pool. Thus there are 2μ individuals in the selection pool. The rankings and the crowding distances of all individuals in the selection pool can be obtained by using the **dominance-checking** and **non-dominated-selection** procedures. μ Individuals will be selected from the selection pool, and they will form the next population of individuals. This evolution process will be repeated until the termination condition is satisfied.

Finally, the non-dominated individual with the smallest value for the second objective in the last population will be returned. In general, the computation of the parallel MOEA can be roughly divided into five types: (1) Fitness value evaluation (steps 3 and 5(d)) (2) Parent selection (step 5(a)) (3) Crossover and mutation (steps 5(b) and 5(c), respectively) (4) The **dominance-checking** procedure designed for parallel algorithms (steps 4(a) and 5(f)) (5) The **non-dominated-selection**

```
Procedure Non-dominated-selection($Pop$, $C$, $S$, $V$, $L$)

• $Pop$ be the input population.
• $C$ be an input array of integers.
• $S$ be an input array of sets of integers.
• $V$ be an array of numbers. If the $i^{th}$ element is 1, the $i^{th}$ individual is
  selected, otherwise it is not selected.
• $L$ be the number of individuals to be selected.

1. Set $Selected$ to 0. It represents the number of individuals that are selected.
2. Set all elements of $V$ to 0.
3. Set $rank$, the current ranking of individuals, to 0.
4. Set $F$, the current non-dominated set, to be an empty set.
5. For each $P_i \in Pop$
       If the corresponding $C_i$ is 0, then
           $F = F \cup \{P_i\}$.
6. Calculate the crowding distances of all individuals in $F$.
7. While $Selected + |F| \leq L$
   a. Set $NF$, the next non-dominated set, to be an empty set.
   b. For each $P_i \in F$
       i. Set the ranking of $P_i$ to $rank$.
      ii. Set the corresponding $V_i$ to 1.
     iii. For each $k \in S_i$
           • Decrease $C_k$ by 1.
           • If $C_k$ is 0, then set $NF = NF \cup \{P_k\}$.
      iv. Increase $Selected$ by 1.
   c. Calculate the crowding distances of all individuals in $NF$.
   d. Increase $rank$ by 1.
   e. Set $F$ to $NF$.
8. If $Selected < L$, then
   • Based on the crowding distances of individuals in $F$, select ($L -$
     $Selected$) individuals.
   • Set their rankings to $rank$.
   • Set their $V_i$ to 1.
```

Fig. 3 The **non-dominated-selection** algorithm

procedure which selects individuals from the selection pool (steps 4(b) and 5(g)) These operations will be discussed in the following subsections.

3.1 Data Organization

Suppose we have μ individuals and each contains N variables. The most natural representation for an individual is an array. Figure 4 shows how we represent μ

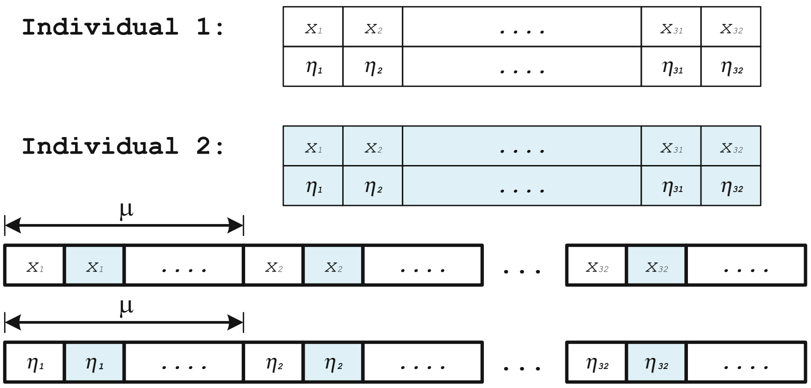

Individual 1:
Individual 2:
x_1 x_2 .... x_{31} x_{32}
η_1 η_2 .... η_{31} η_{32}
μ
x_1 x_1 .... x_2 x_2 .... x_{32} x_{32} ....
η_1 η_1 .... η_2 η_2 .... η_{32} η_{32} ....

Fig. 4 Representing individuals of 32 variables on global memory

individuals in the global memory. Without loss of generality, we take $N = 32$ as an example of illustration throughout this chapter.

Since the global memory space is not cached in some GPUs,[1] it is important to use the right access pattern to get maximum memory bandwidth. When the concurrent memory accesses by 16 CUDA threads in a half wrap (for GPUs of compute capability 1.x) or 32 threads in a warp (for GPUs of compute capability 2.x) [2] can be coalesced into a single memory transaction, the global memory bandwidth can be improved [21]. In order to fulfill the requirements for coalesced memory accesses, the same variables from all individuals are grouped and form a tile of μ values in the global memory as shown in Fig. 4. On the other hand, the efficiency of accessing the same variables of all individuals in parallel will be reduced, if an individual is mapped to 32 consecutive locations, because the simultaneous memory accesses cannot be coalesced and multiple memory transactions are required.

3.2 *Fitness Value Evaluation*

In steps 3 and 5(d) of Fig. 1, the objective vectors of all individuals in the initial population and all offspring in the temporary population P^3 are calculated. Each CUDA thread returns an objective vector by feeding the multi-objective function Γ with the decision variables of the individual. This evaluation process usually consumes significant part of the computational time.

[1]Cache for global memory is only available in GPUs of compute capability 2.x.

[2]In CUDA, the GPU creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. A half-warp is either the first or second half of a warp.

Since no interaction among threads is required during evaluation, the evaluation is fully parallelizable. Recall that the individuals are broken down and stored in the tiles within the global memory. The evaluation kernel looks up the corresponding variable in each tile during the evaluation. The objective vectors are saved in an output array of size $m \times \mu$, because each thread generates a vector of m values.

3.3 Parent Selection

The selection process determines which individuals will be selected as parents to reproduce offspring. Different selection methods including the *roulette wheel selection, truncation selection*, and *stochastic tournament* have been applied in the field [11]. The stochastic tournament is employed in our parallel MOEA, because it is not practical to implement a parallel method on GPU to collect statistical information on the whole population. Since this information is not required in the stochastic tournament while it is needed for the other two methods, the stochastic tournament is more suitable for GPU.

In the tournament selection method, two groups of q individuals are randomly chosen from the population for each CUDA thread. The number q is the tournament size. The two individuals with the smallest rankings within the two groups will be selected as the parents to produce offspring by using crossover and mutation. If more than one individual in a group has the smallest ranking, the one with the largest crowding distance will be chosen. A GPU-based random number generator [14, 22] is used to generate a large number of random numbers stored in the global memory. These random numbers can then be used for selecting individuals randomly. Since there are $\mu/2$ CUDA threads (see step 5(a) of Fig. 1), $\mu \times q$ random numbers are used.

We implement our parent selection method in a kernel program. The input of the kernel is the arrays containing the rankings and the crowding distances of the individuals, as well as the array containing the random numbers. While the output of the kernel is the addresses of the breeding parents selected. The addresses of all selected parents are stored in an output array of size μ.

3.4 Crossover and Mutation

The selection operator focuses on searching promising regions of the solution space. However, it is not able to introduce new solutions that are not in the current population. In order to escape from local optima and introduce larger population diversity, the crossover and mutation operators are applied.

We apply single-point crossover in our parallel MOEA. The kernel program takes input arrays containing the addresses of the selected parents, the individuals, and the random numbers. It generates μ offspring individuals that are stored in the global memory.

To accomplish the mutation process on GPU, we designed two kernel programs, one for computing $\mathbf{x}'$ and the other for $\boldsymbol{\eta}'$. They implement the Cauchy mutation method proposed by Yao and Liu [31]. The individuals $\mathbf{x_i}$ and $\boldsymbol{\eta}_\mathbf{i}$ are stored in two input arrays while the mutated offspring are generated and written to two output arrays $\mathbf{x_i}'$ and $\boldsymbol{\eta}'_\mathbf{i}$. Besides, random numbers stored in the global memory are used by the two kernels.

3.5 Dominance Checking

We implement the fast dominance-checking procedure applied in NSGA-II [8]. For each individual in the population, two entities are evaluated: C_i is the number of the other individuals that dominates the ith individual and S_i is the set of the other individuals that are dominated by the ith individual. Thus the ith individual is non-dominated if the corresponding C_i is 0. This procedure is efficient because only $O(\mu^2)$ dominance comparisons should be performed.

Suppose that there are 2μ individuals (step 5(f) of Fig. 1), the C_i is stored in an array of short integer. On the other hand, the S_i are represented in a 2D array of bit, S. If the jth bit of the ith row of S is 1, the jth individual is dominated by the ith individual. The size of C is 2μ short integers, while that of S is $4\mu^2$ bits. For example, their sizes are 16 KB and 8 MB, respectively, if μ is 4,096. Their sizes are, respectively, 64 KB and 128 MB if μ is 16,384.

The kernel program implementing the procedure of Fig. 2 takes an input array of objective vectors of all individuals and produces C and S in the global memory. Since there is no interaction among the CUDA threads, they can efficiently execute this kernel in parallel.

Because of the reasons described in Sect. 3.6, the objective vectors, C and S, are transferred from the global memory to the CPU memory after computing C and S.

3.6 Non-dominated Selection

Instead of executing non-dominated selection on GPU, this procedure is performed on CPU because of a number of reasons. First, the number of individuals in F (step 7(b) of Fig. 3) varies from one iteration to other iterations. Some CUDA threads may be idle in some iterations. Second, it is necessary to sort the individuals in NF (step 7(c) of Fig. 3) according to their objective vectors, in order to calculate their crowding distances. Although it is possible to execute a sorting algorithm on GPU [15], its efficiency is doubtful when it is used to sort a relatively small number of values. Third, many synchronizations may be performed as the variables C_k and NF may be accessed and modified by different threads concurrently (step 7(b)iii of Fig. 3).

The CPU implementation of the procedure accesses the objective vectors of individuals, C and S, that are already stored in the CPU memory. It finds the rankings and the crowding distances and summarizes the selected individuals in the output array V. Then, the rankings, the crowding distances, and V are transferred to the GPU memory.

Based on the information stored in V, the actual replacements of individuals, the rankings, and the crowding distances are performed on GPU. Thus, the data movement between the GPU memory and the CPU memory can be reduced, because it is not necessary to transfer the individuals between the two memory spaces.

In summary, the whole MOEA program, except the non-dominated-selection procedure, is executed on GPU. Thus, our parallel MOEA gains the most benefit from the SIMT architecture of GPU.

4 Experiments

In this section, the parallel MOEA is applied to a data mining problem in direct marketing. The objective of the problem is to predict potential prospects from the buying records of previous customers. Advertising campaign, which includes mailing of catalogs or brochures, is then targeted on the group of potential prospects. Hence, if the prediction is accurate, it can help to enhance the response rate of the advertising campaign and increase the ROI. The direct marketing problem requires ranking the customer database according to the customers' scores obtained by the prediction models [23].

We compare the parallel MOEA, the parallel HGA [29], and the DMAX approach for learning prediction models. Since the parallel HGA is a single-objective optimization algorithm, it is used to optimize the objective defined in (6). The experiment test bed was an Intel Pentium Dual E2220 CPU with an nVidia GTX 460 display card, with 2,048 MB main memory and 768 MB GPU memory. The CPU speed is 2.40 GHz and the GPU contains 336 unified shaders. Microsoft Windows XP Professional, Microsoft Visual C++ 2008, and nVidia CUDA version 3.1 are used to develop the parallel MOEA and the parallel HGA. On the other hand, the DMAX approach is developed in Java. The following parameters have been used in the experiments:

- Population size: $\mu = 256$
- Tournament size: $q = 2$
- Maximum number of generation: $G = 500$
- The percentage of customers to be solicited: $r\,\% = 20\,\%$

4.1 Methodology

The prediction models are evaluated on a large real-life direct marketing dataset from a US-based catalog company. It sells multiple product lines of merchandise including gifts, apparel, and consumer electronics. This dataset contains the records of 106,284 consumers in a recent promotion as well as their purchase history over a 12-year period. Furthermore, demographic information from the 1995 US Census and credit information from a commercial vendor were appended to the main dataset. Altogether, each record contains 361 variables. The most recent promotion sent a catalog to every customer in this dataset and achieved a 5.4 % response rate, representing 5,740 buyers.

Typical in any data mining process, it is necessary to reduce the dimension of the dataset by selecting the attributes that are considered relevant and necessary. Towards this feature selection process, there are many possible options. For instance, we could use either a *wrapper* selection process or a *filter* selection process [25]. In a wrapper selection process, different combinations are iteratively tried and evaluated by building an actual model out of the selected attributes. In a filter selection process, certain evaluation function, which is based on information theory or statistics, is defined to score a particular combination of attributes. Then, the final combination is obtained in a search process. In this experiment, we have applied the forward selection method to select 17 variables, that are relevant to prediction, out of the 361 variables.

Since direct marketers usually have a fixed budget and can only contact a small portion of the potential customers in their dataset (e.g., top 20 %), simple error rates or overall classification accuracy of models are not meaningful. To support direct marketing and other targeted marketing decisions, maximizing the number of true positives at the top deciles is usually the most important criterion for assessing the performance of prediction models [1, 33].

To compare the performance of different prediction models, we use decile analysis which estimates the enhancement of the response rate and the profit for marketing at different depth-of-file. Essentially, the ranked list is equally divided into ten deciles. Customers in the first decile are the top-ranked customers that are most likely to give response and generate high profit. On the other hand, customers in the tenth decile are ranked lowest. To measure the performance of a model at different depths of file, direct marketing researchers have relied on the "lift," which is the ratio of true positives to the total number of records identified by the model in comparison with that of a random model at a specific decile of the file. Thus, comparing the performance of models across depths of file using cumulative lifts or the "response rate" are necessary to inform decisions in direct marketing. Profit lift is the amount of extra profit generated with the new method over that generated by a random method. In this sense, the goal to achieve higher lifts in the upper deciles becomes a ranking problem based on the scores returned by the model and help to evaluate the effectiveness of targeted marketing and to forecast sales and profitability of promotion campaigns.

4.2 Cross-Validation Results

In order to compare the robustness of the prediction models, we adopt a ten-fold cross-validation approach for performance estimation. A dataset is randomly

Table 1 Cumulative lifts of the models learned by different methods

Decile	Parallel MOEA	Parallel HGA	DMAX
0	**358.47 (25.11)**	147.53 (16.84)$^+$	310.60 (34.10)$^+$
1	**270.46 (9.54)**	132.51 (6.98)$^+$	234.20 (20.50)$^+$
2	**219.11 (6.58)**	125.68 (5.82)$^+$	195.50 (12.30)$^+$
3	**182.48 (3.92)**	120.65 (5.88)$^+$	170.60 (6.30)$^+$
4	**156.11 (2.54)**	115.84 (4.70)$^+$	150.90 (3.90)
5	**138.17 (2.51)**	111.59 (3.42)$^+$	136.60 (2.90)
6	124.95 (2.99)	109.12 (2.69)$^+$	**125.20 (2.10)**
7	114.51 (2.63)	106.17 (1.88)$^+$	**115.40 (1.60)**
8	106.77 (1.18)	103.66 (0.84)$^+$	**106.90 (1.20)**
9	100.00 (0.00)	100.00 (0.00)	100.00(0.00)

partitioned into 10 mutually exclusive and exhaustive folds. Each time, a different fold is chosen as the test set, and other nine folds are combined together as the training set. Prediction models are learned from the training set and evaluated on the corresponding test set.

In Table 1, the average of the cumulative lifts of the models learned by different methods are summarized. Numbers in the parentheses are the standard deviations. The highest cumulative lift in each decile is highlighted in bold. The superscript $^+$ represents that the cumulative lift of the model obtained by the parallel MOEA is significant higher at 0.05 level than that of the models obtained by the corresponding methods. The superscript $^-$ represents that the cumulative lift of the model obtained by the parallel MOEA is significantly lower at 0.05 level than that of the corresponding models.

From Table 1, the models generated by the parallel MOEA have the average cumulative lifts of 358.47 and 270.46 in the first two deciles, respectively, suggesting that by mailing to the top two deciles alone, the models generate over twice as many respondents as a random mailing without a model. Moreover, the average cumulative lifts of the models learnt by the parallel MOEA are significantly higher than those of the models obtained by the other methods for the first four deciles.

The average of the cumulative profit lifts of the models learned by different methods are summarized in Table 2. It is observed that the average cumulative profit lifts of the models learnt by the parallel MOEA are significantly higher than those of the models obtained by the other methods for the first three deciles. The average profits for different models are listed in Table 3. Direct marketers can get \$11,461.63 if they use the parallel MOEA to generate models for selecting 20 % of the customers from the dataset. On the other hand, they can get only \$10,514.24 if they apply the DMAX approach for selecting customers. The parallel HGA cannot learn good models because the objective (i.e., (7)) representing the constraint is not considered in this approach.

In order to study the effect of the value of r on the performance of the models learnt by the parallel MOEA, we apply different values of r and compare

Table 2 Cumulative profit lifts of the models learned by different methods

Decile	Parallel MOEA	Parallel HGA	DMAX
0	**621.00 (49.16)**	242.35 (38.54)$^+$	550.80 (61.00)$^+$
1	**382.03 (14.14)**	188.26 (16.5)$^+$	350.80 (24.10)$^+$
2	**278.72 (10.26)**	166.24 (9.71)$^+$	261.70 (15.70)$^+$
3	**221.96 (8.05)**	151.66 (12.04)$^+$	213.30 (7.40)
4	**181.81 (4.90)**	139.39 (9.96)$^+$	179.10 (5.30)
5	155.32 (5.27)	129.93 (5.91)$^+$	**156.30 (4.20)**
6	135.55 (4.51)	121.79 (4.74)$^+$	**137.60 (3.90)**
7	121.20 (3.71)	114.63 (2.95)$^+$	**122.10 (3.00)**
8	**110.20 (1.57)**	108.2 (1.16)$^+$	109.70 (2.00)
9	100.00 (0.00)	100.00 (0.00)	100.00 (0.00)

Table 3 Average profits for the models learned by different methods

Decile	Parallel MOEA	Parallel HGA	DMAX
0	**$9,339.04**	$3,646.49	$8,254.34
1	**$11,461.63**	$5,650.60	$10,514.24
2	**$12,545.96**	$7,472.12	$11,765.58
3	**$13,317.80**	$9,077.67	$12,786.13
4	**$13,631.38**	$10,429.06	$13,420.04
5	$13,974.12	$11,677.79	**$14,053.96**
6	$14,234.71	$12,768.95	**$14,434.60**
7	$14,542.35	$13,744.59	**$14,638.41**
8	**$14,867.79**	$14,588.65	$14,795.77
9	$14,986.09	$14,986.09	$14,986.09

Table 4 Cumulative lifts of the models learned by the parallel MOEA

Decile	$r = 10\,\%$	$r = 30\,\%$	$r = 40\,\%$	$r = 50\,\%$
0	363.28 (24.59)	354.39 (27.66)	363.28 (24.59)	363.28 (24.59)
1	267.32 (8.40)	267.02 (12.43)	267.32 (8.40)	267.32 (8.40)
2	214.89 (5.59)	220.53 (7.28)	214.89 (5.59)	214.89 (5.59)
3	180.27 (4.61)	185.51 (4.78)	180.27 (4.61)	180.27 (4.61)
4	155.63 (3.76)	161.84 (3.74)	155.63 (3.76)	155.63 (3.76)
5	137.96 (3.03)	143.03 (1.98)	137.96 (3.03)	137.96 (3.03)
6	124.69 (2.68)	128.90 (1.85)	124.69 (2.68)	124.69 (2.68)
7	114.10 (1.95)	117.33 (1.48)	114.10 (1.95)	114.10 (1.95)
8	106.16 (1.50)	108.72 (0.85)	106.16 (1.50)	106.16 (1.50)
9	100.00 (0.00)	100.00 (0.00)	100.00 (0.00)	100.00 (0.00)

the cumulative lifts and the cumulative profit lifts of the induced models. From Tables 4 and 5, it is found that our approach is quite stable because it can learn good models for different values of r.

Table 5 Cumulative profit lifts of the models learned by the parallel MOEA

Decile	$r = 10\,\%$	$r = 30\,\%$	$r = 40\,\%$	$r = 50\,\%$
0	621.08 (46.31)	616.99 (51.25)	621.08 (46.31)	621.08 (46.31)
1	377.88 (14.76)	377.51 (18.49)	377.88 (14.76)	377.88 (14.76)
2	276.31 (9.96)	281.15 (11.13)	276.31 (9.96)	276.31 (9.96)
3	217.91 (6.91)	222.98 (10.05)	217.91 (6.91)	217.91 (6.91)
4	181.69 (5.75)	185.63 (6.98)	181.69 (5.75)	181.69 (5.75)
5	154.24 (4.39)	158.46 (3.67)	154.24 (4.39)	154.24 (4.39)
6	134.84 (4.25)	139.18 (3.18)	134.84 (4.25)	134.84 (4.25)
7	119.91 (2.99)	123.25 (2.36)	119.91 (2.99)	119.91 (2.99)
8	108.40 (2.52)	111.31 (0.82)	108.40 (2.52)	108.40 (2.52)
9	100.00 (0.00)	100.00 (0.00)	100.00 (0.00)	100.00 (0.00)

Table 6 The average execution time (in seconds) of the CPU implementation

	Generation								
	100	150	200	250	300	350	400	450	500
OT	129.29	194.15	259.10	324.09	389.11	454.15	519.20	584.29	648.11
DC	0.50	0.79	1.06	1.34	1.60	1.88	2.15	2.41	2.67
NS	0.01	0.02	0.02	0.03	0.03	0.05	0.06	0.06	0.07
FE	128.78	193.35	258.02	322.73	387.48	452.22	516.99	581.82	645.37
ratio	0.9960	0.9959	0.9958	0.9958	0.9958	0.9958	0.9958	1.00	1.00

The OT, DC, NS, and FE rows show the average time in performing, respectively, all steps, the dominance-checking step, the non-dominated-selection step, and fitness evaluations

Table 7 The average execution time (in seconds) of the GPU implementation

	Generation								
	100	150	200	250	300	350	400	450	500
OT	5.75	8.55	11.35	14.14	16.93	19.72	22.51	25.31	28.04
DC	0.03	0.04	0.06	0.07	0.09	0.11	0.14	0.16	0.17
NS	0.03	0.05	0.06	0.07	0.08	0.09	0.10	0.11	0.11
FE	5.68	8.46	11.23	14.00	16.77	19.52	22.28	25.05	27.75
ratio	0.989	0.989	0.989	0.990	0.990	0.990	0.989	0.990	0.990

4.3 Comparison Between GPU and CPU Approaches

We compare the CPU and the GPU implementations of the MOEA. The average execution time of different steps of the CPU implementation is summarized in Table 6. The ratios of the time used in fitness evaluations to the overall execution time are also reported in this table. It can be observed that the fitness evaluation time is significantly higher than that of the other steps because the training sets are very large. The average execution time of the GPU implementation is summarized in Table 7. The parallel MOEA takes about 28 s to learn a model. On the other hand, it takes about 648 and 7,315 s, respectively, for the CPU implementation of the MOEA and the DMAX approach to learn a model.

Table 8 The speedups of the GPU implementation with the CPU implementation

	Generation								
	100	150	200	250	300	350	400	450	500
OT	22.505	22.713	22.835	22.919	22.978	23.024	23.058	23.085	23.115
DC	15.317	18.740	17.542	19.153	18.368	16.782	15.007	15.457	15.440
NS	0.354	0.305	0.346	0.350	0.388	0.527	0.572	0.542	0.602
FE	22.669	22.869	22.982	23.054	23.111	23.167	23.209	23.231	23.255

Table 8 displays the speedups of the overall programs and different steps of the programs. The speedups of the GPU implementation of the dominance-checking procedure range from 15.00 to 18.74. On the other hand, the Pnon-dominated-selection procedure of the GPU implementation is slower than that of the CPU approach. The overall speedup is about 23.1.

Since a marketing campaign often involves huge dataset and large investment, prediction models which can categorize more prospects into the target list are valuable as they will enhance the response rate as well as the ROI. From the experimental results, the prediction models generated by the parallel MOEA are more effective than the other models and the parallel MOEA is significantly faster than the DMAX approach.

5 Conclusions

An important issue in targeted marketing is how to find potential customers who contribute large profits to a firm under constrained resources. In this chapter, we have proposed a data mining method to learn models for identifying valuable customers. We have formulated this learning problem as a constrained optimization problem of finding a scoring function f and a threshold value τ. We have then converted it to an unconstrained MOP.

By limiting f to be a linear function, a parallel MOEA on GPU has been used to handle the MOP and find the parameters of f as well as the value of τ. We have used tenfold cross-validation and decile analysis to compare the performance of the parallel MOEA, the parallel HGA, and the DMAX approach for a real-life direct marketing problem. Based on the cumulative lifts, cumulative profit lifts, and average profits, it can be concluded that the models generated by the parallel MOEA significantly outperform the models learnt by other methods in many deciles. Thus, the parallel MOEA is more effective. Moreover, it is significantly faster than the DMAX approach.

We have performed experiments to compare our parallel MOEA and a CPU implementation of MOEA. It is found that the overall speedup is about 23.1. Thus, our approach will be very useful for solving difficult direct marketing problems that involve large datasets and require huge population sizes.

For future work, we will extend our method to learn nonlinear scoring functions and apply it to other targeted marketing problems under resource constraints.

Acknowledgements This work is supported by Hong Kong RGC General Research Fund LU310111.

References

1. Bhattacharyya, S.: Direct marketing performance modeling using genetic algorithms. INFORMS J. Comput. **11**(3), 248–257 (1999)
2. Bult, J.R., Wansbeek, T.: Optimal selection for direct mail. Manag. Sci. **14**(4), 1362–1381 (1995)
3. Chitty, D.M.: A data parallel approach to genetic programming using programmable graphics hardware. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO '07), vol. 2, pp. 1566–1573 (2007)
4. Coello Coello, C.A., Toscano Pulido, G., Salazar Lechuga, M.: Handling multiple objectives with particle swarm optimization. IEEE Trans. Evol. Comput. **8**(3), 256–279 (2004)
5. Corne, D.W., Jerram, N.R., Knowles, J.D., Oates, M.J.: PESA-II: region-based selection in evolutionary multiobjective optimization. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'2001), pp. 283–290 (2001)
6. Cui, G., Wong, M.L.: Implementing neural networks for decision support in direct marketing. Int. J. Market Res. **46**(2), 1–20 (2004)
7. Deb, K.: Multi-Objective Optimization Using Evolutionary Algorithms. Wiley, New York (2001)
8. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)
9. Fieldsend, J.E., Everson, R.M., Singh, S.: Using unconstrained elite archives for multiobjective optimization. IEEE Trans. Evol. Comput. **7**(3), 305–323 (2003)
10. Fok, K.L., Wong, T.T., Wong, M.L.: Evolutionary computing on consumer-level graphics hardware. IEEE Intell. Syst. **22**(2), 69–78 (2007)
11. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading (1989)
12. Harding, S., Banzhaf, W.: Fast genetic programming on GPUs. In: Proceedings of the 10th European Conference on Genetic Programming (EuroGP'2007), pp. 90–101 (2007)
13. Horn, J., Nafpliotis, N., Goldberg, D.E.: A niched Pareto genetic algorithm for multiobjective optimization. In: Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence, vol. 1, pp. 82–87 (1994)
14. Howes, L., Thomas, D.: Efficient random number generation and application using CUDA. In: Nguyen, H. (ed.) GPU Gems 3, pp. 805–830. Addison-Wesley, Reading (2007)
15. Kipfer, P., Westermann, R.: Improved GPU Sorting. In: Pharr, M. (ed.) GPU Gems 2, pp. 733–746. Addison-Wesley, Reading (2005)
16. Knowles, J.D., Corne, D.W.: Approximating the nondominated front using the Pareto Archived Evolution Strategy. Evol. Comput. **8**(2), 149–172 (2000)
17. Knowles, J.D., Corne, D.W.: Properties of an adaptive archiving algorithm for storing nondominated vectors. IEEE Trans. Evol. Comput. **7**(2), 100–116 (2003)
18. Langdon, W.B., Banzhaf, W.: A SIMD interpreter for genetic programming on GPU graphics cards. In: Proceedings of the 11th European Conference on Genetic Programming (EuroGP'2008), pp. 73–85 (2008)
19. Laumanns, M., Thiele, L., Deb, K., Zitzler, E.: Combining convergence and diversity in evolutionary multi-objective optimization. Evol. Comput. **10**(3), 263–282 (2002)
20. Mulhern, F.J.: Customer profitability analysis: measurement, concentration, and research directions. J. Interact. Market. **13**(1), 25–40 (1999)
21. nVidia: NVIDIA CUDA C Programming Guide Version 3.1. Technical Report, nVidia Corporate (2010). http://developer.nvidia.com/object/cuda.html

22. Pang, W.M., Wong, T.T., Heng, P.A.: Generating massive high-quality random numbers using GPU. In: Proceedings of the 2008 Congress on Evolutionary Computation (CEC 2008), pp. 841–847 (2008)
23. Rud, O.P.: Data Mining Cookbook: Modeling Data for Marketing, Risk and Customer Relationship Management. Wiley, New York (2001)
24. Runarsson, T.P., Yao, X.: Search biases in constrained evolutionary optimization. IEEE Trans. Syst. Man Cybern. C **35**(2), 233–243 (2005)
25. Singh, M.: Learning Bayesian networks for solving real-world problems. Ph.D. thesis, University of Pennsylvania (1998)
26. Wang, Y., Cai, Z., Guo, G., Zhou, Y.: Multiobjective optimization and hybrid evolutionary algorithm to solve constrained optimization problems. IEEE Trans. Syst. Man Cybern. B **37**(3), 560–575 (2007)
27. Wilson, G., Banzhaf, W.: Linear genetic programming GPGPU on Microsoft's Xbox 360. In: Proceedings of the 2008 Congress on Evolutionary Computation (CEC'2008), pp. 378–385 (2008)
28. Wong, M.L., Wong, T.T., Fok, K.L.: Parallel evolutionary algorithms on graphics processing unit. In: Proceedings of the 2005 Congress on Evolutionary Computation (CEC'2005), pp. 2286–2293 (2005)
29. Wong, M.L., Wong, T.T., Fok, K.L.: Parallel hybrid genetic algorithms on consumer-level graphics hardware. In: Proceedings of the 2006 Congress on Evolutionary Computation (CEC'2006), pp. 10330–10337 (2006)
30. Yan, L., Baldasare, P.: Beyond classification and ranking: constrained optimization of the ROI. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 948–953 (2006)
31. Yao, X., Liu, Y.: Fast evolutionary programming. In: Evolutionary Programming V: Proceedings of the 5th Annual Conference on Evolutionary Programming. MIT Press, Cambridge (1996)
32. Yen, G.G., Lu, H.: Dynamic multiobjective evolutionary algorithm: adaptive cell-based rank and density estimation. IEEE Trans. Evol. Comput. **7**(3), 253–274 (2003)
33. Zahavi, J., Levin, N.: Applying neural computing to target marketing. J. Direct Market. **11**(4), 76–93 (1997)
34. Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the strength Pareto evolutionary algorithm. In: Giannakoglou, K., Tsahalis, D., Periaux, J., Papailou, P., Fogarty, T. (eds.) EUROGEN 2001. Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems, pp. 95–100. International Center for Numerical Methods in Engineering, Barcelona (2002)
35. Zitzler, E., Thiele, L.: Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. IEEE Trans. Evol. Comput. **3**(4), 257–271 (1999)

Part III
Applications

Large-Scale Bioinformatics Data Mining with Parallel Genetic Programming on Graphics Processing Units

William B. Langdon

Abstract The NCBI GEO GSE3494 breast cancer dataset contains hundreds of Affymetrix HG-U133A and HG-U133B GeneChip biopsies each with a million variables. Multiple genetic programming (GP) runs on a graphics processing unit (GPU) hardware, each with a population of five million programs both winnows (selects) useful variables from the chaff and evolves small (three inputs) data models. The SPMD CUDA interpreter exploits the GPU's single instruction multiple data (SIMD) mode of parallel computing, even though the GP populations contain different programs. A 448 node nVidia Fermi C2050 Tesla graphics card delivers 8.5 giga GPops per second. In addition to describing our implementation, we survey current GPGPU work in bioinformatics and genetic programming.

1 Introduction

Since they offer cheap high-performance computing there is great interest in using mass market graphics hardware (GPUs) for scientific applications. For example the Chinese Tianhe-1A 2.566 petaflop supercomputer contains 7,168 nVidia Tesla M2050 general purpose GPUs. However a lot of scientific and engineering can be done with more modest computers, and we will concentrate on affordable personal computers or indeed laptop computers with one or more graphics cards or their Tesla compute-only equivalents. More than 100 million GPUs have been sold [24]. This availability and their price/performance ratio have led to the increasing use of essentially consumer gaming or entertainment hardware for research and engineering purposes. The field is often called general purpose computing on GPU (GPGPU) [84]. Until recently the doubling of the number of

W.B. Langdon (✉)
Department of Computer Science, University College, London, UK
e-mail: w.langdon@cs.ucl.ac.uk

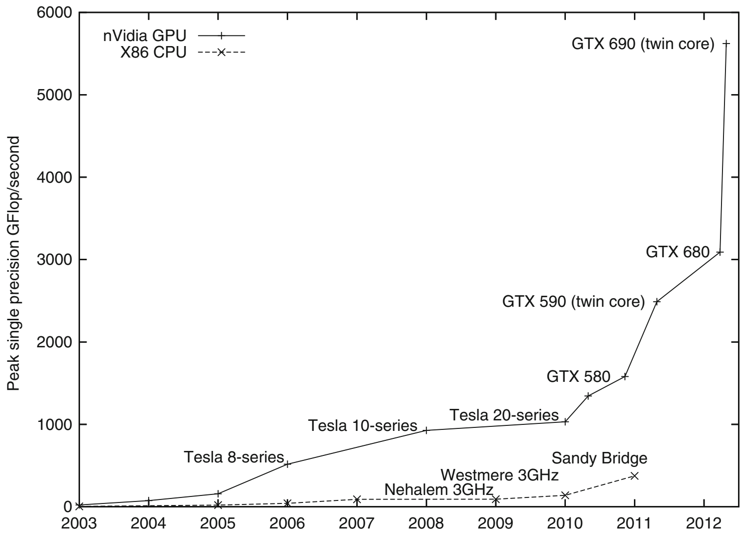


Fig. 1 Comparison of increase in speed of graphics cards (+ GPU) and CPU (× x86) (data supplied by nVidia). Similar trends hold for double precision and integer performance

transistors in computer chips every 18 months ("Moore's Law") was a fact of life [79] and similar exponential rises occurred in processing speed and disk and memory storage capacity. The compound effect of Moore's Law has led to literally millionfold increases in hardware performance during careers in the software industry. Naysayers have frequently pointed out the impossibility of exponential growth continuing indefinitely; however, today it looks like they are right in at least one important aspect, and we have reached the end of Moore's Law as it has been applied to processor speed. In commercial terms, the industry remains dominated by descendants of Intel's 8086 silicon chips, yet for half a dozen years we have seen no major increase in CPU clock speed since the 3 GHz Pentium (see lower plot in Fig. 1). If clock speeds had continued to double every 1.5 years, we would have 25 GHz Pentiums on our desks and in our laptops. This has not happened. It looks like it will never happen.

In its original sense the manufacturers of silicon chips continue to obey Moore's Law, and the number of transistors per chip has continued to increase. Recently Izydorczyk and Izydorczyk [37] suggested Moore's Law will continue to hold for at least the next 22 years. However they appear to accept today's limit of about 3.5 GHz on processor clocks.

The additional transistors packed ever more densely onto chips have been used to create still bigger memory, particularly on-chip cache memory, more exotic instruction sets (e.g. vector, parallel and special purpose instructions), and especially to build multiple CPU cores on the same chip. Dual and quad cores are now commonplace. Eight- and even sixteen-core Pentium computers are now on the horizon. It looks like we are really seeing the parallel future which has been forecast even before the transputer [1].

Since our initial results on the breast cancer survival prediction dataset, GPU development has continued apace. For example, both AMD and nVidia have GPUs which claim to deliver more than a teraflop at a cost of a few hundred dollars.

The next section will describe scientific and engineering computing on GPUs. Some successful applications of GPUs to bioinformatics will be described in Sect. 3. In Sect. 4 we will summarise our original RapidMind work [57] in which genetic programming [55] is used to data mine a small number of indicative mRNA gene transcript signals from breast cancer tissue samples taken during surgery, each with more than a million variables, to predict long-term survival. In [57] we described the medical problem and the way genetic programming [53] and a GPU simultaneously picked three of the million mRNA measurements available and found a simple non-linear combination of them which predicts long-term outcomes at least as well as DLDA, SVM and KNN using 700 measurements [78]. Before concentrating on using genetic programming [3, 40, 59, 86] in parallel on a GPU, Sect. 5 briefly describes the major hardware components of GPUs and programming them. Then Sect. 6 describes the new GP and CUDA code. We refer the interested reader to [57] for details of the data source and how they were obtained, checked and normalised. The experiments are repeated using the new CUDA kernel. (The results are summarised in Sect. 7.) The new system avoids many restrictions imposed by RapidMind and uses modern Tesla hardware (C2050) to deliver a more than tenfold speedup (Sect. 8). Finally in Sect. 9 we consider how successfully our previous predictions about GPGPU have panned out and make new ones. We conclude (Sect. 10) that GPGPU will be one of the parallel techniques of the future, but note that it is still held back by development tools.

2 Using Games Hardware GPUs for Science

Owens et al. [83, 84] surveyed scientific and engineering applications running on mass market graphics cards. Today's GPUs can greatly exceed the floating point performance of their host CPU; see Fig. 1. This speed comes at a price.

GPUs provide a restricted type of parallel processing, often referred to as single instruction multiple data (SIMD) or more precisely single program multiple data (SPMD). Each of the many processors simultaneously runs the same program on different data items (see Fig. 2). Being tailored for fast real-time production of interactive graphics, principally for the computer gaming market, GPUs are tailored to deal with rendering of pixels and processing of fragments of three-dimensional scenes very quickly. Each is allocated a processor and the GPU program is expected to transform it into another data item. The data items need not be of the same type. For example the input might be a triangle in three dimensions, including its orientation, and the output could be a colour expressed as four floating point numbers (RGB and alpha).

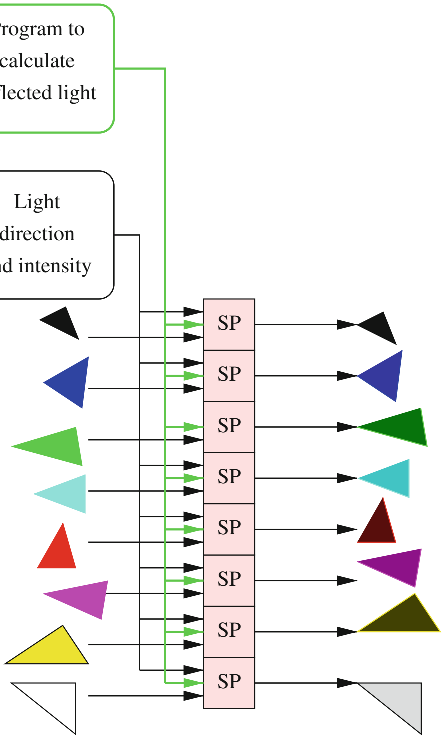

Program to calculate reflected light
Light direction and intensity
SP
SP
SP
SP
SP
SP
SP
SP

Fig. 2 An example of SIMD parallel processing. The stream processors (SP) simultaneously run the same program on different data and produce different answers. In this example the program has two inputs. One describes a *triangle* (position, colour, nature of its surface: matt, how shiny). The second input refers to a common light source and so all stream processors use the same value. Each stream processor calculates the apparent colour of its individual triangle. Notice, here, each output is independent of all the others and so they can all be calculated in parallel

Typical GPUs are optimised so that programs can read data from multiple data sources (e.g. background scenes, placement of lights, reflectivity of surfaces) but generate one output. This parallel writing of data greatly simplifies and speeds the operation of the GPU. Even so, both reading and writing from memory are still bottlenecks. This is true for the GPU's own memory but doubly so when data are transferred to/from the host PC and the GPUs.

The manufacturers continue to publish figures claiming enormous peak floating point performance. In practice such figures are not attainable. A more useful statistic is often how much faster an application runs after it has been converted to run on a GPU. However, like FLOPS, the number of GP operations per second (GPops) allows easier comparison of different GP implementations.

Many scientific applications and in particular bioinformatics applications are inherently suitable for parallel computing. In many cases data can be divided into almost independent chunks which can be acted upon almost independently. There are many different types of parallel computation which might be suitable for bioinformatics. Applications where a GPU might be suitable are characterised by:

- Maximum dataset size $\approx 10^9$.
- Maximum dataset data rate $\approx 10^9$ bytes/s.
- Up to 10^{11} floating point operations per second (FLOPs).

- Applications which are dominated by small computationally heavy cores, i.e. a large number of computations per data item (known as arithmetic intensity).
- Core has simple data flow. Possibly a large fan-in and simple data stream output.

Naturally as GPUs continue to become more powerful these figures continue to change.

3 GPUs in Bioinformatics and Computational Intelligence

As might be expected, GPUs have been suggested for medical image processing applications for several years now. However we concentrate here on molecular bioinformatics. We anticipate that after a few key algorithms are successfully ported to GPUs, within a few years bioinformatics will adopt GPUs for many of its routine applications. As might be expected, early results were mixed.

Charalambous et al. successfully used a relatively low-powered GPU to demonstrate inference of evolutionary inheritance trees (by porting RAxML onto an nVidia FX 5700) [9]. However a more conventional MPI cluster was subsequently used [98]. Recently a CUDA version of the alternative MrBayes tool was published [112].

Sequence comparison is the life blood of bioinformatics. Liu et al. ran the key Smith–Waterman algorithm on a high-end GPU [68]. They demonstrated a reduction by a factor of up to 16 in the lookup times for most proteins. Smith–Waterman has also been ported to the Sony PlayStation 3 [106] and the GeForce 8800 (CUDA) [76]. Trapnell and Schatz also used CUDA to port another sequence searching tool (MUMmer) to another G80 GPU and obtained speedups of up to 13× when matching short DNA strands against much longer sequences [99]. More recently Vouzis and Sahinidis [101] ported NCBI's Blast protein sequence alignment tool to CUDA, but report only modest speedups perhaps because of the large data volumes and their insistence on exactly emulating the original serial code. By breaking queries into GPU-sized fragments, they were able to run short sequences (e.g. 50 DNA bases) against the complete human chromosome. Successful ports and CUDA implementations of sequence tasks include GBOOST [111] (40-fold), SOAP3 [67] (7.5–20× faster) and MrBayes [112] (19×, more with a second GPU).

Liu et al. used GPUs to model biomolecular pathways [66] (26–33×), and Zhou et al. report speedups of 12×, 47× and 367× for Gillespie, LSODA and Euler–Maruyama using their CUDA-sim Python package [113]. Kannan and Ganji [39] also report 10–47-fold speedup when porting AutoDock (a biomolecular drug discovery tool). Gobron et al. used OpenGL on a high-end GPU to drive a cellular automata simulation of the human eye and achieved real-time processing of webcam input [25]. GPUs have also been used in medical engineering, e.g. a GeForce 8800 provided a 15–20-fold speedup, improving the haptic response of a real-time interactive surgery simulation tool [69]. Dowsey et al. wrote 2D gel electrophoresis image registration code in Cg ("C for graphics") so that it could be offloaded onto an nVidia GPU [14].

The better GPU applications may claim speedups of a factor of ten or more; however, the distributed protein folding system folding@home obtains 60 times as much free computation per donated GPU as it does per donated CPU [84, p. 983]. The same authors also claim an almost 3,600-fold speed up on a biomolecule dynamics simulation, albeit at the cost of using four FX 5600 GPUs [84, p. 995].

Computational intelligence applications of GPUs have included artificial neural networks (e.g. multilayer perceptrons [71, 91], self-organising networks [88] and spiking neural networks [110]), fuzzy logic [34], genetic algorithms [22, 72, 80, 85, 95, 97, 107] and genetic programming [4, 6–8, 10, 13, 15–19, 27–33, 35, 36, 43, 45–47, 49, 56, 57, 62–65, 70, 73–75, 77, 87, 90, 92–94, 96, 100, 102–104, 108]. Most GPGPU applications have only required a single graphics card; however, Fan et al. have shown large GPU clusters are also feasible [20]. In 2008 the first computational intelligence on GPU special session (CIGPU-2008) was held in Hong Kong [105]. This has become an annual event. As Owens [83] makes clear, games hardware has now broken out of the bedroom into scientific and engineering computing.

4 Gene Expression in Breast Cancer

We have previously [57] used genetic programming to data mine gene expression measurements provided by Miller et al. [78]. We will mostly be concerned with updating the original RapidMind code to CUDA and its improved performance. However we start by recapping the data-mining problem. Miller et al. describe the collection and analysis of cancerous tissue from most of the women with breast tumours in the three years 1987–1989 in Uppsala in Sweden. Miller's primary goal was to investigate p53, a gene known to be involved in the regulation of other genes and implicated in cancers. In particular they studied the implications of mutations of p53 in breast cancer. The p53 genes of 251 women were sequenced so that it was known whether they were mutant or not. Affymetrix GeneChips (HG-U133A and HG-U133B) were used to measure mRNA concentrations in each biopsy. Various other data were recorded, in particular whether the cancer was fatal or not.

Each of the two types of GeneChips used contained more than half a million DNA probes arranged in a 712×712 square $(12.8\,\text{mm})^2$ array. (Current designs now exceed five million DNA probes on the same half inch square array.)

4.1 Uppsala Breast Cancer Affymetrix GeneChip Datasets

As part of our large survey of GeneChip flaws [60], we had already downloaded all the HG-U133A and HG-U133B datasets in GEO [5] (6,685 and 1,815, respectively) and calculated a robust average for each probe. These averages across all these human tissues were used to normalise the 251 pairs of HG-U133A and HG-U133B GeneChips and flag locations of spatial flaws [57]. R code to quantile normalise and

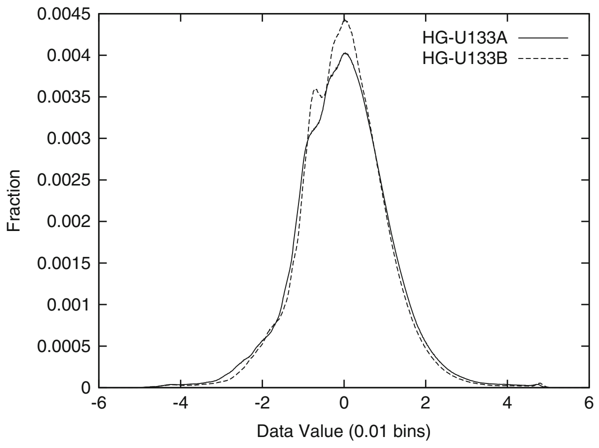


Fig. 3 Uppsala breast cancer distribution of log deviation from average value

detect spatial flaws is available via http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/R. The value presented to GP is the probe's normalised value minus its average value from GEO. This gives an approximately normal distribution centred at zero. See Fig. 3.

The GeneChip data created by [78] were obtained from NCBI's GEO (dataset GSE3494). Other data, e.g. patients' age, survival time, whether breast cancer caused death and tumour size, were also downloaded. Whilst [78] used the whole dataset, with more than a million inputs, we were keen to avoid over-fitting; therefore, the data were split into independent training and verification datasets. See [57].

5 Summary of GPU Hardware and Programming

5.1 Main Hardware Components of GPUs

Figure 4 shows the major components of a C2050 Tesla card. It is typical of current top-end GPUs. The card is connected to the host personal computer via the PC's PCI express bus. The effective speed of the PC–GPU connection varies both with the GPU and with the motherboard into which they both fit. How to get data into and out of the GPU via the PCI bus is one of the major design decisions in any GPU application. Although PCIe bus speeds have risen in recent years, it appears to have peaked. Recent top-end systems have relied on a hierarchy of PCI interconnects which allow simultaneous parallel transfers along their various parts.

Typical GPUs have space for several hundred megabytes or even a few gigabytes of data. The trend is still very much to increase the speed and size of onboard memory. Again, deciding which application data are stored onboard the GPU

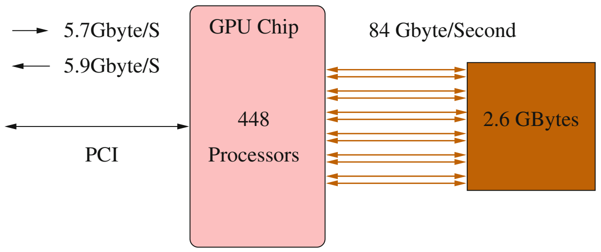


Fig. 4 Links from GPU chip to host computer via PCIe bus and to memory on the GPU board. Fermi C2050 (ECC memory checks turned on)

(and when) is an important design decision. The GPU chip is connected by a very high-speed bus to its own high-speed onboard RAM memory.

There are some two-GPU systems. Typically, although there are two chips and two sets of RAM on the same board, they are programmed as if they were two separate GPUs in the same PC.

It is typical for a single GPU chip to contain more than one multiprocessor; see Fig. 5. These have their own connections to the onboard RAM and act more or less independently in parallel. The number of multiprocessors varies considerably between low-end and older models and high-end GPUs. The C2050 has 14 multiprocessors. There are already GPUs with 16 multiprocessors, and the trend is for the maximum number of multiprocessors to increase whilst retaining low-end GPUs with a single multiprocessor.

The multiprocessors contain banks of stream processors (SPs). These are where the essential SIMD nature of GPU computing arises. All the stream processors are locked together. They do the same calculation at the same time (albeit on different data). Thus, a C2050 multiprocessor can take 32 data items, do 32 calculations and generate 32 answers in parallel. However when a program contains an if or branch instruction, the 32 data items may cause the 32 stream processors to go in different directions. This they cannot do. Instead one branch direction is chosen, and stream processors going in that direction are free to continue calculating. The rest are held. At some time, the freely running stream processors are held long enough for the others to run. It may be quite some time later when all the stream processors return to a common instruction at the same time and all begin running at full speed in synchrony. In the meantime (when the stream processors' paths have diverged) the multiprocessor has been operating correctly but at reduced power. We shall use this property. It is important to remember that GPUs offer cheap computation, so it is okay to waste some of it.

The number of stream processors varies between GPUs. nVidia multiprocessors contain multiples of eight. As with multiprocessors themselves, both the range and maximum number of stream processors have increased and are likely to continue increasing. However the multiprocessor clock speed has not increased and may even have fallen back a little. Typical clocks speeds are now 1.1–1.5 GHz and dramatic change is not likely.

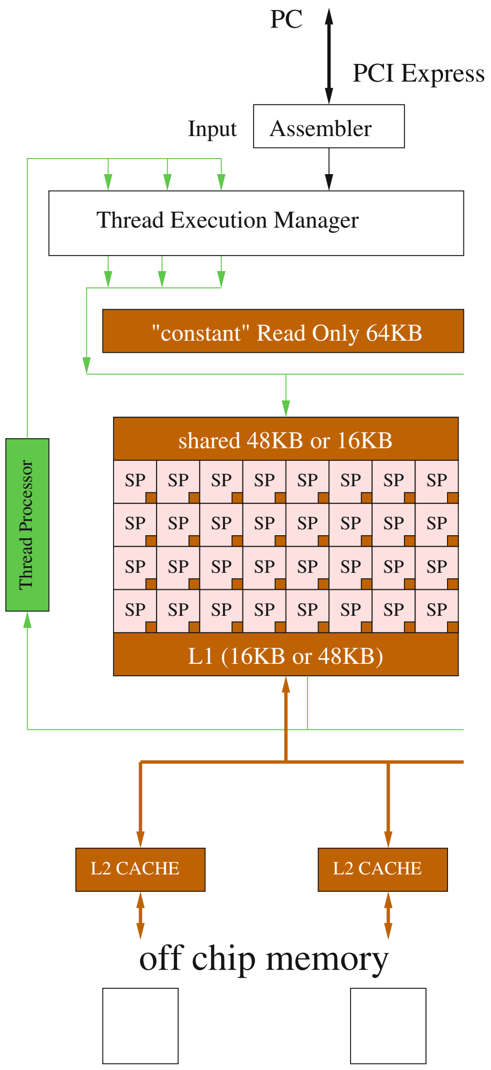


Fig. 5 nVidia GPU multiprocessor with 32 stream processors (SP). The C2050 contains 14 such multiprocessors, giving 448 SPs in total. Each stream processor obeys the same instruction at the same time. However each has its own registers and access to shared and constant memory. The L1 caches coalesce multiple separate accesses to off-chip memory into a single access of 128 bytes each. In default operation each L1 cache occupies 16 Kbytes (giving 128 cache lines); however, the 48 Kbyte shared memory can be reduced to 16 Kbytes to expand the L1 cache to 48 Kbytes

The newer Fermi designs now include both per-multiprocessor (L1) read–write data caches and L2 read–write cache shared between the multiprocessors, whereas older designs relied either on the application designing its own caches or read-only caches provided as part of graphics “texture” memory. The L2 cache also allows some limited communication between multiprocessors via atomic operations. For some time nVidia resisted the application developers’ calls for caches, but now implemented in the Fermi architecture, they seem to be a great success. Future GPUs may see more and/or bigger caches.

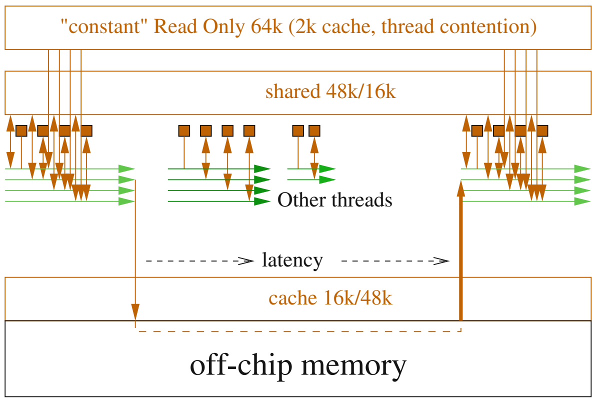


Fig. 6 nVidia CUDA mega threading (Fermi, compute level 2.0). Each thread in a warp (32 threads) executes the same instruction. When a program branches, some threads advance and others are held. This is known as thread divergence (Sect. 5.1). Later the other branches are run to catch up. Only the 32,768 registers (*small squares*) per block can be accessed at full processor speed. If threads in a warp are blocked waiting for off-chip memory (i.e. local, global or texture memory), another warp of threads can be started. The examples assume the requested data are not in a cache. Shared memory and cache can be traded, either 16 Kbytes or 48 Kbytes. Constant memory appears as up to 64 Kbytes via a series of small on-chip caches [2]

5.2 *Memory Latency: Efficiently Programming with Threads*

A significant point in Figs. 4 and 5 which we have not discussed is why caches are important. The fact that dominates GPU programming (even with caches) is that it can take hundreds of times longer to fetch data from the GPU's off-chip memory than to calculate with it. Once data are in its registers, cache or shared memory, the multiprocessor can calculate with it blisteringly fast, but an unfortunate application can perform badly simply by having the stream processors wait for data most of the time. Figure 6 is a schematic which shows the GPU hardware interleaving threads of execution (horizontal arrows) so that as threads are blocked (e.g. waiting for off-chip data to arrive) others are automatically released to run. If there are enough threads, the multiprocessor may still be busy when the data arrives, so keeping it fully loaded and enabling the application to efficiently use the GPU. However the number of active threads is limited.

Earlier nVidia GPUs limited the maximum number of threads to 512. The Fermi architecture has recently doubled this to 1,024. However there is another limit. Each execution thread will need some registers. Unlike a preemptive scheduler on the host computer, when a thread stops, there is nowhere to save these registers when a new thread is scheduled. Thus even when a thread is blocked (e.g. waiting for data to arrive) it cannot release its registers. This enables extremely rapid context switching between threads but means all the multiprocessor's registers have to be shared by its active threads. (A C2050 multiprocessor has 32,768 registers, 1,024 for each stream

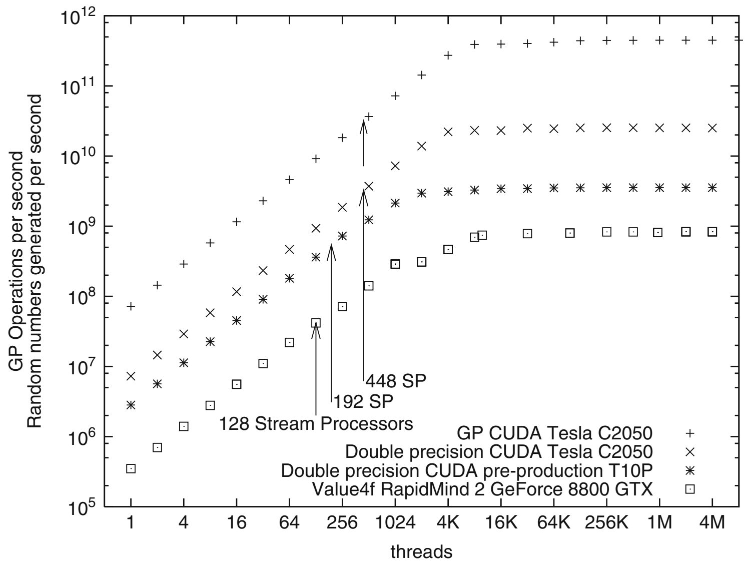


Fig. 7 Speed of genetic programming interpreter [46] and Park–Miller random numbers [44] (excluding host–GPU transfer time) versus number of parallel threads used on a range of nVidia GPUs. *Top* three plots refer to CUDA implementations and *lowest* one to RapidMind code. Code available via http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/random-numbers/cuda_park-miller.tar.gz

processor.) Although the CUDA nvcc compiler is very careful in how it allocates registers, it is possible, in complicated applications, for the number of active threads to be limited by the number of registers each thread requires before reaching the 1,024 limit.

Although GPU and application dependent, Fig. 7 shows that typically a GPU starts to approach its maximum performance when there are more than about 18 threads per stream processor.

6 GeneChip Data Mining Using Genetic Programming on a GPU

6.1 A CUDA Single Instruction Multiple Data Interpreter for GP

Section 3 has listed the previous experiments evolving programs with a GPU. Mostly these have represented the programs either as trees or as networks (Cartesian GP) [29] and used the GPU for fitness evaluation. Harding [29] compiled his networks into GPU programs before transferring the compiled code onto the GPU.

However it turns out to be quite expensive to compile CUDA programs, and so it only makes sense when the program (in our case a GP individual) is to be run many times. (Harding showed the compiled approach can be improved by distributing the compilation across a local area network of workstations and obtained impressive results when each GP program was run more than 100 million times [32].) Since we will be running each GP individual program on each training case (cancer patient) but we have at most only a few hundred training cases (actually only 91), it makes sense to avoid the compilation overhead and accept that interpreting the program may be slower than running compiled code, but interpreting will be faster overall. Therefore, we keep the traditional tree-based GP and use an interpreter running on the GPU.

The host part of the program is a more-or-less traditional GP but with fitness evaluation transferred to the GPU. However it represents evolving genetic programming individuals as trees which are linearised into reverse Polish expressions [53] so that the GPU can interpret them straightforwardly in a single pass without recursive calls. The three mutation operations and crossover act directly on the reverse Polish expressions. This enables them to be passed directly to the GPU without the need to change format between the host and the GPU. Next we shall recap how to interpret multiple programs simultaneously on an SIMD computer [42] before going into the details of the CUDA implementation (Sects. 6.2–6.10). Section 6.11 describes how we use hundreds of GP runs to progressively refine the GeneChip data, how the largest ever GP populations are created and evolve under fitness selection, mutation and crossover. It also describes the non-panmictic fine-grained distributed population and short evolution times used to maintain diversity. All these operations take place on the host PC and are implemented in C source code.

Essentially the interpreter trick is to recognise that in the SIMD model (Sect. 5.1), the "single instruction" belongs to the interpreter and the "multiple data" are the multiple GP trees. The single interpreter is used by millions of programs. It is quite small and needs to be compiled only once. It is loaded onto every stream processor within the GPU. Thus, every clock tick, the GPU can interpret a part of up to 448 different GP trees. The guts of a standard interpreter is traditionally an n-way switch where each case statement executes a different GP opcode; however, Fig. 8 gives an alternative view in which the interpreter works on all possible opcodes and each GP program uses just those that it contains. The CUDA implementation is given in Figs. 9–11.

6.2 CUDA Interpreter for GP

The CUDA code is given in Fig. 9. Potentially it could be improved further: (1) Each program must end in a NOP so the for loop test `PC < LEN-1` could be removed; (2) The array-indexing operation `Pop[PC]` could be replaced by using the pointer `Pop` directly and incrementing it by 4 bytes on each iteration of the loop, which would allow the variable `PC` to be removed.

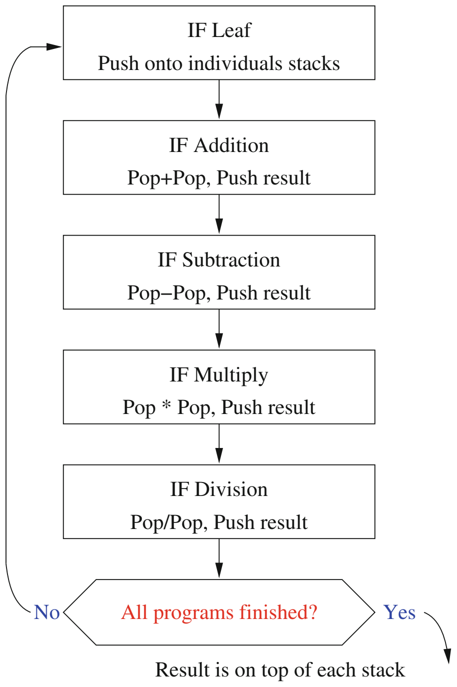


Fig. 8 The original idea for the SIMD interpreter was that it should loop continuously through the whole genetic programming terminal, and function sets with GP individuals select which operations they want as they go past and apply them to their own data and their own stacks. However this can be refined by noting that individual multiprocessors act independently. If all 32 stream processors (SPs) in a warp run the same GP program, they will be synchronised and the SIMD interpreter behaves more like a conventional interpreter acting in parallel 32 times. There is some loss in efficiency if they act on multiple GP individuals and lose synchronisation, since this may cause thread divergence (Sect. 5.1); however, the GPU still performs well

6.3 CUDA Interpreter Stack for GP

The interpreter evaluates each GP tree as a reverse Polish notation expression by pushing and popping intermediate values onto a stack (see Fig. 8). Each expression needs its own stack. Each GPU thread works on its own expression and so needs its own stack.

Since there is no communication between threads, with read–write caches, it might be possible to place the interpreter's stacks in per-thread "local" memory. There is only a little shared memory, whereas there is lots of local memory, but if a cache line holding the stack were displaced, performance would be hit hard.

To avoid the possibility of any stack being moved to off-chip memory, we chose to put them in shared memory. (See code fragment in Fig. 10.) Many GP systems restrict tree depth and function arity. For example, our GP genetic operations ensure

```
int SP = 0;
for(unsigned int PC = 0; PC < LEN-1; PC++) {
  const optype OPCODE = Pop[PC];
  if(OPCODE==OPNOP) break;
  float d;
  if(OPCODE<= lastconst) {
    d = constants[OPCODE];
  } else if(OPCODE<= lastleaf) {
    d = d_Train0[(OPCODE-firstinput)*nexamples];
  } else {
    const float sp1 = stack(--SP);
    const float sp2 = stack(--SP);
    switch(OPCODE) {
    case OPADD:  d = sp2+sp1; break;
    case OPSUB:  d = sp2-sp1; break;
    case OPMUL:  d = sp2*sp1; break;
    case OPDIV:  d = sp2/sp1; break;
    }
  }
  push(d);
}
```

Fig. 9 GPU Reverse Polish Notation SIMD interpreter. The interpreter is invoked by every thread in the block (1,001) in parallel and cycles through each of the programs' instructions leaving the answer generated by each on the programs' stacks. (Fitness calculation in Figs. 13–15.) Notice division is not protected [40]. `Pop` is a pointer to the start of the RPN program which is being evaluated on this stream processor. `d_Train0` points to the data for the current cancer victim (see Sect. 6.6)

```
extern __shared__ float shared_array[];
const int pStackMax = (MaxArity-1)*(pMaxDepth-1)+1;
#define stack(sp) shared_array[(sp)*blockDim.x+threadIdx.x]
#define push(x) {stack(SP) = x; SP++;}
```

Fig. 10 CUDA implementation of stack required by SIMD interpreter (given in Fig. 9). The stack is placed in shared memory to ensure it remains on-chip. CUDA allows indexed access to shared memory and so implementing a stack is much simpler than it was with RapidMind (version 2.0) and using deeper stacks is also straightforward. Indexing by `threadIdx.x` ensures each thread accesses adjacent words of shared memory so there are no bank conflicts

tree depth does not exceed eight (`pMaxDepth`) and Koza [40] enforces a depth limit of 17. If unusually deep trees were needed or the function set contained functions with more than just two inputs (our data-mining trees use binary functions), more memory would be required. In this case the limited shared memory could start to restrict the number of threads that the interpreter can use.

Examining the PTX assembler produced by the nvcc compiler suggests that although accessing shared memory should be almost as fast as the threads' own registers, a surprisingly large number of PTX instructions are needed to implement push and pop. However it is not clear why and also the mapping between PTX assembler and final machine code is far from straightforward. Even though efficient stack operations are vital, this makes further optimisation of the stack tricky.

```
float* const constants = &shared_array[pStackMax*blockDim.x];
for(unsigned int i=threadIdx.x; i<=lastconst; i += blockDim.x){
  constants[i] = float(-5.0) + float(i) * float(0.01);
}
__syncthreads();
```

Fig. 11 Setting up GP constants in shared memory. The 1,001 constants are stored immediately above the interpreter's stack (Figs. 10 and 12). The calculation is spread across all the available threads. `__syncthreads()` prevents any thread moving on to interpret any program until all the constants have been initialised. As the calculation happens before any global data is read, `__syncthreads()` causes little overhead. Usually adjacent threads interpret the same GP individual so they will simultaneously read the same constant. This does not cause a bank conflict. Since there are multiple banks of shared memory, only occasionally will a delay occur as a bank conflict arises from threads in the same warp interpreting two different programs simultaneously

6.4 Constants

In this application the GP system needs 1,001 constants (with values between -5.0 and $+5.0$, every 0.01). To simplify the interpreter, the old RapidMind system precalculated these and loaded them as part of the training data. However pushing constants onto the stack is one of the most common operations, and so to avoid reading them from global data (as has to be done with the training data), originally the new CUDA interpreter calculated them as required. This overhead was reduced by precalculating them once per multiprocessor and saving them in shared memory (see Figs. 11 and 12). This only occupies 4,004 bytes of shared memory, but the speedup was modest.

It would also be possible to store them in constant memory, so avoiding calculating them on the GPU at all, but where two different programs cause the interpreter to simultaneously read different constants, there is a surprising overhead [50, 61].

6.5 Thread Layout

As we described under the heading of "The Computational Cube" in [47], one of the virtues of the SIMD GP interpreter is that it gives different ways to access the huge amount of parallelism inherent in having a population of individuals and multiple training cases on which they need to be evaluated. As we showed in Fig. 7, the efficient use of GPUs requires many active threads. While it will vary between applications, Fig. 7 suggests even something as simple as generating random values will need at least 8,000 threads to fully load a C2050. With this in mind we designed the thread layout to use as many of the 1,024 threads per multiprocessor block as possible. However we decided to combine the fitness calculation with the interpreter into one CUDA kernel so all the threads interpreting one program must be in the

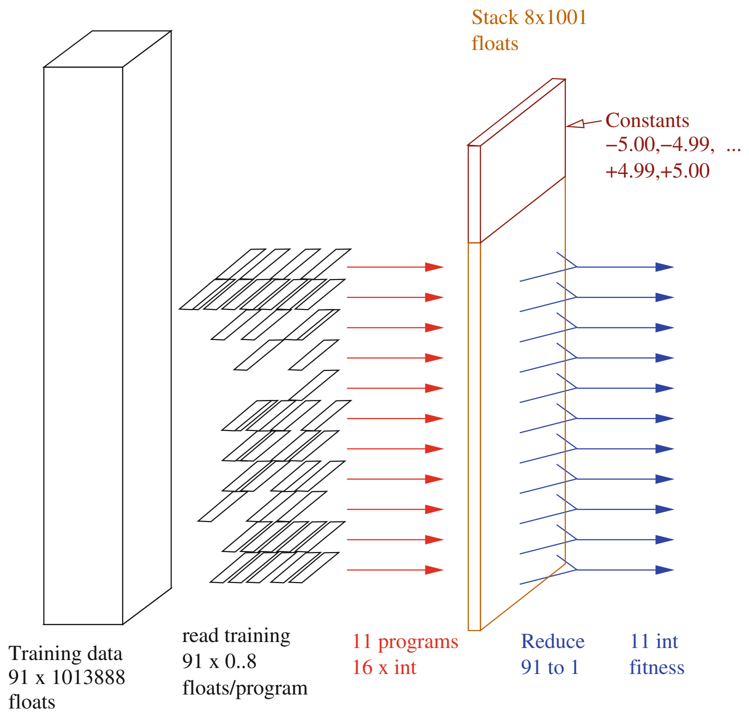


Fig. 12 On each of the 14 C2050 Tesla multiprocessors, 11 GP programs of between 1 and 15 instructions (11 middle arrows) are interpreted in parallel, each processing data for 91 of the breast cancer gene expression datasets. This uses 11 × 91 = 1,001 of the 1,024 available threads (97.8 %). Each interpreter thread has its own stack in shared memory (slab between the two sets of arrows). Apart from warp divergence the 1,001 threads act independently until fitness calculation. After comparing each program's output with the actual class, the CUDA kernel uses seven reduce operations to sum the number of training cases which the program got right and convert these to a fitness value which is written to global memory (11 arrows on right)

same block, and they are forced to synchronise when fitness is calculated. Also we decided to use one thread per GP program per test case. With 91 training cases, this means each block simultaneously interprets 11 programs using 1,001 threads (98 % of the 1,024 maximum). See Fig. 12. This gives a maximum of 14,014 active threads per C2050.

The interpreter threads are tightly packed, which means ignoring the 32 thread warp boundaries [93]. Thus 10 of our 32 warps will be interpreting two GP programs at once and so will suffer from divergence. For 91 training examples, we could have packed the thread into three warps (using 95 % of the available threads), allowing ten programs per block and up to 12,740 active threads per C2050. However, tightly packing the programs into warps has the advantage that the number of training cases can be readily changed without detailed consideration of its impact. For example, the system worked well (without modification) with 41 training cases.

Other approaches are also possible. For example, all 91 fitness cases for one program could be interpreted by warps in the same block. This would simplify the across-thread summations needed to calculate the program's final fitness value and remove the need to use `__syncthreads()` in the fitness reduction (Sect. 6.8). Alternatively we could have used one thread per program, so avoiding the need for any data transfers between threads. This also avoids any idle threads. However as well as problems of the threads diverging (Sect. 5.1), having a large number of separate programs independently requesting uncorrelated data items would overwhelm the data caches.

A potential good compromise would be to allocate each program a whole warp (avoiding thread divergence), enabling it to read and use training data a cache line at a time. Having read and processed it, typically the program would not re-read it. With 91 training cases, each interpreter thread would have to process the program between two and three times. (This also uses 95 % of the available threads.)

As the computational cube approach makes clear, other compromises are possible. While their efficiency will vary according to circumstances, many are viable.

6.6 Training Data

Each training example has data from both HG-U133A and HG-U133B, i.e. $2 \times 712^2 = 1{,}013{,}888$ floats. The training data are not modified. (This is usual in machine learning applications.) They are stored in the GPU (left-hand side of Fig. 12) at the start of the run and then only read. This transfer happens only once so there would only be a marginal advantage in using non-paged ("pinned") memory on the host to speed up the transfer. Once loaded onto the GPU, the host does not use it again. Placing the data in normal host memory allows the operating system to page them out to re-use the RAM they were occupying if need be.

When the training data are read in, they are effectively transposed so that all the data for the same GeneChip probe are placed consecutively. This enables probe data to be read into a few cache lines in a small number of operations (three or four depending upon alignment).

6.7 Thread Divergence

Although all our reverse Polish (RPN) flatten trees will start with pushing a data item, in a usual GP population, the second, third, fourth and so on instruction will tend to be different. As the code in Fig. 9 shows, if a warp of threads interprets two different GP individuals, their paths through the interpreter code will be different and only a small part (the top and bottom of the main loop) will be common. Since this is impossible, we get "divergence" (Sect. 5.1). This means one set of threads proceed, with the others headed to different code, held up. Sometime later the first

```
const unsigned int correct = (pos ^ (stack(0) <= 0)) & 1;
volatile unsigned int *sdata = (unsigned int*) shared_array;
sdata[threadIdx.x] = correct;
```

Fig. 13 Final part of `runprog()` (Fig. 9). The value calculated by GP (on the top of the stack, Fig. 10) is compared with the class of training examples `pos`, converted into a Boolean (was it `correct` or not) and then saved in shared memory, overwriting part of the stack (which is no longer needed)

set of threads is held up and the second set allowed to run. At some later point all the threads in the warp resynchronise. Obviously this is slower than the usual case where the whole warp is interpreting the same GP program. From the computational point of view, we would expect such a warp to take a bit less than twice as long as a single program warp.

Potentially more important is reading data. Two different programs (even though adjacent in the GP population) will typically access different data. In the first set of runs there is a huge volume of training data and reading different parts of it will probably mean they are not in the L1 cache; hence, the threads will have to wait until it can be read into the GPU chip. Hopefully there will be other threads elsewhere on the same multiprocessor ready to run, but even so delays caused by reading data may be more important than thread divergence.

Unfortunately it is difficult to tune the code to get the best from the GPU, and it could need re-tuning for other datasets and problems [50]. Nonetheless, while this may not be the absolute optimum code, we feel it is a good compromise.

6.8 Fitness Calculation

There are three stages to fitness calculation (arrows right-hand side of Fig. 12):

1. Each thread compares the sign of the value calculated by the GP individual with that desired. For the 21 positive cases it should be positive. For the 70 negative cases it should not be positive. The value (0 or 1) is saved in shared memory; see Fig. 13.
2. The 91 `correct` or not values are summed using a reduction technique to give the number of true negatives (`TN`) and number of true positives (`TP`) the GP individual scored. See Fig. 14.
3. A single thread is used to convert (`TN`) and (`TP`) into a single fitness value which is stored in the GP individual's output (see Fig. 15) for later transfer to the host.

Each thread always works on the same training case for each of the $\approx$34,000 GP programs it interprets each generation. Therefore, `pos`[1] (Fig. 13), like `d_Train0`

[1]`pos` is 1 for positive training cases and 0 for negative cases.

```
__device__
void reduce_sum(const unsigned int start, const unsigned int n){
//Ok to overlay on Stack as used syncthreads to ensure all done
volatile unsigned int *sdata = (unsigned int*) shared_array;
const unsigned int tid = threadIdx.x;
const unsigned int top = start+n;

// do reduction in shared mem
//__syncthreads() needed as operate across warp boundaries
if(tid>=start && tid<top) sdata[tid] += fsdata(tid+128,top);
__syncthreads();
if(tid>=start && tid<top) sdata[tid] += fsdata(tid+64,top);
__syncthreads();
if(tid>=start && tid<top) sdata[tid] += fsdata(tid+32,top);
__syncthreads();
if(tid>=start && tid<top) sdata[tid] += fsdata(tid+16,top);
__syncthreads();
if(tid>=start && tid<top) sdata[tid] += fsdata(tid +8,top);
__syncthreads();
if(tid>=start && tid<top) sdata[tid] += fsdata(tid +4,top);
__syncthreads();
if(tid>=start && tid<top) sdata[tid] += fsdata(tid +2,top);
__syncthreads();
if(tid>=start && tid<top) sdata[tid] += fsdata(tid +1,top);
__syncthreads();
}
```

Fig. 14 Reduction code to add `n` items in $\log_2(n)$ steps. It calculates the sum of both `correct` (Fig. 13) negative and positive training examples simultaneously. `__device__` function `fsdata()` ensures the reduction code does not include data from threads running other programs or indeed different classes for the same thread. Totals are left in shared memory index `start`. It will cope with up to 256 negative and 256 positive training cases. Clever use of templates and/or conditional compilation could eliminate operations which are not needed with fewer training cases. Atomic or barrier synchronisation might be an alternative to `__syncthreads()`

```
reduce_sum(start1,n);
__syncthreads();
if(threadIdx.x==start) {
  volatile unsigned int *sdata = (unsigned int*) shared_array;
  const unsigned int TN = sdata[start];
  const unsigned int TP = sdata[start+nneg];
  const int penalty = (TP==0||TN==0)? 0 : 2*npos*nneg;
  *d_Output = 1 + penalty + TP*nneg + TN*npos;
}
```

Fig. 15 `sumfit()` uses `reduce_sum()` (Fig. 14) to give the number of `correct` negative (`TN`) and positive (`TP`) training examples. One thread per program calculates AUROC fitness without division (and keeping integer values) but keeping the same relative weighting of `TN`, `TP` and the penalty (Table 1). To aid debugging `1` is added to ensure fitness is never zero. Finally the thread writes fitness to global memory (`d_Output`)

(Fig. 9), and the boundaries of the negative and positive cases (given by `start1` and `n` in Fig. 15) are calculated once when the thread starts and then are reused.

6.9 Fermi L1 Caches

GP individuals are stored as 16 `unsigned int` (`LEN` = 16). Thus when the first thread interprets the first instruction, it will actually cause the whole individual (`Pop`) to be loaded from off-chip global memory into L1 cache and remain in cache on the multiprocessor until the interpreter finishes with it. Actually since each program occupies only half a cache line, the first instruction can also trigger the loading of `Pop` for the adjacent program. (A C2050 cache line covers 128 contiguous bytes). Since all the threads in a block work on 11 contiguous programs (Fig. 12), they should fit into six cache lines. Eventually all of `Pop` will have to be read, but this is done efficiently, and it does not have to be read more than once by that individual. Notice we also avoid explicitly caching the population in shared memory [93].

As mentioned in Sect. 6.6, the training data are organised to be adjacent to each other, so if one part of a training case is loaded into the multiprocessor L1 cache, then 31 data items in the corresponding training cases are also loaded into the cache at the same time. It appears that with 91 training cases three cache lines per data item are needed. (Perhaps four, depending upon how the cache handles alignment.) Thus in the initial runs where there are thousands or indeed millions of data items, the L1 cache cannot hope to avoid reloading. However all the training data required to interpret each GP individual will be read efficiently into the multiprocessor and it will only be read once by that individual.

In the final run, in which we interpret many millions of GP programs, they read only eight training cases. Since the L1 cache occupies 16 Kbytes, these 728 values of training data will fit into it and so should remain cached. (`Pop` still occupies 80 Mbytes and so now becomes the major data item.) The interpreter in the final run achieves only about 200 million more GPops/s than it does on the large training data runs. This hints that the kernel data I/O is working well and it is operating near the Fermi's computational limit.

6.10 CUDA Gives Improvements

Whereas RapidMind 2.0 imposed a 2^{22} bit addressing limit (i.e. no more than $\approx$4 million items per array) and no more than 16 arrays per GPU, CUDA imposes no such limits. Instead all the GPU's memory is directly addressable. Thus originally the population of five million GP programs had to be split into 20 parts and the training data split into eight or more arrays. Therefore 256,000 GP programs were passed to the GPU (a GTX 8800) which, on average, took slightly less than a second to interpret them and return their fitness values. This had to be done with each of the 20 parts of the population. Now the whole population is passed to the Tesla C2050 in one go, interpreted and five million fitness values returned to the host, in under a second.

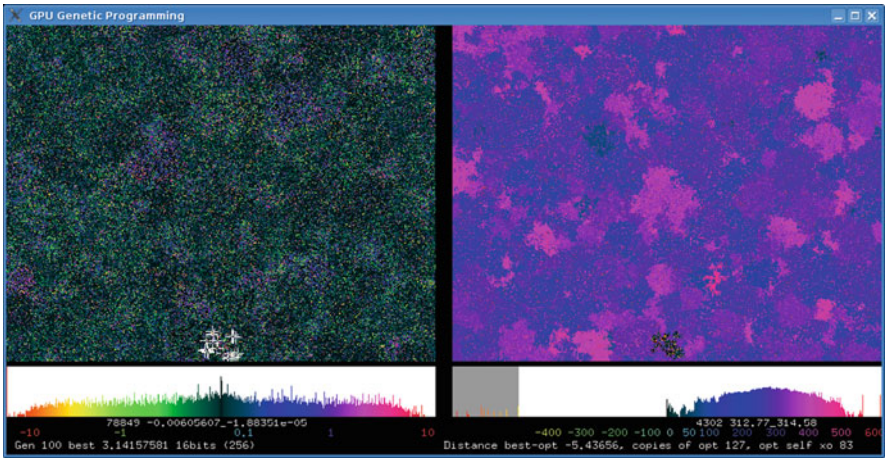
Originally the multiple program outputs (required by splitting the training data into four separate arrays) were summed and combined into a single fitness value per GP individual by three additional GPU programs, making a total of seven GPU programs. Now with the simplification allowed by bigger address ranges, the complete GP interpreter and fitness calculation is done by a single CUDA kernel.

6.11 GP for Large-Scale Data Mining

We previously described using genetic programming to data mine GeneChip data [55]. Our intention was to automatically evolve a simple (possibly non-linear) classifier which uses a few simple inputs to predict the future about 10 years ahead. To ensure the solutions are simple (and for speed), the GP trees are limited to 15 nodes. (Whilst this is obviously small, it is not unreasonable. For example, Yu et al. successfully evolved classifiers limited to only eight nodes [109].) Since many GPUs offer more than a gigabyte of onboard RAM, both the population size and length of individuals could be increased. Indeed since GPUs can now directly access the host computer's RAM, larger populations might be accommodated in large-RAM 64 bit servers without explicit direct transfer to the Tesla. Undoubtedly there will be performance implications, but assuming reasonable locality, so the data caches now available are not overwhelmed, this might be quite a successful approach. However we have not tried this as yet and instead have kept the explicit data transfer. Typically this takes 55 ms (PCIe bandwidth 5.7 Gbytes/s). Explicit transfer of the five million fitness values in the other direction back to the host server takes 3 ms (PCIe bandwidth 5.9 Gbytes/s). Both transfers are to/from non-paged ("pinned") host memory. These large data transfers make the best use of the PCIe bus. If they were replaced by the GPU directly accessing the host ("zero copy" transfers), presumably they would be replaced by data transfers limited to the width of the GPU cache, which might be less efficient. However they would seamlessly allow the GPU to overlap data transfers and computation, whilst we have not attempted such asynchronous use of the GPU.

Previously [55] we demonstrated GP on datasets with more than 7,000 inputs (created by pre-processed raw data). Now we have more than a million individual probe values (and the compute power to use them). Therefore we asked GP to evolve combinations of the probe values rather than use Affymetrix or other human-designed combinations of them. In our approach the first step is to use GP as its own feature selector.

Essentially the idea is to use Price's theorem [89]. Price showed that the number of fit genes in the population will increase each generation and the number of unfit genes will decrease. We run GP 100 times. We ignore the performance of the



```
78849 -0.00605607_-1.88351e-05
Gen 100 best 3.14157581 16bits (256)
4302 312.77_314.58
Distance best-opt -5.43656, copies of opt 127, opt self xo 83
```

Fig. 16 Screen shot of a 512×400 GP population, i.e. 204,800 programs (from runs approximating π [53]) evolving under selection, crossover and subtree mutation after 100 generations. Colour indicates fitness (*left*) and syntax (*right*). *Below* are two histograms (log scale) showing distribution of population by fitness and genotypic distance from the first optimal solution. (Colour scales below each histogram.) Local convergence and the production of species is visible (especially right). See http://www.cs.ucl.ac.uk/staff/W.Langdon/pi2_movie.html and Google videos for animation and more explanation

best-of-run individual and instead look at the genes it contains. Thus the first pass starts with a million inputs, and we select in the region of 10,000 for the second pass and so on until we get down to a reasonable number. Finally GP is run with a much-enriched terminal set containing only inputs which have showed themselves to be highly fit in previous GP runs. See Sect. 7.

The question of how big to make the GP population can be solved by considering the coupon collector problem [21, p. 284]. On average $n(\log(n) + 0.37)$ random trials are needed to collect all of n coupons. Since we are using GP to filter inputs, we insist that the initial random population contains at least one copy of each input. That is, we treat each input as a coupon (so $n = 1{,}013{,}888$) and ask how many randomly chosen inputs must we have in the initial random population to be reasonably confident that we have them all. The answer is 14 million. The spread in the distribution of answers to the coupon collector problem is of the order of square root of n. Therefore if we overshoot by a few thousands, we are sure to get all the inputs (GP tree leafs) into the initial population. Since a program of 15 nodes has eight leafs and half of these are constants, we need at least (1/4)(14 million) = 3.6 million random trees. An initial population of five million ensures this.

At the end of the first pass, we want on the order of 100,000 inputs to chose from. This means we need about 25,000 good programs (each with about four inputs). We do not want to run our GP 25,000 times. The compromise is to use overlapping fine-grained demes [41] to delay convergence of the population; see Fig. 16. The GP population is laid out on a rectangular 2560 × 2048 grid (see Fig. 17). This is

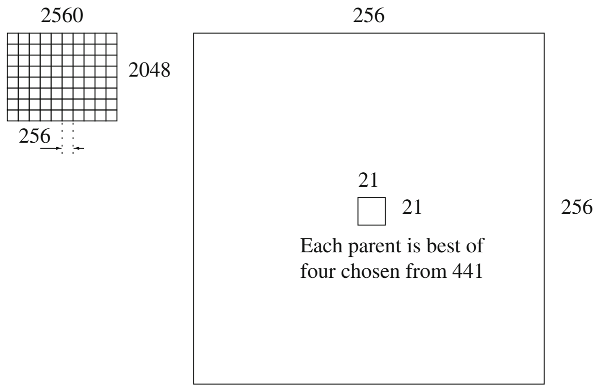


Fig. 17 *Left*: The GP population of five million programs is arranged on a 2560 × 2048 grid, which does not wrap around at the edges. At the end of the run the best individual in each 256×256 tile is recorded. *Right*: (note different scale) parents are drawn by 4-tournament selection from within a 21 × 21 region centred on their offspring

Table 1 GP parameters for data mining Uppsala breast tumour biopsies

Function set	ADD SUB MUL DIV operating on floats
Terminal set	712^2 Affymetrix HG-U133A and 712^2 HG-U133B probe mRNA concentrations. 1,001 constants $-5, -4.99, -4.98, \ldots, 4.98, 4.99, 5$
Fitness	Area under ROC curve (AUROC) $= \left(\frac{1}{2}\frac{\text{TP}}{\text{No. pos}} + \frac{1}{2}\frac{\text{TN}}{\text{No. neg}}\right)$
	Less 1.0 penalty if either all the positive cases or all the negative cases are wrong (TP = 0 or TN = 0) [54]
Selection	Tournament size 4 in overlapping fine-grained 21 × 21 demes [41], non-elitist, population size 2560 × 2048
Initial pop	Ramped half-and-half 1:3 (50 % of terminals are constants)
Parameters	50 % subtree crossover. 50 % mutation (point 22.5 %, constants 22.5 %, subtree 5 %). Max tree size 15, no tree depth limit
Termination	Ten generations

divided into 80 256 × 256 squares. At the end of the run, the genetic composition of the best individual in each square is recorded. Note that to prevent the best of one square invading the next, parents are selected to be within ten grid points of their offspring. Thus genes can travel at most 100 grid points in ten generations. The GP parameters are summarised in Table 1.

7 Experimental Results

The new CUDA system is much faster, but, as expected, the results are similar. GP was run 100 times with all inputs taken from the 91 training examples using the parameters given in Table 1. After ten generations the best program in each of

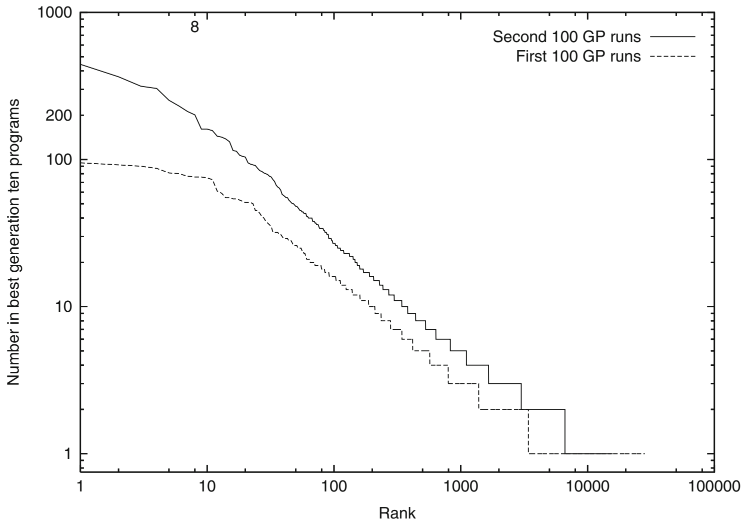


Fig. 18 Distribution of usage of Affymetrix probes in 8,000 best generation-10 GP programs. Both distributions are almost straight lines (note log scales, cf. Zipf's law [114]) and closely agree with earlier runs [57]

the 80 256 × 256 squares was recorded. The distribution of inputs used by these 100 × 80 programs is given in Fig. 18. Most probes were not used by any of the 8,000 programs, 24,892 were used by only one, 2,029 by two and so on.

The 28,305 probes which appeared in any of the 8,000 best generation-ten programs were used in a second pass. In the second pass GP was also run 100 times. (The GP parameters were kept the same.)

Eight probes which appeared in more than 200 of the best 8,000 programs of the second pass were the inputs to a final GP run. (The GP parameters were again kept the same.) See also upper trace in Fig. 18 and Table 2.

As Fig. 19 shows, GP finds many good matches to the 91 training examples, most of the 80 score above 90 % and several score more than 92 %. Ever mindful of overfitting [12], in the original RapidMind runs as a solution we chose one with the fewest inputs (three). GP found a non-linear combination of two PM probes and one MM probe from near the middle of HG-U133A; see Fig. 20 and Table 2. The evolved predictor is the sum of two non-linear combinations of two human genes; see Fig. 21). Both sub-expressions have some predictive ability. The three probes chosen by GP are each highly correlated with all PM probes in their probeset [58] and so can be taken as a true indication of the corresponding gene's activity. The gene names used in Fig. 20 were given by the manufacturer Netaffx's www pages. Possibly terms like decorin/C17orf81 are simply using division as a convenient way to compare two probe values. Indeed the sign indicates whether two values are both above or both below average. (Division appears in all 80 of the best generation-ten programs, slightly more often than + and − but much more often than multiplication: /80, +72, −71, ×27.)

Table 2 Eight Affymetrix probes used the most in 8,000 best generation-10 second pass RapidMind GP programs which were used in the final RapidMind run [57]. See Fig. 18. The second column gives rank in these experiments

		Used	X,Y	chip	Affy id	NetAffx gene title
1	2	579	350,514	A	200903_s_at.mm8	S-adenosylhomocysteine hydrolase
2	10	493	325,511	A	219260_s_at.pm7	C17orf81. chromosome 17 open reading frame 81
3	6	363	254,667	A	201893_x_at.pm2	Decorin
4	1	291	392,213	A	219778_at.pm4	Zinc finger protein, multitype 2
5	4	286	366,310	B	230984_s_at.mm10	230984_s_at was annotated using the accession-mapped cluster-based pipeline to a UniGene identifier using 17 transcript(s). *This assignment is strictly based on mapping accession IDs from the original UniGene design cluster to the latest UniGene design cluster*
6	3	265	324,484	A	216593_s_at.mm9	Phosphatidylinositol glycan anchor biosynthesis, class C
7	19	263	542,192	B	233989_at.mm4	EST from clone 35214, full insert. UniGene ID Build 201 (01 Mar 2007) Hs.594768 NCBI
8	41	245	269,553	B	223818_s_at.pm2	Remodelling and spacing factor 1

The chosen solution compares well with that produced by Miller et al. [78], which used more than 704 data items compared to GP's three. We also showed in [57] that the RapidMind interpreted 535 million GP operations per second (535 MGPop/s). This corresponded to a 7.59× speedup compared to an Intel 2.40 GHz CPU.

8 Genetic Programming Interpreter Speed on Tesla C2050 GPU

On average across the 201 GP runs the C2050 processed 8.5 billion GP primitives per second. This is fairly consistent, even on the last run, where there are only eight inputs (effectively 3 Kbytes of global training data). The server has two C2050 Teslas, so the 100 runs of each phase can be split into two and 50 run on each one. On the four-core server, there is little interaction between them, and so the combined speed of fitness evaluation using two C2050s is 17 billion GPop/s.

The GPU interpreter's performance of 8.5 gigaGPOP/s (line marked ^{o} in Table 3) is very good. It is by far the fastest for a floating point GP data-mining application, being surpassed only by our Boolean multiplexor benchmarks [46], graphics applications [32] and a special bench mark [64]. The number of successful applications has expanded in recent years. Where GP operations rates (rather than just

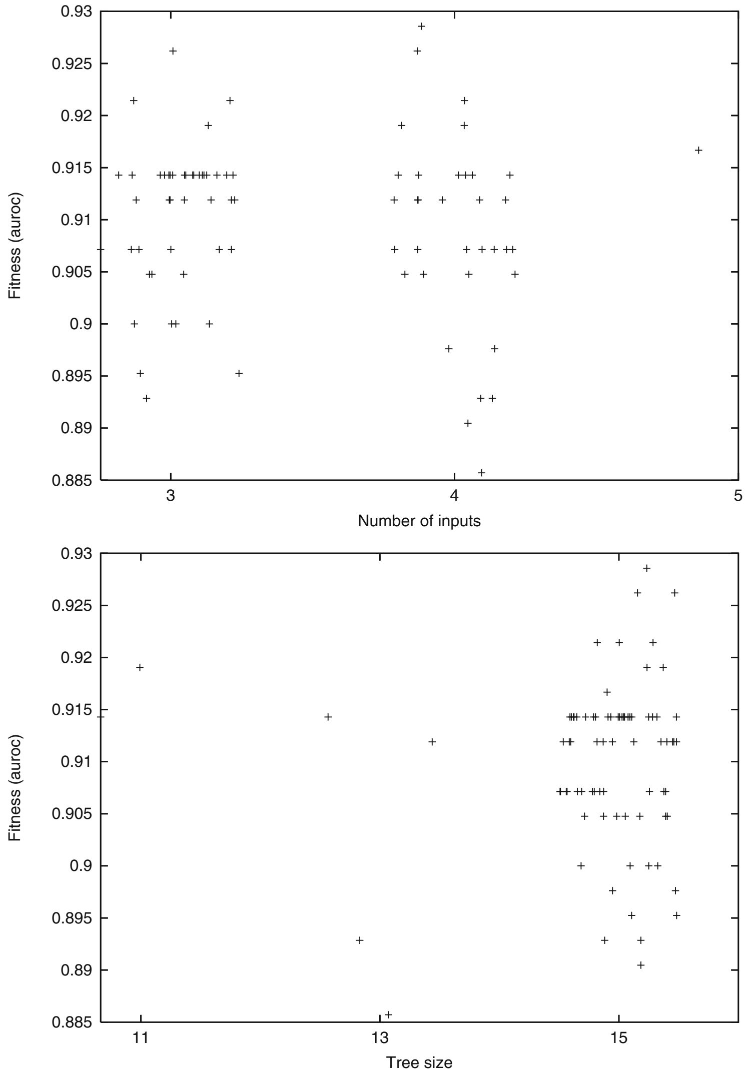


Fig. 19 Spread of performance on training data v. program size. 80 best generation-10 programs in final CUDA GP run with eight inputs. Size is given in *top* graph by the number of different inputs and by the number of GP instructions in the *bottom* graph. Noise added to spread data horizontally. Whilst most of these high fitness predictors are of the maximum size (15) most use only three or four of the eight available inputs

speedup ratios) were given, the result is included in Table 3. Interpreter performance is expected to vary somewhat with the size of the terminal and function sets (columns 2–4) [38]. The performance of compiled GPs on GPUs can vary widely, e.g. with program size (column 6) and number of training examples (column 8).

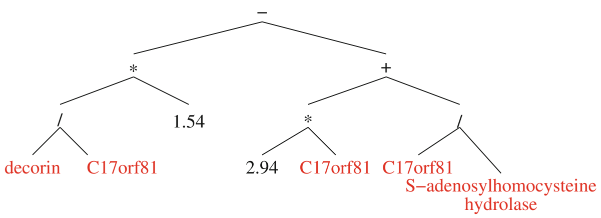


Fig. 20 GP-evolved three-input classifier. The figure uses gene names. We could also use Affymetrix probe names. Using probe names, the evolved tree says survival is predicted if $1.54 \frac{201893\_x\_at.2pm}{219260\_s\_at.7pm} - 2.94\, 219260\_s\_at.7pm - \frac{219260\_s\_at.7pm}{200903\_s\_at.8mm} < 0$

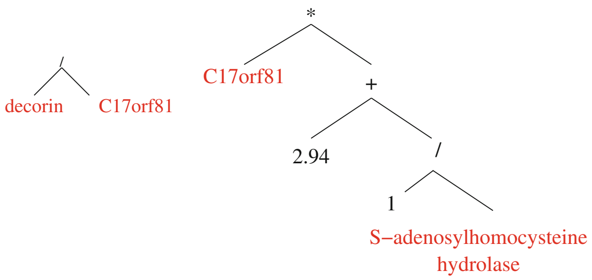


Fig. 21 The GP classifier (Fig. 20) is the weighted addition of two input classifiers (*left* and *right*)

Table 3 gives the maximum speeds; see the individual references for details of which factors affect speed.

Run time in most genetic programming systems is dominated by the time to calculate fitness; now this is done by the GPU, the remaining operations (still done on the host) become more important. Our host code is almost identical to the original RapidMind experiments and has not been optimised. As fitness evaluation speeds up, it may become necessary to parallelise these other parts of the evolutionary process.

The earliest evolutionary computation GPGPU [22] implemented both genetic operations and fitness evaluation on the GPU. More recently Pospichal et al. [87] ported both genetic operations and fitness of grammatical evolution onto a GTX 480 with CUDA.

9 The Future of General Purpose GPU Computing

It is gratifying to note that some of our earlier predictions [57] have already come about. For example, we see more and more on-chip transistors being used to introduce on-chip caches and more on-chip memory. We also see routine support for double precision, removal of the 22 bit limit on data sizes, direct access to host PC RAM, routine support for 64 bit addressing and direct transfer of data between

Table 3 Genetic programming primitives interpreted per second. Unless otherwise noted the GPU was an nVidia GeForce 8800 GTX

Experiment	No. of terminals inputs + consts	Funcs	Pop size	Prog size	Stack depth	Test cases	Speed 10^6 OP/s	GPU
Mackey–Glass	8 + 128	4	204,800	11.0	4	1,200	895	
Mackey–Glass	8 + 128	4	204,800	13.0	4	1,200	1,056	
Mackey–Glass[a]	8 + 128	4	204,800	10.2	4	1,200	1,720	
Protein	20 + 128	4	1,048,576	56.9	8	200	504	
Laser	3 + 128	4	18,225	55.4	8	151,360	656	
Laser	9 + 128	4	5,000	49.6	8	376,640	190	
Sextic[b]	1 + 0	4	100	16	n/a	200	.5	XBox 360
Sextic[c]	1 + 0	8	12,500	70.0	17	100,000	4,073	
Image processing[d]	9 + na	?	2,048	2,048	n/a	$\approx 10^8$	26,200	28×8200
TMBL	? + ?	4	120	300	n/a	65,536	191,724[e]	260 GTX
Multiplexor-6[f]	6 + 0	4	12,500	120.6	17	64	47	
Multiplexor-11[g]	11 + 0	4	12,500	156.2	17	2,048	501	
Multiplexor-20[h]	20 + 0	4	262,144	428.5	15	2,048[i]	254,000	295 GTX
Multiplexor-37[j]	37 + 0	4	262,144	915.6	15	8,192[k]	665,000	295 GTX
GeneChip	47 + 1,001	6	16,384	$\leq$63.0	8	200[l]	314	
Cancer	1,013,888 + 1,001	4	5,242,880	$\leq$15.0	4	128	535[m]	
Cancer[n]	1,013,888 + 1,001	4	5,242,880	12.9	4	91	1,352	C2050
Cancer	1,013,888 + 1,001	4	5,242,880	12.9	4	91	8,517[o]	C2050

Cancer[p]	1,013,888 + 1,001	4	$5,242,880 \times 24$	12.9	4	91	7,140	M2090
Cancer	1,013,888 + 1,001	4	$5,242,880 \times 24$	12.9	4	91	9,943[q]	M2090

[a][93] clusters of ten programs per CUDA block
[b]$x^6 - 2x^4 + x^2$ [104]
[c]$x^6 - 2x^4 + x^2$ [93]
[d][32] "Emboss" image filter evolved with Cartesian GP with distributed nvcc compilation on up to 28 nodes
[e][64] PTX evaluation only
[f][93]
[g][93]
[h][46]
[i]The 2,048 test cases used were randomly sampled from 1,048,576 available every generation
[j][46]
[k]The 8,192 test cases used were randomly sampled from 137,438,953,472 available every generation
[l]The 200 test cases used were randomly sampled from 300,000 available every generation
[m]Interpreter speed only
[n]These runs
[o]These runs. Interpreter speed only
[p][51] Three 8-GPU nodes of the STFC Rutherford Appleton Laboratory Emerald GPU supercomputer, i.e. Total 24×M2090 Tesla
[q]Same runs as [p]. Mean interpreter speed on each of 24 M2090s. That is, the same kernel runs 40 % faster on a shared supercomputer M2090 than it does on our server-mounted C2050s. Over all 201 runs (including data transfers and host operations), the Emerald supercomputer interpreted 33.8×10^9 GP operations per second. This used up to 80 M2090s; however the average performance was reduced by significant scheduling delays due to sharing Emerald with other users [52]

GPU in the same host computer. The concept of GPGPU has broaden out and is directly supported by nVidia's Tesla range (of non-graphics card GPUs). GPGPU continues to grow.

Like the x86 processor range, modern GPU chips are accumulating functionality with the manufacturers showing great reluctance to remove transistors designed to support older graphics applications such as anti-aliasing. This hardware is unlikely to be useful for scientific computing and so represents an overhead.

Published GPGPU computation has been dominated by nVidia. Initial publications were by people programming scientific applications using graphics tools (e.g. Cg). There was then a move to nVidia's CUDA. In the last couple of years there has been a little interest in OpenCL applications on nVidia cards. OpenCL offers the possibility of porting applications between different graphics hardware. Indeed recently some new GPGPU applications [26] have been coded to use OpenCL on ATI cards.

It is clear that GPU programming is aimed squarely at the high-level language programmer. Even the CUDA assembler language PTX is remote from the machine code that the GPU actually runs. Both PTX and high-level language sources must be compiled before the GPU can use them. The compilation tools are aimed at one programmer working on one (or a few) program at a time and aim to produce the very best machine code for the GPU and do not worry about how long it will take to compile. This is fundamentally not suitable for populations of programs. We have worked around this problem. Harding [32] (and now more recently others) ran the CUDA compiler multiple times in parallel. Lewis showed evolving PTX can reduce the compilation overhead by more than 20 % [64]. We have taken the approach of not compiling the GP programs but instead interpreting them. Whilst this allows a single GPU to run millions of programs simultaneously, an interpreter will always be slower than machine code. Nordin [81] was the first to recognise this and built GP systems that both genetically manipulated and ran first SUN machine code and later Intel x86 code. Indeed his x86 system is now the basis of a successful commercial GP system [23]. Whilst GPU machine code is not straightforward [64], we anticipate soon someone will bite the bullet and remove the compiler/interpreter bottleneck by implementing a GP system which evolves GPU machine code directly.

Despite improving tools, both debugging (see [48] and the chapter "Understanding NVIDIA GPGPU Hardware") and performance tuning [50] remain difficult. There is still a risk that if GPUs remain difficult to use, they will remain limited to specialised niches. To quote John Owens, "It's the software, stupid" [82].

10 Conclusions

Previously [57] we took a large GeneChip breast cancer biopsy dataset with more than a million inputs and demonstrated genetic programming running in parallel on an nVidia GeForce 8800 GTS and showed a 7.6× speedup compared to a single-core PC. The compute-intensive fitness evaluation has now been recoded in CUDA and run on modern hardware, the C2050 Tesla. With the new kernel a single nVidia C2050 delivers about 8.5 billion GP operations per second, i.e. 16 times faster than the old code with an 8800 GTS, even though in simple terms of peak floating point (single-precision) performance, the C2050 is just 2.5 times faster.

Two C2050s can deliver 17 GPop/s. This includes interaction times between host and GPU, but not selection, crossover and mutation, which are still done by the original C++ code on the host.

In some ways a genetic programming interpreter is an ideal GPU application. The cross product of the GP population and training case sizes is already huge. If we also include running multiple GP runs in parallel, in these experiments we have 48 billion almost independent calculations which could be done in parallel. Sometimes highly parallel applications can give disappointing results on GPUs because there is little computation per data item and so more time is spent moving data than computing with it. We estimate very roughly 30 machine instructions are needed to interpret each GP primitive. This gives an "arithmetic intensity" (i.e. the ratio of calculations per data item) of about 20, which puts the GP interpreter in the upper part of the typical range of arithmetic intensities of 4–64 FLOP/TDE for successful parallel applications [11, p. 206].

Sections 2 and 3 showed that general-purpose computation on graphics processing units is becoming established, and there are an expanding range of GPGPU applications, particularly in bioinformatics. Today GPGPU is dominated by nVidia's GPUs and CUDA. It may be OpenCL will soon open the way, not to portable GPGPU applications but to more use of ATI and Intel GPU hardware. Undoubtedly the 3GHz ceiling on CPU clocks will mean that the future of computing is parallel and GPGPU will be one of the popular approaches whereby desktop and other applications will exploit parallel hardware.

C++ Source Code

CUDA code can be downloaded via anonymous ftp from `ftp.cs.ucl.ac.uk` or via http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/gpu_gp_cuda.tar.gz. The large dataset GSE3494 can also be downloaded from the UCL ftp site ftp.cs.ucl.ac.uk/genetic/gp-code/GSE3494/.

Acknowledgements I would like to thank Shigeyoshi Tsutsui, Stan Seibert, Neil Daeche (UCL) and Derek Ross (STFC Rutherford Appleton Laboratory). The two C2050s were donated by nVidia as part of the GISMO EPSRC project.

References

1. Arabnia, H.R., Oliver, M.A.: A transputer network for the arbitrary rotation of digitised images. Comput. J. **30**(5), 425–432 (1987)
2. Bakhoda, A., et al.: Analyzing CUDA workloads using a detailed GPU simulator. In: International Symposium on Performance Analysis of Systems and Software, Boston, MA, USA, 2009, pp. 163–174. IEEE (2009)
3. Banzhaf, W., et al.: Genetic Programming. Morgan Kaufmann, Los Altos (1998)
4. Banzhaf, W., et al.: Accelerating genetic programming through graphics processing units. In: Riolo, R.L., et al. (eds.) Genetic Programming Theory and Practice VI, Chap. 15, pp. 229–249. Springer, Ann Arbor (2008)
5. Barrett, T., et al.: NCBI GEO: mining tens of millions of expression profiles—database and tools update. Nucleic Acids Res. **35**(Database issue), D760–D765 (2007)
6. Camargo Bareno, C.I., et al.: Intrinsic evolvable hardware for combinatorial synthesis based on soC + FPGA and GPU platforms. In: Krasnogor, N., et al. (eds.) GECCO Companion, Dublin, 2011, pp. 189–190. ACM, New York (2011)
7. Cano, A., et al.: Solving classification problems using genetic programming algorithms on GPUs. In: Corchado, E., et al. (eds.) Hybrid Artificial Intelligence Systems, San Sebastian, Spain, 2010. Lecture Notes in Computer Science, vol. 6077, pp. 17–26. Springer, Berlin (2010)
8. Cano, A., et al.: Speeding up the evaluation phase of GP classification algorithms on GPUs. Soft Comput. Fusion Found. Methodol. Appl. 187–202 (2011)
9. Charalambous, M., Trancoso, P., Stamatakis, A.: Initial experiences porting a bioinformatics application to a graphics processor. In: Bozanis, P., Houstis, E.N. (eds.) Advances in Informatics, 10th Panhellenic Conference on Informatics, PCI 2005, Volos, Greece, 2005. Lecture Notes in Computer Science, vol. 3746, pp. 415–425. Springer, Berlin (2005)
10. Chitty, D.M.: A data parallel approach to genetic programming using programmable graphics hardware. In: Thierens, D., et al. (eds.) Genetic and Evolutionary Computation Conference, London, 2007, vol. 2, pp. 1566–1573. ACM, New York (2007)
11. Christen, M., Schenk, O., Burkhart, H.: Automatic code generation and tuning for stencil kernels on modern shared memory architectures. Comput. Sci. Res. Dev. **26**(3), 205–210 (2011)
12. Corney, D.P.A.: Intelligent analysis of small datasets for food design. Ph.D. thesis, University College, London (2002)
13. Cupertino, L.F., et al.: Evolving CUDA PTX programs by quantum inspired linear genetic programming. In: Harding, S., et al. (eds.) GECCO 2011 Computational Intelligence on Consumer Games and Graphics Hardware (CIGPU), Dublin, 2011, pp. 399–406. ACM, New York (2011)
14. Dowsey, A.W., Dunn, M.J., Yang, G.-Z.: Automated image alignment for 2D gel electrophoresis in a high-throughput proteomics pipeline. Bioinformatics **24**(7), 950–957 (2008)
15. Ebner, M.: Engineering of computer vision algorithms using evolutionary algorithms. In: Blanc-Talon, J., et al. (eds.) Advanced Concepts in Intelligent Vision Systems, Bordeaux, France, 2009. Lecture Notes in Computer Science, vol. 5807, pp. 367–378. Springer, Berlin (2009)
16. Ebner, M.: Towards automated learning of object detectors. In: Di Chio, C., et al. (eds.) Evolutionary Computation in Image Analysis and Signal Processing, Istanbul, 2010. Lecture Notes in Computer Science, vol. 6024, pp. 231–240. Springer, Berlin (2010)
17. Ebner, M.: Evolving object detectors with a GPU accelerated vision system. In: Tempesti, G., et al. (eds.) International Conference on Evolvable Systems, York, 2010. Lecture Notes in Computer Science, vol. 6274, pp. 109–120. Springer, Berlin (2010)

18. Ebner, M., et al.: Evolution of vertex and pixel shaders. In: Keijzer, M., et al. (eds.) European Conference on Genetic Programming, Lausanne, Switzerland, 2005. Lecture Notes in Computer Science, vol. 3447, pp. 261–270. Springer, Berlin (2005)
19. Faler, W.: Automatic algorithm invention with GPU. In: 28th Chaos Communication Congress, Berlin, 2011, p. ID 4764 (2011)
20. Fan, Z., et al.: GPU cluster for high performance computing. In: Proceedings of the ACM/IEEE SC2004 Conference Supercomputing (2004)
21. Feller, W.: An Introduction to Probability Theory and Its Applications, vol. 1, 2nd edn. Wiley, New York (1957)
22. Fok, K.-L.: et al.: Evolutionary computing on consumer graphics hardware. IEEE Intell. Syst. **22**(2), 69–78 (2007)
23. Francone, F.D.: Discipulus Owner's Manual. Littleton, USA, version 3.0 draft edition (2001)
24. Garland, M., Kirk, D.B.: Understanding throughput-oriented architectures. Commun. ACM **53**(11), 58–66 (2010)
25. Gobron, S., Devillard, F., Heit, B.: Retina simulation using cellular automata and GPU programming. Mach. Vis. Appl. **18**(6), 331–342 (2007)
26. Grewe, D., Lokhmotov, A.: Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation. In: General Purpose Processing on Graphics Processing Units, Newport Beach, CA, USA, 2011. ACM, New York (2011)
27. Harding, S.: Evolution of image filters on graphics processor units using Cartesian genetic programming. In: Wang, J. (ed.) World Congress on Computational Intelligence, Hong Kong, 2008, pp. 1921–1928. IEEE Press, New York (2008)
28. Harding, S.L., Banzhaf, W.: Fast genetic programming and artificial developmental systems on GPUs. In: High Performance Computing Systems and Applications, Canada, 2007, p. 2. IEEE Computer Society, Silver Spring (2007)
29. Harding, S., Banzhaf, W.: Fast genetic programming on GPUs. In: Ebner, M., et al. (eds.) European Conference on Genetic Programming, Valencia, Spain, 2007. Lecture Notes in Computer Science, vol. 4445, pp. 90–101. Springer, Berlin (2007)
30. Harding, S.L., Banzhaf, W.: Distributed genetic programming on GPUs using CUDA. In: Hidalgo, I., et al. (eds.) Workshop on Parallel Architectures and Bioinspired Algorithms, Raleigh, NC, USA, 2009, pp. 1–10. Universidad Complutense de Madrid, Madrid (2009)
31. Harding, S., Banzhaf, W.: Implementing Cartesian genetic programming classifiers on graphics processing units using GPU.NET. In: Harding, S., et al. (eds.) GECCO 2011 Computational Intelligence on Consumer Games and Graphics Hardware (CIGPU), Dublin, 2011, pp. 463–470. ACM, New York (2011)
32. Harding, S.L., Banzhaf, W.: Hardware acceleration for CGP: graphics processing units. In: Miller, J.F. (ed.) Cartesian Genetic Programming, Chap. 8, pp. 231–253. Springer, Berlin (2011)
33. Harding, S.L., et al.: Self-modifying Cartesian genetic programming. In: Thierens, D., et al. (eds.) Genetic and Evolutionary Computation Conference, London, 2007, vol. 1, pp. 1021–1028. ACM, New York (2007)
34. Harvey, N., Luke, R., Keller, J.M., Anderson, D.: Speed up of fuzzy logic through stream processing on graphics processing units. In: Wang, J. (ed.) World Congress on Computational Intelligence, Hong Kong, 2008, pp. 3809–3815. IEEE Press, New York (2008)
35. Howlett, A., et al.: Evolving pixel shaders for the prototype video game subversion. In: The Thirty Sixth Annual Convention of the Society for the Study of Artificial Intelligence and Simulation of Behaviour (AISB'10), De Montfort University, Leicester, UK, 2010. AI & Games Symposium (2010)
36. Hu, T., et al.: Variable population size and evolution acceleration: a case study with a parallel evolutionary algorithm. Genet. Program. Evolvable Mach. **11**(2), 205–225 (2010)
37. Izydorczyk, J., Izydorczyk, M.: Microprocessor scaling: what limits will hold? IEEE Comput. **43**(8), 20–26 (2010)

38. Juille, H., Pollack, J.B.: Parallel genetic programming and fine-grained SIMD architecture. In: Siegel, E.V., Koza, J.R. (eds.) Working Notes for the AAAI Symposium on Genetic Programming, MIT, 1995, pp. 31–37. AAAI, Menlo Park (1995)
39. Kannan, S., Ganji, R.: Porting Autodock to CUDA. In: Sobrevilla, P. (ed.) World Congress on Computational Intelligence, Barcelona, 2010, pp. 3815–3822. IEEE, New York (2010)
40. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
41. Langdon, W.B.: Genetic Programming and Data Structures. Kluwer, Boston (1998)
42. Langdon, W.B.: A SIMD interpreter for genetic programming on GPU graphics cards. Technical Report CSM-470, Department of Computer Science, University of Essex, Colchester, 3 July 2007
43. Langdon, W.B.: Evolving GeneChip correlation predictors on parallel graphics hardware. In: Wang, J. (ed.) World Congress on Computational Intelligence, Hong Kong, 2008, pp. 4152–4157. IEEE Press, New York (2008)
44. Langdon, W.B.: A fast high quality pseudo random number generator for nVidia CUDA. In: Wilson, G. (ed.) CIGPU Workshop at GECCO, Montreal, 2009, pp. 2511–2513. ACM, New York (2009)
45. Langdon, W.B.: Large scale bioinformatics data mining with parallel genetic programming on graphics processing units. In: Fernandez de Vega, F., Cantu-Paz, E. (eds.) Parallel and Distributed Computational Intelligence, Chap. 5, pp. 113–141. Springer, Berlin (2010)
46. Langdon, W.B.: A many threaded CUDA interpreter for genetic programming. In: Esparcia-Alcazar, A.I., et al. (eds.) European Conference on Genetic Programming, Istanbul, 2010. Lecture Notes in Computer Science, vol. 6021, pp. 146–158. Springer, Berlin (2010)
47. Langdon, W.B.: Graphics processing units and genetic programming: an overview. Soft Comput. **15**, 1657–1669 (2011)
48. Langdon, W.B.: Debugging CUDA. In: Harding, S., Langdon, W.B., Wong, M.L., Wilson, G., Lewis, T. (eds.) GECCO 2011 Computational intelligence on consumer games and graphics hardware (CIGPU), Dublin, 2011, pp. 415–422. ACM, New York (2011)
49. Langdon, W.B.: Generalisation in genetic programming. In: Krasnogor, N., et al. (eds.) Genetic and Evolutionary Computation Conference, Dublin, 2011, p. 205. ACM, New York (2011)
50. Langdon, W.B.: Creating and debugging performance CUDA C. In: Fernandez de Vega, F., et al. (eds.) Parallel Architectures and Bioinspired Algorithms, Chap. 1, pp. 7–50. Springer, Berlin (2012)
51. Langdon, W.B.: Initial experiences of the Emerald: e-infrastructure south GPU supercomputer. Research Note RN/12/08, Department of Computer Science, University College London, 2012
52. Langdon, W.B.: Distilling GeneChips with genetic programming on the Emerald GPU supercomputer. SIGEvolution **6**(1), 15–21 (2012)
53. Langdon, W.B., Banzhaf, W.: A SIMD interpreter for genetic programming on GPU graphics cards. In: O'Neill, M., et al. (eds.) European Conference on Genetic Programming. Lecture Notes in Computer Science, vol. 4971, Naples, 2008, pp. 73–85. Springer, Berlin (2008)
54. Langdon, W.B., Barrett, S.J.: Genetic programming in data mining for drug discovery. In: Ghosh, A., Jain, L.C. (eds.) Evolutionary Computing in Data Mining, Chap. 10, pp. 211–235. Springer, Berlin (2004)
55. Langdon, W.B., Buxton, B.F.: Genetic programming for mining DNA chip data from cancer patients. Genet. Program. Evolvable Mach. **5**(3), 251–257 (2004)
56. Langdon, W.B., Harman, M.: Evolving a CUDA kernel from an nVidia template. In: Sobrevilla, P. (ed.) World Congress on Computational Intelligence, Barcelona, 2010, pp. 2376–2383. IEEE, New York (2010)
57. Langdon, W.B., Harrison, A.P.: GP on SPMD parallel graphics hardware for mega bioinformatics data mining. Soft Comput. **12**(12), 1169–1183 (2008)
58. Langdon, W.B., Harrison, A.P., Sanchez Graillet, O.: RNAnet a map of human gene expression. In: EMBO-2008, Heidelberg, 2008. Abstract presented

59. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer, Berlin (2002)
60. Langdon, W.B., Upton, G.J.G., da Silva Camargo, R., Harrison, A.P.: A survey of spatial defects in Homo Sapiens Affymetrix GeneChips. IEEE/ACM Trans. Comput. Biol. Bioinform. **7**(4), 647–653 (2009)
61. Langdon, W.B., Yoo, S., Harman, M.: Formal concept analysis on graphics hardware. In: Napoli, A., Vychodil, V. (eds.) The Eighth International Conference on Concept Lattices and Their Applications, Nancy, France, 2011, pp. 413–416 (2011) [INRIA Nancy and LORIA]
62. Lewis, T.E., Magoulas, G.D.: Strategies to minimise the total run time of cyclic graph based genetic programming with GPUs. In: Raidl, G., et al. (eds.) Genetic and Evolutionary Computation Conference, Montreal, 2009, pp. 1379–1386. ACM, New York (2009)
63. Lewis, T.E., Magoulas, G.D.: Identifying similarities in TMBL programs with alignment to quicken their compilation for GPUs. In: Harding, S., et al. (eds.) GECCO 2011 Computational Intelligence on Consumer Games and Graphics Hardware (CIGPU), Dublin, 2011, pp. 447–454. ACM, New York (2011)
64. Lewis, T.E., Magoulas, G.D. TMBL kernels for CUDA GPUs compile faster using PTX. In: Harding, S., et al. (eds.) GECCO 2011 Computational Intelligence on Consumer Games and Graphics Hardware (CIGPU), Dublin, 2011, pp. 455–462. ACM, New York (2011)
65. Lindblad, F., et al.: Evolving 3D model interpretation of images using graphics hardware. In: Fogel, D.B., et al. (eds.) Conference on Evolutionary Computation, 2002, pp. 225–230. IEEE Press, New York (2002)
66. Liu, B., et al.: Approximate probabilistic analysis of biopathway dynamics. Bioinformatics **28**(11), 150–1516 (2012)
67. Liu, C.-M., et al.: SOAP3: ultra-fast GPU-based parallel alignment tool for short reads. Bioinformatics **28**(6), 878–879 (2012)
68. Liu, W., et al.: Bio-sequence database scanning on a GPU. In: International Parallel and Distributed Processing Symposium, Rhodes, Greece, 2006. IEEE Press, New York (2006)
69. Liu, Y., Suvranu, D.: CUDA-based real time surgery simulation. Stud. Health Technol. Inform. **132**, 260–262 (2008)
70. Loviscach, J., Meyer-Spradow, J.: Genetic programming of vertex shaders. In: Chover, M., et al. (eds.) Proceedings of EuroMedia 2003, University of Plymouth, UK, 2003, pp. 29–31 (2003)
71. Luo, Z., Liu, H., Wu, X.: Artificial neural network computation on graphic process unit. In: International Joint Conference on Neural Networks, 2005, vol. 1, pp. 622–626. IEEE, New York (2005)
72. Luong, T.V., Melab, N., Talbi, E.-G.: Parallel hybrid evolutionary algorithms on GPU. In: Sobrevilla, P. (ed.) World Congress on Computational Intelligence, Barcelona, 2010, pp. 2734–2741. IEEE, New York (2010)
73. Maitre, O., et al.: Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA. In: Raidl, G., et al. (eds.) Genetic and Evolutionary Computation Conference, Montreal, 2009, pp. 1403–1410. ACM, New York (2009)
74. Maitre, O., et al.: Fast evaluation of GP trees on GPGPU by optimizing hardware scheduling. In: Esparcia-Alcazar, A.I., et al. (eds.) European Conference on Genetic Programming. Lecture Notes in Computer Science, vol. 6021, Istanbul, 2010, pp. 301–312. Springer, Berlin (2010)
75. Maitre, O., et al.: EASEA parallelization of tree-based genetic programming. In: Sobrevilla, P. (ed.) World Congress on Computational Intelligence, Barcelona, 2010, pp. 1997–2004. IEEE, New York (2010)
76. Manavski, S., Valle, G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith–Waterman sequence alignment. BMC Bioinformatics **9**(Suppl. 2), S10 (2008)
77. Meyer-Spradow, J., Loviscach, J.: Evolutionary design of BRDFs. In: Chover, M., et al. (eds.) Eurographics 2003 Short Paper Proceedings, pp. 301–306 (2003)
78. Miller, L.D., et al.: An expression signature for p53 status in human breast cancer predicts mutation status, transcriptional effects, and patient survival. Proc. Natl. Acad. Sci. USA **102**(38), 13550–13555 (2005)

79. Moore, G.E.: Cramming more components onto integrated circuits. Electronics **38**(8), 114–117 (1965)
80. Munawar, A., et al.: Hybrid of genetic algorithm and local search to solve MAX-SAT problem using nvidia CUDA framework. Genet. Program. Evolvable Mach. **10**(4), 391–415 (2009)
81. Nordin, P.: A compiling genetic programming system that directly manipulates the machine code. In: Kinnear Jr., K.E. (ed.) Advances in Genetic Programming, Chap. 14, pp. 311–331. MIT Press, Cambridge (1994)
82. Owens, J.: Experiences with GPU computing, 2007. Presentation slides
83. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Comput. Graph. Forum **26**(1), 80–113 (2007)
84. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proc. IEEE **96**(5), 879–899 (2008) [Invited paper]
85. Pedemonte, M., e al.: Bitwise operations for GPU implementation of genetic algorithms. In: Harding, S., et al. (eds.) GECCO 2011 Computational Intelligence on Consumer Games and Graphics Hardware (CIGPU), Dublin, 2011, pp. 439–446. ACM, New York (2011)
86. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk (2008) [With contributions by J.R. Koza]
87. Pospichal, P., et al.: Acceleration of grammatical evolution using graphics processing units: computational intelligence on consumer games and graphics hardware. In: Harding, S., et al. (eds.) GECCO 2011 Computational Intelligence on Consumer Games and Graphics Hardware (CIGPU), Dublin, 2011, pp. 431–438. ACM, New York (2011)
88. Prabhu, R.D.: SOMGPU: an unsupervised pattern classifier on graphical processing unit. In: Wang, J. (ed.) World Congress on Computational Intelligence, Hong Kong, 2008, pp. 1011–1018. IEEE Press, New York (2008)
89. Price, G.R.: Selection and covariance. Nature **227**, 520–521 (1970)
90. Reggia, J., et al.: Development of a large-scale integrated neurocognitive architecture—part 2: design and architecture. Technical Report TR-CS-4827, UMIACS-TR-2006-43, University of Maryland, USA, October 2006
91. Ribeiro, B., Lopes, N., Silva, C.: High-performance bankruptcy prediction model using graphics processing units. In: Sobrevilla, P. (ed.) World Congress on Computational Intelligence, Barcelona, 2010, pp. 2210–2216. IEEE, New York (2010)
92. Robilliard, D., et al.: Population parallel GP on the G80 GPU. In: O'Neill, M., et al. (eds.) European Conference on Genetic Programming, Naples, 2008. Lecture Notes in Computer Science, vol. 4971, pp. 98–109. Springer, Berlin (2008)
93. Robilliard, D., et al.: Genetic programming on graphics processing units. Genet. Program. Evolvable Mach. **10**(4), 447–471 (2009)
94. Rouhipour, M., et al.: Systemic computation using graphics processors. In: Tempesti, G., et al. (eds.) International Conference on Evolvable Systems, York, 2010. Lecture Notes in Computer Science, vol. 6274, pp. 121–132. Springer, Berlin (2010)
95. Sato, M., Sato, Y., Namiki, M.: Acceleration experiment of genetic computations for Sudoku solution on multi-core processors. In: Blum, C. (ed.) GECCO Late Breaking Abstracts, Dublin, 2011, pp. 823–824. ACM, New York (2011)
96. Sitthi-amorn, P., et al.: Genetic programming for shader simplification. ACM Trans. Graph. **30**(6), article:152 (2011) [Proceedings of ACM SIGGRAPH Asia 2011]
97. Soca, N., et al.: PUGACE, a cellular evolutionary algorithm framework on GPUs. In: Sobrevilla, P. (ed.) World Congress on Computational Intelligence, Barcelona, 2010, pp. 3891–3898. IEEE, New York (2010)
98. Stamatakis, A.: RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. Bioinformatics **22**(21), 2688–2690 (2006)
99. Trapnell, C., Schatz, M.C.: Optimizing data intensive GPGPU computations for DNA sequence alignment. Parallel Comput. **35**(8–9), 429–440 (2009)

100. Unemi, T.: SBArt4—breeding abstract animations in realtime. In: World Congress on Computational Intelligence, Barcelona, Spain, 2010. IEEE Press, New York (2010)
101. Vouzis, P.D., Sahinidis, N.V.: GPU-BLAST: using graphics processors to accelerate protein sequence alignment. Bioinformatics **27**(2), 182–188 (2011)
102. Wilson, G., Banzhaf, W.: Linear genetic programming GPGPU on Microsoft's Xbox 360. In: Wang, J., (ed.) World Congress on Computational Intelligence, Hong Kong, 2008, pp. 378–385. IEEE Press, New York (2008)
103. Wilson, G.C., Banzhaf, W.: Deployment of CPU and GPU-based genetic programming on heterogeneous devices. In: Esparcia, A.I., et al. (eds.) GECCO Workshop on Computational Intelligence on Consumer Games and Graphics Hardware (CIGPU-2009), Montreal, 2009, pp. 2531–2538. ACM, New York (2009)
104. Wilson, G., Banzhaf, W.: Deployment of parallel linear genetic programming using GPUs on PC and video game console platforms. Genet. Program. Evolvable Mach. **11**(2), 147–184 (2010)
105. Wilson, G., Harding, S.: WCCI 2008 special session: computational intelligence on consumer games and graphics hardware (CIGPU-2008). SIGEvolution **3**(1), 19–21 (2008)
106. Wirawan, A., Kwoh, C., Hieu, N., Schmidt, B.: CBESW: sequence alignment on the Playstation 3. BMC Bioinformatics **9**(1), 377 (2008)
107. Wong, M.L.: Parallel multi-objective evolutionary algorithms on graphics processing units. In: Genetic and Evolutionary Computation Conference, Montreal, 2009, pp. 2515–2522. ACM, New York (2009)
108. Yoo, S.: Evolving human competitive spectra-based fault localisation techniques. Research Note RN/12/03, Department of Computer Science, University College, London, UK, 2012
109. Yu, J., et al.: Feature selection and molecular classification of cancer using genetic programming. Neoplasia **9**(4), 292–303 (2007)
110. Yudanov, D., Shaaban, M., Melton, R., Reznik, L.: GPU-based implementation of real-time system for spiking neural networks. In: Sobrevilla, P., (ed.) World Congress on Computational Intelligence, Barcelona, 2010, pp. 2143–2150. IEEE, New York (2010)
111. Yung, L.S., Yang, C., Wan, X., Yu, W.: GBOOST: a GPU-based tool for detecting gene-gene interactions in genome-wide case control studies. Bioinformatics **27**(9), 1309–1310 (2011)
112. Zhou, J., Liu, X., Stones, D.S., Xie, Q., Wang, G.: MrBayes on a graphics processing unit. Bioinformatics **27**(9), 1255–1261 (2011)
113. Zhou, Y., Liepe, J., Sheng, X., Stumpf, M.P.H., C. Barnes: GPU accelerated biochemical network simulation. Bioinformatics **27**(6), 874–876 (2011)
114. Zipf, G.K.: Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology. Addison-Wesley, Cambridge (1949)

GPU-Accelerated High-Accuracy Molecular Docking Using Guided Differential Evolution

Martin Simonsen, Mikael H. Christensen, René Thomsen, and Christian N.S. Pedersen

Abstract The objective in molecular docking is to determine the best binding mode of two molecules *in silico*. A common application of molecular docking is in drug discovery where a large number of ligands are docked into a protein to identify potential drug candidates. This is a computationally intensive problem especially if the flexibility of the molecules is taken into account. We show how MolDock, which is a high-accuracy method for flexible molecular docking using a variant of differential evolution, can be parallelised on both CPU and GPU. The methods presented for parallelising the workload result in an average speedup of 3.9× on a four-core CPU and 27.4× on a comparable CUDA-enabled GPU when docking 133 ligands of different sizes. Furthermore, the presented parallelisation schemes are generally applicable and can easily be adapted to other flexible docking methods.

1 Introduction

In a modern drug discovery process, *in silico* identification of ligands (small molecules) which bind to a target protein is often used in the search for novel drug candidates. Such ligands are likely to change the function of the target protein and

---

This work was done while the author "Martin Simonsen" was affiliated to the Bioinformatics Research Centre (BiRC), Aarhus University, Aarhus, Denmark.

M. Simonsen (✉) · M.H. Christensen · R. Thomsen
CLC Bio, Finlandsgade 10–12, Katrinebjerg, DK-8200 Aarhus N, Denmark
e-mail: msimonsen@clcbio.com; mchristensen@clcbio.com; rthomsen@clcbio.com

C.N.S. Pedersen
Bioinformatics Research Centre (BiRC), Aarhus University, C. F. Møllers Allé 8, DK-8000 Aarhus C, Denmark
e-mail: cstorm@birc.au.dk

may therefore be used to design new pharmaceuticals. From a known 3D structure of a target protein, virtual screening methods can be used to search large ligand databases and create a ranked list of drug candidates. By focusing subsequent *in vitro* experiments on the top ranked ligands, the cost of experimental testing can be reduced.

A common virtual screening approach is to use molecular docking methods to identify binding modes between a ligand and a protein. Molecular docking methods use scoring functions to identify likely binding modes. Rigid docking methods consider only the translation and orientation of the ligand relative to the protein, while the shape of both molecules is fixed during the docking process. In flexible molecular docking the conformation, i.e. the internal structure, of at least one of the molecules is allowed to change. It is common to consider only ligand flexibility while the protein is kept rigid to reduce the number of parameters which have to be optimised. Heuristic methods such as simulated annealing [4, 7], ant colony optimisation [2, 8] and evolutionary algorithms [16, 18] are commonly used to efficiently sample the large search space in flexible docking.

In molecular docking experiments the accuracy of the method is a primary concern but performance in terms of running time per ligand plays a significant role in virtual screening, where many ligands have to be docked into one or more target proteins. Docking methods that take molecular flexibility into account typically use several minutes per ligand, which makes it necessary to employ distributed computing for large virtual screening experiments. One way of reducing the need for expensive hardware in virtual screening is to accelerate molecular docking methods by taking advantage of modern general purpose GPUs.

A first application of GPUs to accelerate molecular docking is [9] where shaders are used to compute the scoring function of PLANTS [8] which is a flexible molecular docking method. Another application of GPUs to flexible docking is [6] where both the genetic algorithm (GA) and scoring function used in AutoDock [3] are accelerated. GPUs are also used in [17] to accelerate the scoring function of PIPER [10] which performs rigid docking using fast Fourier transforms. Other applications of GPUs in molecular modelling are reviewed in [15].

In our work, we explore parallelisation on CPUs and GPUs of MolDock [18] which is a flexible molecular docking method. We present parallelisation schemes for both the differential evolution (DE) algorithm and scoring function used in MolDock which are also applicable to other flexible docking methods. Experiments with a GPU-accelerated version of MolDock show an average speedup of 27.4$\times$ for docking a ligand into a protein compared to a similar implementation running on a single core of a CPU. Compared to previous work in this area, the GPU parallelisation scheme presented here results in a better utilisation of GPU cores and hence a better speedup.

The rest of this chapter is organised as follows: In Sect. 2, we introduce the basic concepts of flexible molecular docking. In Sect. 3, we present the MolDock method for flexible molecular docking, its score function and guided DE. In Sect. 4, we

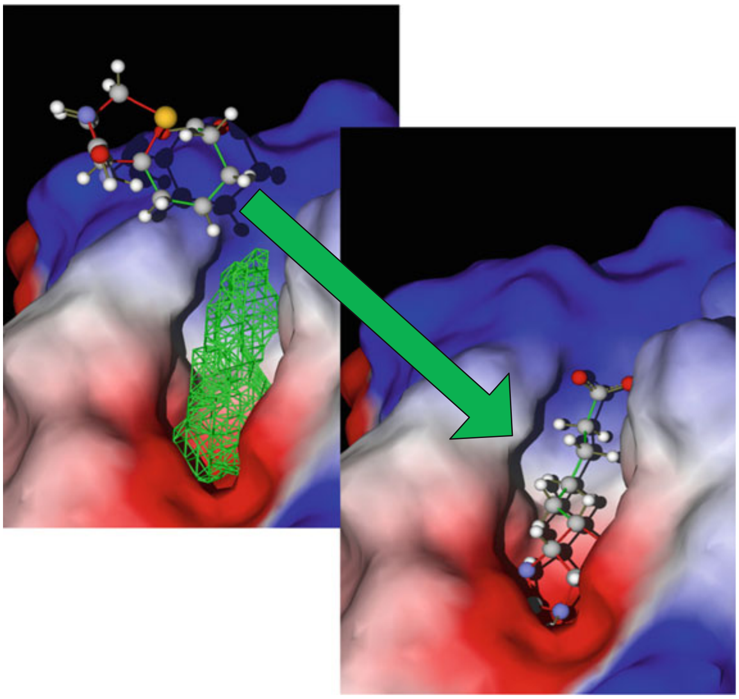


Fig. 1 Illustration of a ligand outside a protein cavity and a docking of the ligand in the cavity

present our parallelisation schemes for the DE and score function of MolDock using CUDA. In Sect. 5, we present the obtained experimental results.

2 Flexible Molecular Docking

We model ligands as one or more rigid structures of atoms connected by rotatable covalent bonds, where each bond has a dihedral angle associated with it. The dihedral angels define the ligand conformation and by allowing them to change, the flexibility of the ligand is taken into account. With a flexible ligand and a rigid protein, the optimal binding mode is found by optimising the translation, orientation and configuration of dihedral angles of the ligand with respect to a scoring function. Figure 1 shows an example of docking a ligand into a protein cavity.

A scoring function is needed to estimate the interaction strengths and internal ligand energy. The most common approach is to use energy force fields which model molecular energy forces such as electrostatic and van der Waals forces that compute the interaction energy between the ligand and the target protein by summing over the energy contributions from each pair of atoms in the ligand and protein [5, 12].

The number of atoms in the protein is usually considerably larger than in the ligand, and precomputed energy grids are often used to speed up energy computations. Energy grids are constructed for each atom type by placing a grid over the protein and computing the energy contribution from all atoms in the protein for each point in the grid. The energy of a ligand atom can then be obtained by interpolating between the grid points closest to the position of the atom.

3 MolDock

MolDock [18] is a method for performing high-accuracy flexible molecular docking which uses a variant of DE called *guided* DE as search heuristic. The method takes full ligand flexibility into account, while the protein remains rigid during the docking process.

3.1 Guided Differential Evolution

The guided DE algorithm used in MolDock is based on the original DE optimisation method by Storn and Price [16] but uses knowledge of cavities in the target protein to constrain the search space. As in the original DE method, guided DE starts out with an initial population of candidate solutions (individuals). In MolDock, each solution is a parameter vector which describes a pose (candidate binding mode). The vectors each contain a 3D translation, an orientation (a rotation by an angle around a unit vector) and a dihedral angle for each rotatable bond in the ligand. Initially, all parameters are set to random values and the corresponding poses are evaluated using the scoring function described in Sect. 3.2. The parameters in each vector are then optimised with respect to the scoring function as follows: Trial vectors (offspring) for each individual are created in a *mutation* and *crossover* step. These vectors are then evaluated using the scoring function and used to update the population in a *selection* step. During selection each vector in the population is replaced with the corresponding trial vector if the trial vector resulted in a better score. An outline of the generic DE algorithm and the offspring creation scheme is shown in Fig. 2.

The difference between DE and guided DE lies in the way translation parameters for trial vectors are computed. Binding sites are often located in cavities in the protein and this knowledge is used in guided DE to restrict the translation of poses as follows. If all atoms in a pose are positioned outside cavity areas, the pose is translated such that a random ligand atom is placed inside the cavity area. Cavities are predicted as described in [18] and represented as points in a 3D grid with a resolution of 0.8Å where each grid point has a Boolean value indicating if the point

```
procedure Differential Evolution
    initialise population with random individuals
    evaluate individuals
    while (not termination-condition)
        for ($i = 0$; $i < popsize$; $i$++)
            create offspring $O[i]$ from parent $P[i]$ by procedure below
            evaluate offspring $O[i]$
            if (offspring $O[i]$ is better than parent $P[i]$)
                replace parent with offspring
            else
                keep parent in population
            end if
        end for
    end while

procedure create offspring $O[i]$ from parent $P[i]$
    randomly select parents $P[i_1]$, $P[i_2]$, $P[i_3]$, where $i_1 \neq i_2 \neq i_3 \neq i$
    for ($j = 0$; $j < n$; $j$++)
        if ($U(0,1) < CR$)
            $O[i][j] = P[i_1][j] + F \times (P[i_2][j] - P[i_3][j])$
        else
            $O[i][j] = P[i][j]$
        end if
    end for
```

Fig. 2 Pseudo-code for the generic DE algorithm. (The function $U(0, 1)$ is a uniformly distributed value between 0 and 1, CR is the crossover rate, F is the scaling factor and n is the number of problem parameters)

is in a cavity or not. If no cavities are detected, pose positions are restricted to a cubic search volume.

3.2 Scoring Function

The scoring function used by MolDock to estimate the interaction energy of poses is defined as

$$E_{\text{score}} = E_{\text{inter}} + E_{\text{intra}}. \tag{1}$$

E_{inter} is the intermolecular energy, i.e. the interaction energy between heavy atoms in the ligand and protein. It is computed as

$$E_{\text{inter}} = \sum_{i \in \text{ligand}} \sum_{j \in \text{protein}} \left(E_{\text{PLP}}(r_{ij}) + 332.0 \frac{q_i q_j}{4 r_{ij}^2} \right). \tag{2}$$

The PLP term, E_{PLP}, is a piecewise linear potential which given the distance between two atoms, r_{ij}, approximates the van der Waals forces and the potential energy of hydrogen bonds between atoms (see [18] for further details). The second term in the summation is a Coulomb potential with a distance-dependent dielectric constant which approximates the electrostatic energy between two atoms, where q_i and q_j are the electric charges of the two atoms. In this work, E_{inter} is computed using trilinear interpolation on the precomputed energy grids.

E_{intra} is the intramolecular energy of the ligand, i.e. the internal energy of the ligand which is computed as

$$E_{\text{intra}} = \sum_{i \in \text{ligand}} \sum_{j \in \text{ligand}} \left(E_{\text{PLP}}(r_{ij}) + E_{\text{clash}}(r_{ij})\right) + \sum_{\theta \in \text{rotatable bonds}} A\left[1 - \cos(m \cdot \theta - \theta_0)\right]. \tag{3}$$

The double summation runs over all heavy atom pairs in the ligand which are more than two bonds apart. E_{clash} is a fixed penalty of 1,000 given to atom pairs that have a mutual distance of less than 2.0Å to prevent unrealistic conformations of the ligand. The second summation is the torsional energy of all rotatable bonds in the ligand, where the A, θ_0 and m parameters are set as described in [18] and θ is the dihedral angle for a given rotatable bond.

The original MolDock scoring function also takes the directionality of hydrogen bonds into account by scaling the PLP contribution of hydrogen bonds according to the angles between atoms involved in the bond. However, this requires an iteration over all protein atoms involved in hydrogen bonds with ligand atoms. In this work, the directionality of hydrogen bonds is therefore not taken into account.

3.3 *Flexible Docking with MolDock*

Initially a ligand and protein is loaded into memory and preprocessed. The preprocessing involves detecting all rigid structures and rotatable bonds in the ligand, cavity detection in the protein and computation of energy grids. Additionally, a search space for the DE algorithm needs to be created, but this cannot always be done without user interaction. In this work we used the centre of the experimentally known ligand as the centre for a search space with size 30 Å × 30 Å × 30 Å.

The next step is energy minimisation using guided DE and the MolDock scoring function. Because of the stochastic nature of DE algorithms, multiple energy minimisation runs increase the chance of finding the lowest scoring binding mode. In [18] ten runs were performed for each ligand and after each run the solution with the best score was stored. In the original implementation, an additional final step re-ranked the stored solutions using a more accurate scoring function in order to produce a ranked list of solutions.

Table 1 Performance profile of MolDock. Running times were obtained using one core of the CPU as described in Sect. 5. Computation time for energy grids is not shown

Step	Time (s)	%
Ligand preprocessing	<0.01	0.00
Pose computation	4.44	16.53
Scoring function	21.82	81.24
DE	0.60	2.23
Total	26.86 s	

Runs = 10; population size = 64; DE iterations = 3,000; ligand: heavy atoms = 26; rotatable bonds = 8

3.4 Parallelisation of Scoring Function and Differential Evolution

Table 1 shows the time consumption of the different steps in the MolDock algorithm. The time used on re-ranking poses is not shown as this step has been omitted in this work. However, re-ranking is fast and only has to be performed once for every run. The time used on preprocessing the protein is also not shown as this only has to be done once for each protein and when multiple ligands are docked into the same protein, this time becomes insignificant. The protein used in Table 1 contains 2,163 heavy atoms, and here cavity prediction takes 0.93 s while computation of energy grids takes 24.06 s. Parallelising the computation of energy grids, which is the most time-consuming step of the two, is easily done since each entry in a grid can be computed independently.

The most time-consuming step of docking a ligand into a preprocessed protein is pose evaluation (computation of pose atom positions and scoring of poses). Pose evaluation can be parallelised in a straightforward way as each individual in the population is evaluated independently (population level parallelisation). The mutation, recombination and selection steps in DE can also be parallelised on population level, i.e. the parameter vector of each individual in the population can be computed independently. However, to take advantage of the hundreds of cores in modern GPUs it is necessary to increase the parallelisation granularity. By using a large number of threads it is possible to hide the high memory latency by switching to other threads while waiting for memory access. CUDA and GPUs in general provide fast synchronisation between threads which makes fine-grained parallelisation schemes, such as the one presented in Sect. 4, possible. On CPUs, high granularity parallelisation using multi-threading often gives rise to a high overhead due to thread synchronisation which results in lower performance. SIMD (single instruction multiple data) instructions could be used to implement a fine-grained parallelisation scheme for CPUs, but this has not been done in the current work. A much simpler approach for parallelisation of MolDock on CPUs is to create multiple processes each running an instance of the MolDock application and then let each instance execute one or more energy minimisation runs. If multiple ligands are docked, the total number of runs needed will be large enough to fully utilise all cores in most modern CPUs. However, this approach is not suitable for GPUs.

To reduce the overhead of thread synchronisation and increase the workload for each step of the MolDock algorithm, multiple runs of energy minimisation can be performed in parallel. Performing r runs of energy minimisation in MolDock with a population of p individuals correspond to a single energy minimisation run with a population of $p \cdot r$ where mutation and crossover are restricted to groups of p individuals. As shown in Sect. 5.3 the use of this strategy is important to achieve good performance on GPUs.

4 Parallelisation of MolDock with CUDA

This section describes the CUDA kernels used to perform energy grid computation, pose evaluation and guided DE on a CUDA-enabled GPU. The main objective of the kernel designs is to maximise parallelisation granularity, minimise thread serialisation and ensure coalesced access to data stored in the global memory. The dimensions of the thread block grids used in the kernels have no impact on performance and were chosen to reduce complexity of index computations. Thread blocks contain 64 threads (unless otherwise stated) which gave the best overall performance. In our experiments both the pose evaluation and the DE are executed on the GPU. However, as shown in Fig. 3, it is also possible to execute only the pose evaluation on the GPU and let the DE runs on the CPU.

In [18] 10 runs were performed with a population size of 50. This allows a maximum of 500 independent units of work using the parallelisation of MolDock on population level described in Sect. 3.4. On modern GPUs, 500 threads are not enough to fully utilise all cores; hence, the parallelisation granularity needs to be increased. This can only be done to some degree when computing pose atom positions (see Sect. 4.3), but in case of both the scoring function and DE algorithm, a high granularity can be achieved. Fine-grained parallelism gives rise to dependencies between threads in different thread blocks. Consequently, some steps are implemented in multiple kernels to allow global thread synchronisation.

4.1 Preprocessing and Data Structures

In the preprocessing phase we compute energy grids, perform cavity detection and create a representation of the ligand which is suitable for CUDA-enabled GPUs. Energy grids are computed on the GPU while all other steps are performed on the CPU.

Computation of energy grids is done using a single CUDA kernel, where each thread block contains 128 threads and computes all grid points for a fixed z-coordinate. After all grids have been computed they are stored in 3D textures to enable the use of hardware accelerated trilinear interpolation. Cavity points are

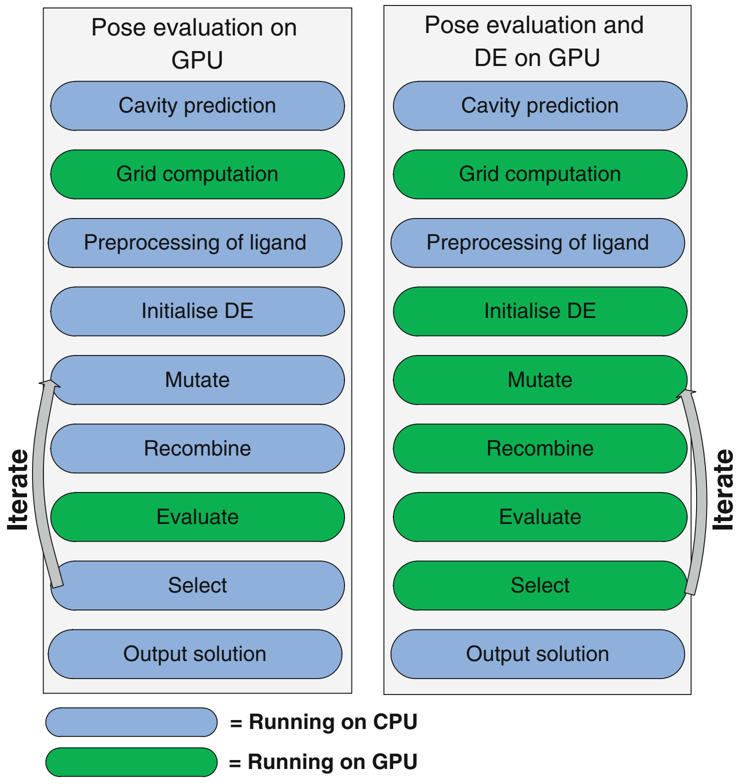


Fig. 3 The program flow of the two GPU-accelerated versions of MolDock

represented using a 3D grid and also stored in a 3D texture to allow cached access using the CUDA point filter access mode.

Preprocessing of ligands consists of several steps. First, internal rigid structures (RS) are identified along with rotatable bonds which are the single covalent bonds that connect rigid structures. Figure 4 shows a decomposition of an example ligand molecule into rigid structures. The atoms in the ligand are represented as an array containing 3D coordinates, the atom types and the charge of each atom. Data for atoms in the same rigid structure are stored sequentially such that data for all atoms in a rigid structure can be located using two indices stored in a list. Rotatable bonds are represented as a list of index pairs which identify the two heavy atoms in each bond. For each rotatable bond we also store information on all possible dihedral angles to enable computation of torsional energies. All these data structures are relatively small and can therefore be stored in the GPUs constant memory to allow fast cached access.

When computing the internal energy of the ligand, only atoms more than two bonds apart are considered. To avoid computing the bond distance between all atom pairs more than once, a list of atom pairs that need to be evaluated is created. As this

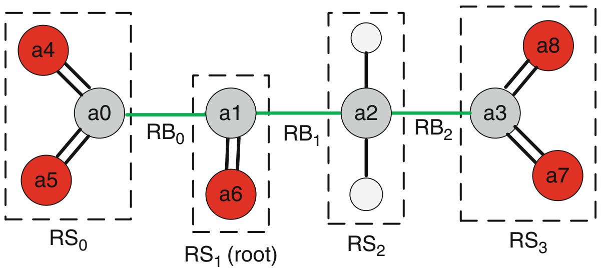


Fig. 4 The decomposition of an example ligand molecule into rigid structures

list can become fairly large, it is stored in global memory to prevent pollution of the constant memory cache.

4.2 *Guided Differential Evolution*

We use the parallelised Mersenne Twister [11, 14] from the CUDA SDK to generate uniformly distributed random numbers. To reduce the number of calls to the Mersenne Twister kernel, each call generates enough random numbers for 100 iterations of the DE algorithm (where 100 was chosen because it gave the best performance in our experiments). The random numbers are stored in global memory and loaded by the DE kernel when needed.

Three kernels are used to implement the three steps of DE (mutation, crossover and selection) in CUDA. All kernels use one thread for each parameter in each individual where threads in the same block handle the same parameter type in different individuals. There are some challenges associated with handling random numbers when DE is parallelised in this way. In most cases, threads require unique random numbers which are easily handled. But in the mutation and crossover steps, threads working on the same individual must share some random numbers.

In the mutation kernel, each thread must select three random individuals which are mutually distinct and also distinct from the individual handled by the thread. Additionally, for threads working on parameters in the same individual these three individuals must be the same. This is achieved by loading the same three random numbers in threads working on the same individual and then mapping the numbers to parameter vector indices satisfying the above constraints. Next, all parameters in the population of the type handled by a thread are loaded into shared memory to allow fast random access by threads in a block. Mutated parameters are then computed and stored in global memory. The mutation kernel is also responsible for updating the minimum energy of each individual as this cannot be done in the selection kernel (see below) because of concurrent access to pose energies by threads in different thread blocks.

The recombination kernel creates a trial parameter vector for all individuals. Each parameter in a trial vector is selected from either the mutant or target parameter vector of the corresponding individual. The source vector is selected randomly using a random number and a crossover constant as follows. Each thread loads a unique random number from the global memory and uses it to select a random source vector. As specified by the original DE method, at least one random parameter must be chosen from the mutated vector for each individual. Using the same strategy as in the mutation kernel, each thread reads another random number that is identical for threads associated with the same individual. This number is then used to choose a parameter from the mutated vector. The final trial parameter vectors are stored in global memory in a single 1D array.

In the selection kernel, threads use interaction energies, computed by the scoring function, to choose between trial vectors and the target vectors. Each thread reads the interaction energies of a trial and target vector, and if the energy of the trial vector is lower than the target vector, the trial parameter handled by the thread is used to overwrite the corresponding target parameter in global memory.

Finally, we need to determine if a pose has at least one atom inside a cavity area. This is done in a different way than in the original MolDock method to enable a better parallelisation. Instead of translating poses into cavities, poses are simply discarded if they lie outside all cavity areas. However, this step requires access to the positions of all atoms in a pose and is therefore better handled by the scoring function kernel (see Sect. 4.4).

4.3 Pose Computation

In order to evaluate the ligand energies, explicit atom coordinates must be computed. These coordinates are computed using two kernels. The first kernel computes a transformation matrix for each rigid structure in all poses using the parameter vectors from the DE kernels. Threads in the same thread block compute transformation matrices for the same rigid structure in different poses as shown in Fig. 5a. The RS closest to the centre of mass in the ligand is denoted RS_{root}. Transformation matrices for $\text{RS} \neq \text{RS}_{\text{root}}$ rotate rigid structures around a rotatable bound, while transformation matrices for RS_{root} translate and rotate the whole molecule. Each transformation matrix is computed using a single thread which stores the result temporarily in shared memory. When all threads in a block have computed their assigned transformation matrix, the matrices are written to the global memory.

The second kernel computes the position of atoms in each pose using the transformation matrices stored in global memory as shown in Fig. 5b. For each rigid structure, i, in pose, j, a transformation matrix which transforms the coordinates of all atoms in $\text{RS}_{i,j}$ is computed using multiplication of the transformation matrices computed by the first kernel and four threads for each matrix. Next, three of the four

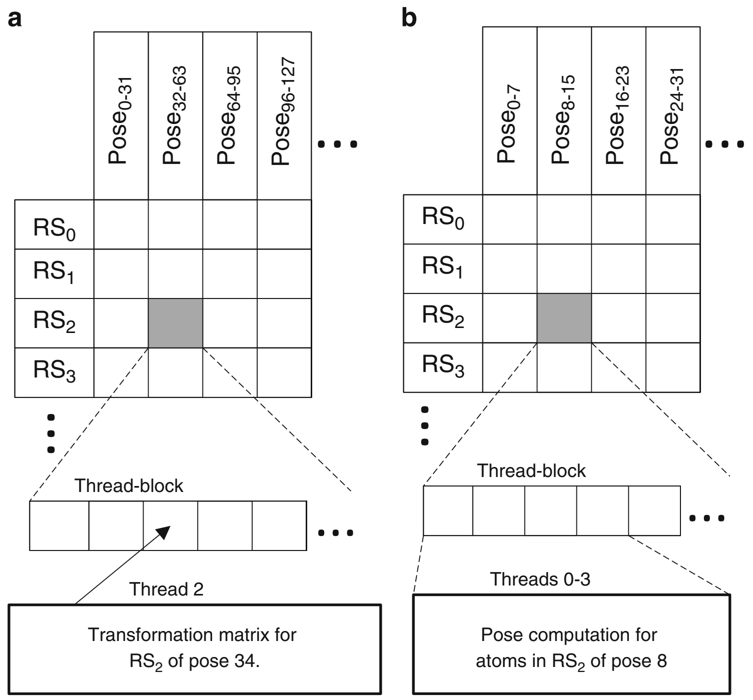


Fig. 5 The two kernels used for computing pose atom positions. (**a**) Grid layout for the transformation matrix computation, (**b**) grid layout for the atom position computation

threads are used to compute atom coordinates for atoms in RS$_{i,j}$ which are then stored in global memory.

4.4 Scoring Function

The scoring function is computed using two kernels. In both kernels atom coordinates are loaded from the global memory into shared memory as needed using coalesced reads and accessed there.

The first kernel, illustrated in Fig. 6, performs two different functions. The first function is computation of inter and intramolecular energies for each atom excluding torsional energies. Here, one thread is used for each atom in a pose where threads in the same thread block work on the same ligand atom but from different poses. The intermolecular energy is computed using trilinear interpolation in energy grids. Intramolecular energy is computed as the PLP of atom pairs which are more than two bonds away from the atom handled by a thread. All threads in a block run through the same list of atom pairs as these threads handle the same ligand atom.

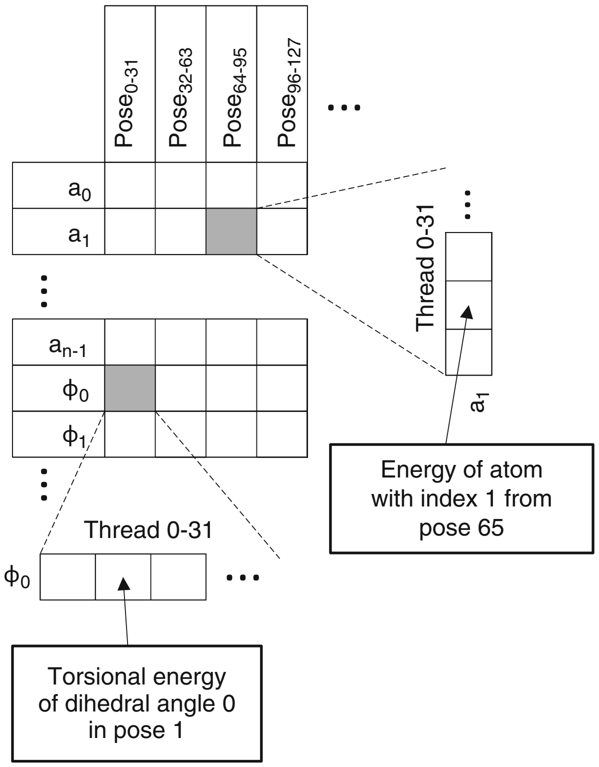


Fig. 6 Grid layout for computing both atom-specific energies and torsional energies

The total energy contribution of each atom is stored in global memory as a final step in this kernel.

The second function performed by the kernel is computation of the torsional energy term. This is done in parallel with computation of atom-specific energies using additional thread blocks where the grid coordinates of a thread block are used to determine the function of each block (see Fig. 6). In a thread block computing torsional energies, all threads compute the energy of one possible dihedral angel from the same bond but for different poses. The torsional energy contribution is also stored in global memory along with the atom-specific energies.

The second kernel is used to compute the total interaction energy of poses and to handle cavities. A single thread is used to sum the atom-specific and torsional energies for each pose. Each thread also determines if at least one atom in the pose is inside a cavity by performing a lookup in the cavity grid for each atom. If no atoms are inside a cavity, a penalty of 10,000 is given to the pose interaction energy, thereby discarding the pose. In this way the hard constraint used in MolDock where ligands are always positioned in cavities is replaced by a soft constraint which can be efficiently computed on a GPU. Finally, the total interaction energy of each pose is stored in the global memory.

Table 2 Dataset statistics (133 complexes)

	Min	Max	Avg.
#Heavy atoms	6	55	21.32
#Rotatable bonds	0	23	5.89

5 Results and Discussion

Our parallelisation schemes handle functions which are common for several popular flexible docking methods similar to MolDock such as PLANTS [8], GEMDOCK [19] and AutoDock [3] and are thus generally applicable. In particular, the parallelisation scheme for computation of pose atom coordinates can be directly used in most flexible docking methods. The parallelisation scheme for the MolDock scoring function is also fairly easy to apply on other methods (fx PLANTS, GEMDOCK and AutoDock) and can also be used in combination with other optimisation methods than DE.

An optimised implementation of MolDock was made for x86-64 compatible CPUs in $C++$ and parallelised on population level with OpenMP [1] as described in Sect. 3.4. The kernels described in Sect. 4 were implemented using the CUDA SDK 3.2 and used to accelerate pose evaluation and DE steps from the CPU implementation (see Fig. 3). All code was compiled with the -O3 optimisation option using GCC 4.4.3 and nvcc 3.2. Both the CPU and GPU implementations use single precision floating point numbers and compute multiple runs of the energy minimisation step by creating one large population as described in Sect. 3.4. Unless stated otherwise, experiments using the GPU perform both pose evaluation and DE on the GPU. The crossover rate (CR) and scaling factor (F) for the DE algorithm were set to 0.9 and 0.5, respectively, which are the same values used for experiments in [18]. These parameters affect the rate of convergence and the docking accuracy, but they do not have any direct influence on time consumption.

The experiments were performed on a machine containing an Intel Core 2 Quad Core (Q9450) CPU @ 2.66 GHz with 4 GB RAM running Ubuntu 10.04 and equipped with a Geforce 8800GT GPU (112 cores @ 1.5 GHz). The CPU and GPU are both mainstream products in the same price range and therefore provide a good basis for comparison.

5.1 Dataset

We used the original GOLD benchmark dataset [13] to perform all experiments. This dataset contains the same 77 co-crystallised protein-ligand complexes used for experiments in [18] plus an additional 56 complexes. When the energy grids have been computed, the running time of MolDock depends mainly on the number of heavy atoms and rotatable bonds in the ligand. The GOLD dataset contains a diverse range of ligand sizes as shown in Table 2 and is therefore well suited

Table 3 The average number (and standard deviation) of successfully docked complexes out of 77 complexes using best score ("Score" column) and best RMSD ("RMSD" column) of all runs

	CPU		GPU	
#Runs	Score	RMSD	Score	RMSD
1	56.0 ±1.5	56.0 ±1.5	55.9 ±2.0	55.9 ±2.0
5	59.9 ±2.1	68.0 ±1.8	60.0 ±1.7	67.7 ±1.4
10	59.6 ±1.1	69.6 ±1.2	59.6 ±1.2	70.1 ±0.9
15	58.4 ±1.4	71.9 ±0.8	59.5 ±1.7	72.3 ±0.9
20	58.8 ±1.2	71.7 ±0.9	58.0 ±0.8	72.4 ±1.2

The numbers are the average of ten individual experiments

for benchmarking the parallelised MolDock implementation. In all experiments individuals in DE populations were initialised by randomly translating and orientating the co-crystallised ligand and assigning random dihedral angles to rotatable bonds.

5.2 *Docking Accuracy*

Several differences between the implementation used here and the one used in [18] could affect the accuracy. Both the use of energy grids and the omission of a scaling factor for hydrogen bonds change the energy landscape. Also, the use of trilinear interpolation and arithmetic functions with low precision on the GPU might affect the accuracy. Omission of the pose re-ranking step does, however, not affect the energy minimisation step as it is used in a post-processing step to select the best pose of all energy minimisation runs.

To determine any loss of accuracy we docked the same 77 complexes used in [18] on both the CPU and GPU. Table 3 shows the number of successfully docked complexes where successful means that a solution was found with a RMSD ≤ 2 Å relative to the co-crystallised ligand. The results were obtained using a population size of 64 and 3,000 iterations of the DE algorithm. To enable a comparison with the results in [18], Table 3 shows both the number of successful docks using the best scoring pose of all runs (the "Score" column) and using the pose with the lowest RMSD of all runs (the "RMSD" column). With re-ranking of poses using a more precise scoring function, the number of successfully docked poses can be expected to lie between these two numbers. The original MolDock method successfully docked 67 of the 77 complexes and the results in Table 3 show that both the CPU and GPU implementations have comparable high accuracy with ten runs if re-ranking is used.

In [6] a decrease in accuracy was observed when using hardware accelerated trilinear interpolation on the GPU to compute intermolecular energies. We did not

Table 4 Time consumption of the steps for docking a single ligand in MolDock on CPU and GPU

	CPU		GPU	
Step	1 Core	4 Cores	PE	PE + DE
Ligand pre-processing (s)	<0.01	<0.01	<0.01	<0.01
Pose computation (s)	4.44	1.16	0.34	0.36
Scoring (s)	21.82	6.36	0.61	0.61
DE (s)	0.60	0.61	0.59	0.15
GPU data copy	N/A	N/A	0.18 s	~0 s
Total time (s)	26.86	8.13	1.72	1.12
Speedup	1.0×	3.3×	15.6×	24.0×

PE pose evaluation
Runs = 10; population size = 64; DE iterations = 3,000;
ligand: heavy atoms = 26; rotatable bonds = 8

observe any difference in accuracy when using this feature compared to performing interpolation in software on the GPU. The performance is, however, increased by up to 68 % when hardware accelerated interpolation is used.

5.3 Benchmark Results

Table 4 shows the time consumption of the different steps in MolDock when docking a single complex of average size. In case of the CPU, running times using both a single core and four cores for the pose evaluation step are shown. Parallelisation of MolDock with threads on the CPU does not fully utilise all cores because of thread synchronisation overhead. The DE algorithm was not parallelised in the CPU implementation as experiments showed that the overhead of thread synchronisation caused the running time of DE to increase by up to 4×. However, as shown in Fig. 7, a speedup of close to 4× with four cores can be achieved by running multiple instances of the program (one for each core) in parallel and is therefore the preferable way to parallelise MolDock on a CPU.

In case of the GPU, Table 4 shows the running times for performing only pose evaluation on the GPU and when both pose evaluation and DE are running on the GPU. With DE running on the CPU, data must be copied between the CPU and GPU after each iteration of the DE algorithm. Here, the time used on copying data and running the DE algorithm on the CPU is responsible for 45 % of the total time consumption. By running the DE algorithm on the GPU, the time consumption of the DE algorithm is reduced and most data transfers are avoided which increases performance significantly. The preprocessing time of the protein is also improved by parallelising the computation of energy grids. Compared to 20.04 s using one core on the CPU, the energy grids can be computed in 12.23 s when 4 cores are used and in only 1.42 s on the GPU. Consequently, the total time (including the time used

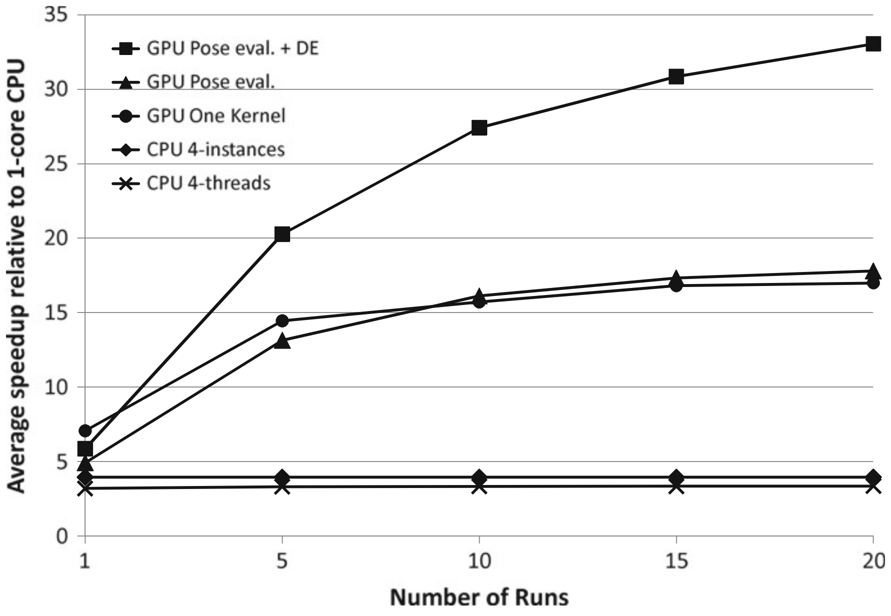


Fig. 7 Average speedups relative to the CPU implementation using one core when docking 133 complexes with different number of runs

on initialising all data structures on the CPU and GPU) needed to dock the complex used in Table 4 is reduced from 52.2 s using a single core on the CPU to 3.96 s on the GPU which is a 13.2× speedup.

Figure 7 shows the average speedup achieved on all 133 complexes in the GOLD dataset by both CPU and GPU as a function of the number of runs performed on each complex. The running times were measured as the time used on energy minimisation plus the time used on preprocessing the ligand, i.e. the total time needed to dock a ligand into a preprocessed protein. When performing ten runs, multi-threading on the CPU resulted in an average speedup of 3.3×, whereas an average speedup of 3.9× was achieved by running multiple instances of the program on the CPU each docking a different ligand.

With only four cores available in the CPU, it is easy to achieve a good utilisation of all cores. However, on the GPU a large number of thread blocks are needed to fully utilise all 112 cores which can be achieved by increasing the number of runs. When DE is running on the CPU it is a bottleneck and for more than ten runs, the performance does not increase significantly. However, by running DE on the GPU, the bottleneck is removed and we achieve an average speedup of 27.4× and 33.1× with 10 and 20 runs, respectively. Minimum and maximum speedup for docking a single ligand with ten runs and both pose evaluation and DE running on the GPU are 7.7× and 37.5×, respectively. An alternative parallelisation scheme proposed in [6] was also evaluated. Here, the evaluation of poses is done in a single kernel where each thread block is assigned to a pose and each thread is assigned to an atom. The threads compute atom positions and interaction energies for the assigned atom and share the computation of torsional energies. The advantage of this approach is that all intermediate results can be cached in shared memory which minimises access to the global memory. The main disadvantage is that computations on rotational bonds cannot utilise all threads in a block. Figure 7 shows the speedup achieved by this parallelisation scheme (GPU one kernel) with DE running on the GPU. For a single run it performs well but as the number of runs increases, the GPU runs out of resources due to a large number of idle threads. In our parallelisation scheme, all threads have no or few idle clock cycles which improves the performance by 74 % on average with ten runs.

The average docking time with ten optimisation runs for the 133 ligands is 0.77 s on the GPU. This is fast enough to perform large virtual screening experiments in a reasonable amount of time. However, screening, e.g., a million ligands would still take more than a week using a single GPU. We therefore plan to investigate different methods for increasing the performance even further. As seen in Fig. 7 the GPU used in these experiments is not fully utilised with ten runs, and the increase in accuracy is minimal when using more than ten runs on both the CPU and GPU. Therefore, we need another way of increasing the utilisation of GPUs than by increasing the number of runs. It is questionable if the parallelisation granularity for MolDock can be increased and unlikely that this approach will scale well for GPUs with more than 100 cores. Instead, multiple ligands could be docked simultaneously, either into the same or different proteins, which would increase the number of thread blocks that are executed concurrently. Factors such as an increased load on the GPU cache and complex indexing could become a bottleneck in this approach, but it seems to be the most promising direction for further research.

6 Conclusion

In this chapter, we have presented methods for parallelising flexible molecular docking with MolDock on both CPUs and CUDA-enabled GPUs which are applicable to other flexible molecular docking methods. When docking multiple ligands into a target protein on a four-core CPU, we achieved an average speedup of 3.3× with multi-threading and 3.9× by running multiple instances of the program concurrently. Using a comparable GPU with 112 cores, the average speedup was increased to 27.4× which translates to an average docking time of less than a second per ligand. The modifications made to the MolDock method did not decrease the accuracy on either the CPU nor the GPU. The reduction in running time per ligand achieved with a cheap consumer GPU reduces the cost and the amount of hardware needed to perform virtual screening with MolDock significantly.

References

1. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (2002)
2. Dorigo, M., Birattari, M., Stutzle, T.: Ant colony optimization. IEEE Comput. Intell. Mag. **1**(4), 28–39 (2006)
3. Goodsell, D.S., Morris, G.M., Olson, A.J.: Automated docking of flexible ligands: applications of AutoDock. J. Mol. Recognit. **9**(1), 1–5 (1996)
4. Goodsell, D.S., Olson, A.J.: Automated docking of substrates to proteins by simulated annealing. Proteins **8**(3), 195–202 (1990)
5. Halperin, I., Ma, B., Wolfson, H., Nussinov, R.: Principles of docking: an overview of search algorithms and a guide to scoring functions. Proteins **47**(4), 409–443 (2002)
6. Kannan, S., Ganji, R.: Porting AutoDock to CUDA. In: Computational Intelligence on Consumer Games and Graphics Hardware (2010)
7. Kirkpatrick, S.: Optimization by simulated annealing: quantitative studies. J. Stat. Phys. **220**(4598), 671–80 (1984)
8. Korb, O.: PLANTS: application of ant colony optimization to structure-based drug design. Chem. Cent. J. **3**(Suppl 1), O10 (2006)
9. Korb, O.: Efficient ant colony optimization algorithms for structure- and ligand-based drug design. Dissertation, Universität Konstanz (2008)
10. Kozakov, D., Brenke, R., Comeau, S., Vajda, S.: PIPER: an FFT-based protein docking program with pairwise potentials. Proteins **65**(2), 392–406 (2006)
11. Matsumoto, M., Nishimura, T.: Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simulat. **8**(1), 3–30 (1998)
12. Muegge, I., Rarey, M.: Small molecule docking and scoring. In: Lipkowitz, K.B., Boyd, D.B. (eds.) Reviews in Computational Chemistry, chap. 1, pp. 1–60. Wiley, New York (2001)
13. Nissink, J.W.M., Murray, C., Hartshorn, M., Verdonk, M.L., Cole, J.C., Taylor, R.: A new test set for validating predictions of protein-ligand interaction. Proteins **49**(4), 457–471 (2002)
14. Podlozhnyuk, V.: Parallel Mersenne Twister. Technical report, NVidia (2007)
15. Stone, J.E., Hardy, D.J., Ufimtsev, I.S., Schulten, K.: GPU-accelerated molecular modeling coming of age. J. Mol. Graph. Model. **29**(2), 116–125 (2010)
16. Storn, R., Price, K.: Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces. J. Global Optim. **11**(4), 341–359 (1997)
17. Sukhwani, B., Herbordt, M.C.: GPU acceleration of a production molecular docking code. In: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2), pp. 19–27. ACM, New York (2009)
18. Thomsen, R., Christensen, M.H.: MolDock: a new technique for high-accuracy molecular docking. J. Med. Chem. **49**(11), 3315–3321 (2006)
19. Yang, J.M., Chen, C.C.: GEMDOCK: a generic evolutionary method for molecular docking. Proteins **55**(2), 288–304 (2004)

Using Large-Scale Parallel Systems for Complex Crystallographic Problems in Materials Science

Laurent A. Baumes, Frédéric Krüger, and Pierre Collet

Abstract Very recently, the design and understanding of materials synthesis have received a huge effort in which modeling approaches are decisive. Here, we focus on the generation of crystalline inorganic frameworks. Despite the high-throughput (HT) methods that have proved to be useful for the discovery of zeolites, the determination of the new phase's structure takes up a large part of the entire process. Therefore, we show how graphic processing units or GPUs can be used in order to speed up this mandatory step. We describe GPUs and predictive methods for phase determination. Then, we show all the details that allowed us to reach a stable and robust solution with benchmark analysis and real applications to zeolites.

1 Introduction

Modelling approaches for the design and understanding of new materials have received an important effort during the last decade [8, 9, 16, 22, 38–40] due to the increasing power of computers. Among the different solutions that allow CPU-greedy problems to be tackled, parallelism and related programming environments are decisive. Based on the particularities/characteristics of the problem to be solved, one or another solution may be better suited. The MPI and OpenMP approaches are now mature solutions that can be considered as standards in the contexts of shared and non-shared memory systems.

L.A. Baumes (✉)
Insituto de Tecnologia Quimica, UPV-CSIC, Valencia Spain

Present Organization: ExxonMobil Research and Engineering Co. RDSS - Research Computing
1545 Route 22 East, Room CC286 Annandale, NJ 08801
e-mail: laurent.a.baumes@exxonmobil.com

F. Krüger · P. Collet
University of Strasbourg, ICUBE, Illkirch, France
e-mail: Frederic.Kruger@etu.unistra.fr; Pierre.Collet@unistra.fr

These solutions are widespread in the scientific community, and what makes them even more attractive is the sustenance and backing of a group of large companies, such as Intel, that promise their long-term support. On the other hand, another branch of the industry of computers and related components supports a new approach based on the use of graphics cards offering one of the most attractive cost/performance ratios. However, programming of the so-called general purpose graphics processing units (GPGPUs) remains difficult due to the very particular architecture of memory and processors. This chapter describes our experience of the use of GPGPU-based evolutionary calculations with applications in materials science [24, 27, 28].

The system is tested for the very fast generation of four-connected 3D nets as potential zeolitic structures. Results are very promising and examples of hypothetical materials are successfully provided. Finally, new features are integrated into EASEA that allow a full use of parallelism at all levels in an island model which is employed for zeolitic framework determination.

2 Fast Generation of Four-Connected 3D Nets as Potential Zeolitic Structures

Researchers have already shown their interest in using GPGPUs for chemical problems such as molecular modeling, DFT and quantum chemistry calculations [3, 32, 45, 50, 51, 58]. However, we were pioneers for their use on the problem of zeolite structure determination, and for this application, we extended the EASEA platform to memetic algorithms (*cf.* the chapter "Generic Local Search (Memetic) Algorithm on a Single GPGPU Chip").

Zeolites are crystalline materials with regular structures consisting of molecular-sized pores and channels. These crystals are widely spread and used in a lot of important industrial applications such as in the fields of absorption, ion exchange and heterogeneous catalysis, as well as in health, sensors and solar energy conversion. They are made of tetrahedrons of oxygen atoms that contain a single silicon or aluminium atom in their center.

Rietveld refinement techniques can be used to extract structural details from an X-Ray powder Diffraction (XRD) pattern, provided an approximate structure is known. However, if a structural model is not available, its determination from powder diffraction data is a non-trivial problem. The structural information contained in the diffracted intensities is obscured by systematic or accidental overlap of reflections in the powder pattern. As a consequence, the application of structure determination techniques which are very successful for single crystal data (primarily direct methods) is, in general, limited to simple structures or materials that can be obtained as single crystals. Here, we focus on inherently complex structures of a special type of crystalline materials whose periodic structure is a four-connected three-dimensional net such as aluminosilicates, e.g. zeolite.

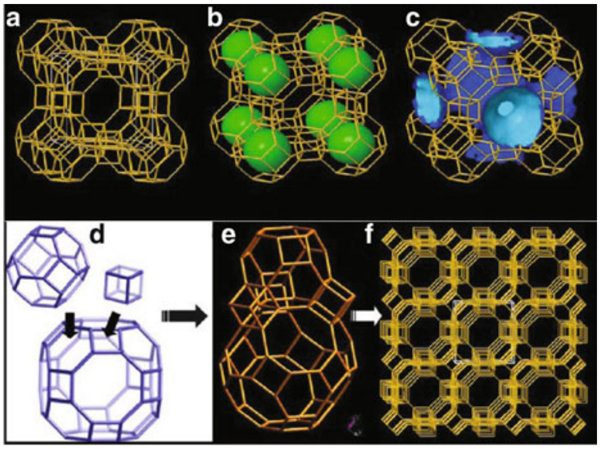


Fig. 1 LTA crystal framework: (**a**) the *white cube* is the unit cell, (**b**) cages are represented in *green*, (**c**) 3D channels are in *blue*, (**d**) split of the structure LTA into building units, (**e**) piece of LTA structure and (**f**) crystal structure with 27 unit cells

Silico-alumino-phosphates (SAPO), alumino-phosphates (AlPO), etc., could be tackled using the same methodology. These materials are all microporous materials, whose structures allow molecules to be sorted based on a size exclusion process due to the presence of channels (Fig. 1c) and cages (Fig. 1b), as shown in Fig. 1 for the zeolite framework called LTA. The picture only shows a finite part of the structure as crystals are 3D periodic, i.e. the unit cell in white (Fig. 1a) is repeated by simple translation in all the dimensions, (Fig. 1f) containing 27, i.e. $3 \times 3 \times 3$, unit cells.

New zeolite synthesis remains a challenge even if high-throughput (HT) methods have proved to accelerate the rate of discovery for zeolites [11], as well as other solid functional materials [10, 19, 23, 44]. Still, the identification of multi-phase samples through powder diffraction [4, 5] and the determination of the new phase's structure take up a large part of the entire process. Considering the latter, we propose the use of general purpose graphic processing unit (GPGPU) cards in order to speed up the determination of new microporous crystalline structures, which are fundamental in condensed matter [57].

The problem posed by prediction is to obtain approximate models which can then subsequently be easily refined by routine modelling techniques. Various techniques have been employed such as Monte Carlo basin hopping (MCBH) [52, 53], simulated annealing (SA) [21, 33, 36, 37], and genetic algorithms (GA) [1, 20, 26, 29, 34, 35, 49, 55]. Topological principles can be considered as another solution since they allow the enumeration of networks [54], and among them four-connected nets such as zeolites [14, 17, 18, 25, 30, 31, 41–43, 47, 48]. Despite the fact that the use of computational techniques is becoming standard for structural studies of solid materials [2, 6, 7, 12, 13, 46, 56], the main problem remains to obtain correct solutions in reasonable time. As opposed to most predictive techniques, rather than

Table 1 Constants definition

ANGLE_MIN	75.0000
ANGLE_MAX	160.0000
ANGLE_AVG_OPT	109.5400
ANGLE_AVG_OPT_MIN	106.0000
ANGLE_AVG_OPT_MAX	112.0000
DIST_MIN	2.7000
DIST_MAX	3.5000
DIST_OPT	3.0700
DIST_OPT_MIN	2.9000
DIST_OPT_MAX	3.2500
DIST_MIN_SQ	7.2900
DIST_MAX_SQ	12.2500
DIST_OPT_SQ	9.4249
DIST_OPT_MIN_SQ	8.4100
DIST_OPT_MAX_SQ	10.5625

evaluating an energy function, a simple fitness function [15] provides a framework assessment good and flexible enough not to discard correct solutions while retaining only a reasonable number of candidate solutions that can easily be refined by routine procedures. It should be noted that for the greatest part of the search space, the distribution of atoms does not refer to any realistic solution, for which a simple and fast assessment remains sufficient.

When looking for new structures, strategies usually employ XRD data in order to compare potential solutions by calculating theoretical diffraction data. Here, no diffraction calculation is integrated in the evaluation of candidates as our aim is double. The algorithms purpose is to provide a practical quantity of different potential solutions that are stored in a DB and refined on a second computer by a routine based on an interatomic potential technique using GULP code, and among them, we expect the structure to be discovered. Because recent zeolite synthesis in discovery programs involves T atoms that differ from the original formulation of aluminosilicates such as Ge, P or B, e.g. considering ITQ and SSZ materials, there will not be any implicit chemical composition. Only unspecified T atoms are considered, and oxygen is omitted. Our function is reduced to geometric descriptors based on T-T-T angles, T-T distances and a connectivity term. The best possible fitness is zero when all T are tetra-coordinated, and distances and average angles exactly correspond to optimized values, i.e. 3.07 A and 109.51 (see Table 1), while deviations from this optimum sum penalties (see pseudo code in Table 2). Required inputs (see Table 3) are the unit cell lattice parameters, dimensions and angles and density range.

Concerning the choice of the strategy, it should be noted that nothing leads us to think that only simple functions can be optimized. Recent GPU cards such as the Tesla 2070 have up to 6 GB of memory. The only real restriction on the programming of these cards is that they cannot execute recursive code, and

Table 2 Simple fitness function

```
//Local variables
float distSq;                   // Distance square for two given atoms
float dist;                     // Distance for two given atoms
float aux;                      // auxiliar value to get the fitness
float errors;                   // contains distance and nbLink errors meassure
float errors3MR;                // contains the 3MR errors meassure
float errorsAngles;             // contains a angle errors meassure
int mult;                       // Current atAU multiplicity
Atom  linkedAtoms[];            // Array containing the atoms linked to atAU
float linkedDists[];            // Array containing the link distances to atAU
int nbLinks;                    // Current number of link distances
float angle;                    // Angle for three given atoms
float avgAngles;                // Angles average for the current atAU atom
int nbAngles;                   // Current number of angles formed with atAU
// Local variables initialization
int   NBLinkErrors =  4 * auAtoms.Count;
float Fitness = (4 * ucAtoms.Count * DIST_OPT) + (6 * ucAtoms.Count * ANGLE_AVG_OPT);
aux = 0.0f;    errors = 0.0f;
errors3MR = 0.0f;           errorsAngles  = 0.0f;
foreach(Atom atAU in auAtoms){
        nbLinks=0;    mult=auMultiplicity[atAU];
        //au x uc
        foreach(Atom atUC in ucAtoms){
                if(atAU is not atUC){
                        distSq = distanceSq(uc, atAU, atUC);
                        if(distSq in the range [DIST_MIN_SQ, DIST_MAX_SQ]){
                                dist = sqrt(myDistSq);        addAtom(linkedAtoms, atUC);   addDist(linkedDists, di nbLinks++;
                                if(nbLinks <= 4){
                                        if(dist in the range [DIST_OPT_RANGE_MIN, DIST_OPT_RANG aux += (DIST_OPT * mult);
                                        else           aux += (DIST_OPT ? abs(DIST_OPT - dist)) * mult;
                                        NBLinkErrors--;
                                }
                                else{
                                        errors += DIST_OPT * mult * (1.0f + abs(DIST_OPT - dist));          NBLinkErrors++;
                                }
                        }
                        else if(myDistSq < DIST_MIN_SQ)           errors += DIST_OPT_SQ * mult * (1.0f + abs(DIST_MIN_SQ - distSq));
                }
        }
        // au x nb
        foreach(Atom atNB in nbAtoms){
                distSq = distanceSq(uc, atAU, atNB);
                if(distSq in the range [DIST_MIN_SQ,  DIST_MAX_SQ]){
                        dist = sqrt(myDistSq);
                        addAtom(linkedAtoms, atNB);
                        addDist(linkedDists, dist);
                        nbLinks++;
                        if(nbLinks <= 4){
                                if(dist in the range [DIST_OPT_MIN,  DIST_OPT_MAX])    aux += (DIST_OPT * mult);
                                else                  aux += (DIST_OPT - abs(DIST_OPT - dist)) * mult;
                                NBLinkErrors--;
                        }
                        else{  // more than 4 links
                                errors += DIST_OPT * mult * (1.0f + abs(DIST_OPT - dist));             NBLinkErrors++;
                        }
                }
                else if(distSq < DIST_MIN_SQ errors += DIST_OPT_SQ * mult * (1.0f + abs(DIST_MIN_SQ - distSq));
        }
        // Reset the number and average angles values for this atAU
        nbAngles=0; avgAngles=0.0f;
        foreach(Atom linkedAtomJ/j=0 in linkedAtoms){
                for(Atom linkedAtomK/k=j+1 in linkedAtoms){
                        // 3MR
                        distSq = distanceSq(uc, linkedAtomJ, linkedAtomK);
                        if(distSq is in the range [DIST_MIN_SQ, DIST_MAX_SQ] errors3MR += DIST_OPT_SQ * mult * 1.0f + (DIST_MAX_SQ - distSq));
                        //ANGLES
                        myDist = sqrt(myDistSq);          angle  = getAngle(atAU, linkedAtomJ, linkedAtomK);
                        if(angle is in the range [ANGLE_MIN, ANGLE_MAX]){
                                nbAngles++;
                                if(nbAngles <= 6)              avgAngles += angle;
                                else                  errorsAngles += ANGLE_AVG_OPT - abs(ANGLE_AVG_OPT - angle);  // more than 6 angles
                        }
                        else                  errorsAngles+=ANGLE_AVG_OPT-abs(ANGLE_AVG_OPT-angle);
                }
        }
        // only takes the first 6 angles
        if(nbAngles > 6)  nbAngles = 6;
        if(nbAngles > 0)  avgAngles = avgAngles / nbAngles;
        if(avgAnges in the range [ANGLE_AVG_OPT_MIN, ANGLE_AVG_OPT_MAX])          aux += ANGLE_AVG_OPT * nbAngles * mult;
        else              aux += (ANGLE_AVG_OPT - abs(ANGLE_AVG_OPT - avgAngles))*nbAngles * mult;
}
Fitness -= aux;
if(Fitness >= 0)  Fitness += errors + errors3MR + errorsAngles;
else                  Fitness = abs(Fitness - errors - errors3MR ? errorsAngles);
output [Fitness];
```

Table 3 Inputs

UnitCell uc	Unit cell dimensions (a, b, c, α, β, γ)
Atom[] auAtoms	Array containing the asymmetric unit atoms coordinates. (T-Atoms)
int[] auMultiplicity	Symmetry multiplicity for each T-Atom
Atom[] ucAtoms	Array containing the unit cell atoms coordinates (Symmetry operations over T-Atoms)
Atom[] nbAtoms	Array containing the atoms in the neighbour cells. only the atoms that could create links with ucAtoms

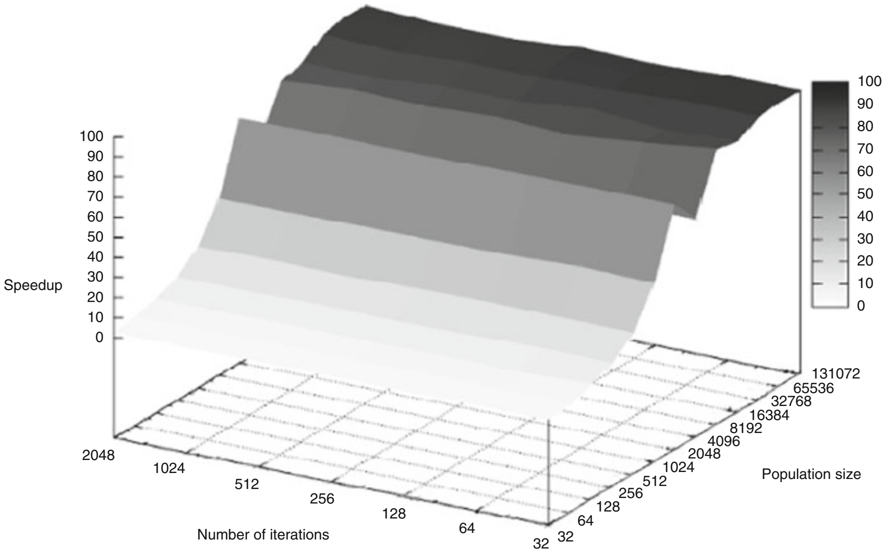


Fig. 2 Speedup for the zeolite fitness function

they cannot perform dynamic memory allocation. Both restrictions are removed since computer science theory shows that it is possible to rewrite any recursive function with non-recursive loops and that dynamic memory allocation is not really needed if one can allocate a large-enough chunk of memory when starting the program.

In order to compare the speedup obtained on benchmarks and that for the zeolite application, the same setup was tested. In Fig. 2, the speedup for a genome of 12 floats (x; y; z position for four atoms) is given as a function of the population size and the number of iterations. The fitness function takes much more time to evaluate than the Rosenbrock function, meaning that the number of iterations does not much impact speedup, and overhead is minimized; 32 iterations are sufficient to keep the GPU cores busy (where many more iterations were needed for the ultrafast Rosenbrock function). Therefore, only population size has an influence on speedup. As usual, the card needs at least 2048 individuals to get a "reasonable" speedup of 71 for only 32 iterations. Then, adding more individuals allows the card to optimize its scheduling, and a maximum speedup of 94 is obtained for 65,000 individuals and 64 iterations (to be compared with 91 for 65,000 individuals and 32 iterations or 84 obtained with only 16,000 individuals and 32 iterations). Note that the fitness function performs numerous crystallographic calculations such as matrix multiplications, e.g. space group, for the transformation of atoms from the asymmetric unit to the unit cell in order to be able to further calculate the geometrical features that characterize a stable zeolite.

Tests (not shown here) using this strategy demonstrate that each zeolite from the IZA website with 5T can be solved in minutes; 32,000 individuals were employed. Figure 3 is a six-dimension chart in which each axis contains two coordinates of the AFX framework, and the color refers to fitness intervals where the darkest zone represents near-optimal fitness. AFX is selected as an example in order to show how the fitness function allows the correct structure to be located (see *circled* area in Fig. 3) and thus guides the search toward the objective. AFX is made of two T atoms: T1 (0.2265, 0.2268, 0.0788) and T2 (0.3328, 0.4398, 0.1712). Figure 3 shows the fitness landscape taking x1 and y2 fixed at their objective values, while the other four coordinates vary with a step of 0.06A within the following ranges: y1 in the range [0.15–0.25] in fractional corresponding to [2.0511–3.4185] in orthogonal, x2 in [0.25–0.35], [3.4185–4.7859] in orthogonal, z1 and z2 in [0.05–0.20] in fractional equivalent to [0.9847–3.93] in orthogonal. On the x-axis x2 and z2 are varied, while on the y-axis y1 and z1 are varied. In order to plot a 4D into a 2D chart, the z1 coordinate varies to the next step ($step\_z1\_t + 1 = step\_z1\_t + 0.06$) after all values of y1 have been plotted for step_z1_t; the same procedure is employed for x2 and z2. For example, the second value of z1 appears after all values of y1 have been calculated with the first value of z1, respectively, for z2 and x2. It can be observed that the AFX framework that corresponds to the coordinates circled in black is correctly marked as a dark dot, while deformed structures are displayed as lighter (orange) grey.

In order to show that such a strategy allows stable structures to be found even when they contain a relatively high number of T atoms (here 6–14), we will suppose that the three different configurations of unit cells (UC) listed in Table 4 are extracted from XRD indexing, while a measure of density allows a number of 96$\pm$8 T per UC to be defined. Note that these unit cell configurations do not correspond to any existing frameworks nor to previously investigated hypothetical ones. Therefore, the resulting structures that may come out of the minimization of the fitness function taking account of the framework restrictions (density and unit cell dimensions) given earlier should be new hypothetical structures.

Three different instances of our algorithm were run sequentially on the full GTX 295 for a total of one week. Nearly 400 structures were found, of which 88 % show an energy per T after refinement in [128.2–129.8], indicating that this technique is clearly able to pre-select stable zeolitic frameworks. Figures 4–6 show

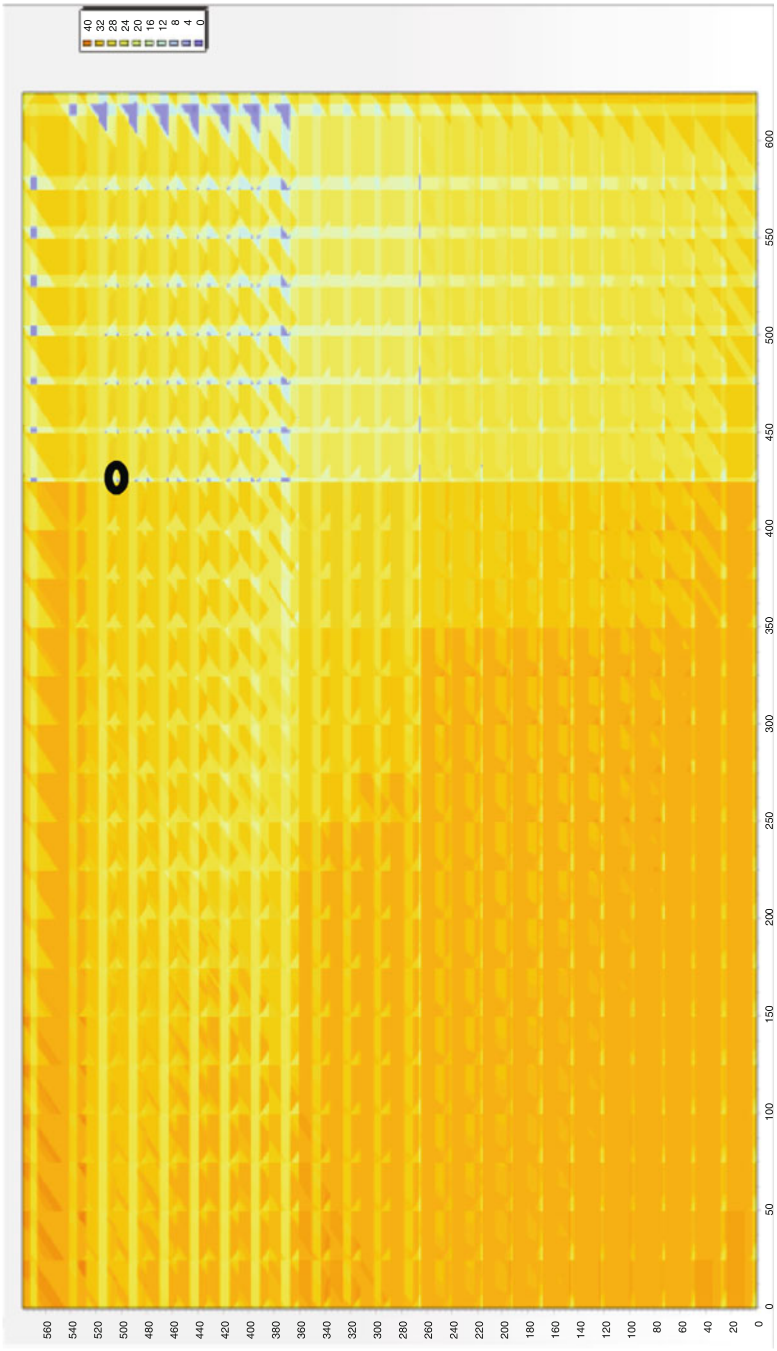


Fig. 3 AFX fitness landscape

Table 4 Configurations of unit cells. V = 5550A3, 96T/UC±8T

Unit cell	*a*	*b*	*c*	α	β	γ	Space group
A	17.868	13.858	22.41	90	90	90	Imma (74)
B	13.858	17.868	22.41	90	90	90	Imma (74)
C	17.868	22.41	13.858	90	90	90	Ima2 (46)

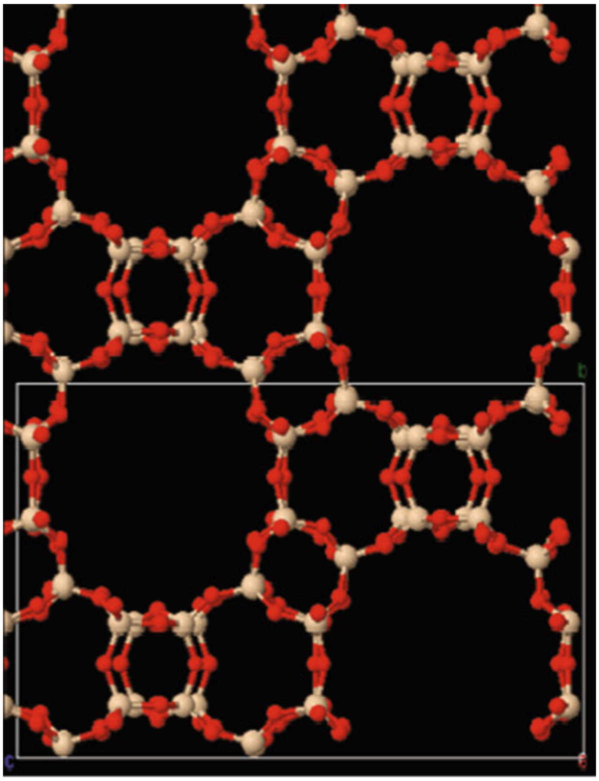


Fig. 4 View of solution #21 with 6T in unit cell A along axis bc

three different structures where balls in bright grey correspond to T atoms and dark red ones represent oxygen (oxygen atoms are placed in the middle of each pair of T atoms which are within bond distance before refinement) and whose related parameters and energies are given in Table 5. For each of the cell configurations, the best structure is shown, italicized in the corresponding table (see Tables 6–8).

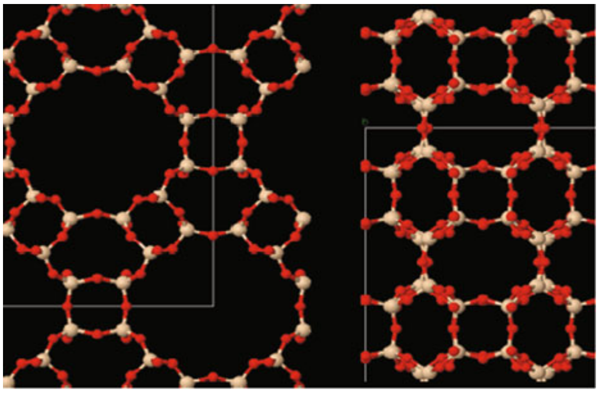


Fig. 5 View of solution #36 with 6T in unit cell B along axis ac (*left*), axis ba (*right*)

3 An EASEA Parallelization at Higher Levels

3.1 Implementation

In the previous section, we presented our implementation of evolutionary algorithms (EAs) using graphics processing units (GPUs) for solving hypothetical zeolite structures. We therefore created artificial cases defined by unit cell dimensions and density of atoms, and we tested whether our fitness function was able to find stable structures while being evaluated on the GPGPU. The methodology allowed various possible crystalline solutions to be generated by optimizing the geometry of the zeolites under the defined constraints after one week of calculation. In order to be able to get faster results and, thus, to increase the complexity of our fitness function, such as integrating the comparison of theoretical X-ray diffraction calculated into the fitness function or using the memetic strategy presented earlier, EASEA has been modified in order to take advantage of parallelism at all the different levels: SPMD, SIMD and MIMD. An asynchronous island model running on clusters of machines equipped with GPU cards, i.e. the current trend for supercomputers and cloud computing, is presented.

This last improvement of the EASEA platform allows an effortless exploitation of hierarchical massively parallel systems. It is demonstrated that supra-linear speedup over one machine and linear speedup considering clusters of different sizes are obtained. Such an island implementation over several potentially heterogeneous machines opens new horizons for various domains of application where computation time for optimization remains the principal bottleneck. Additionally, cloud computing is now coming and there is still a lack of generic optimization software allowing programs to efficiently and effortlessly run over such infrastructure. Therefore, this section shows how machines connected through a local network or the Internet may be employed adequately using EASEA, massively parallel platform.

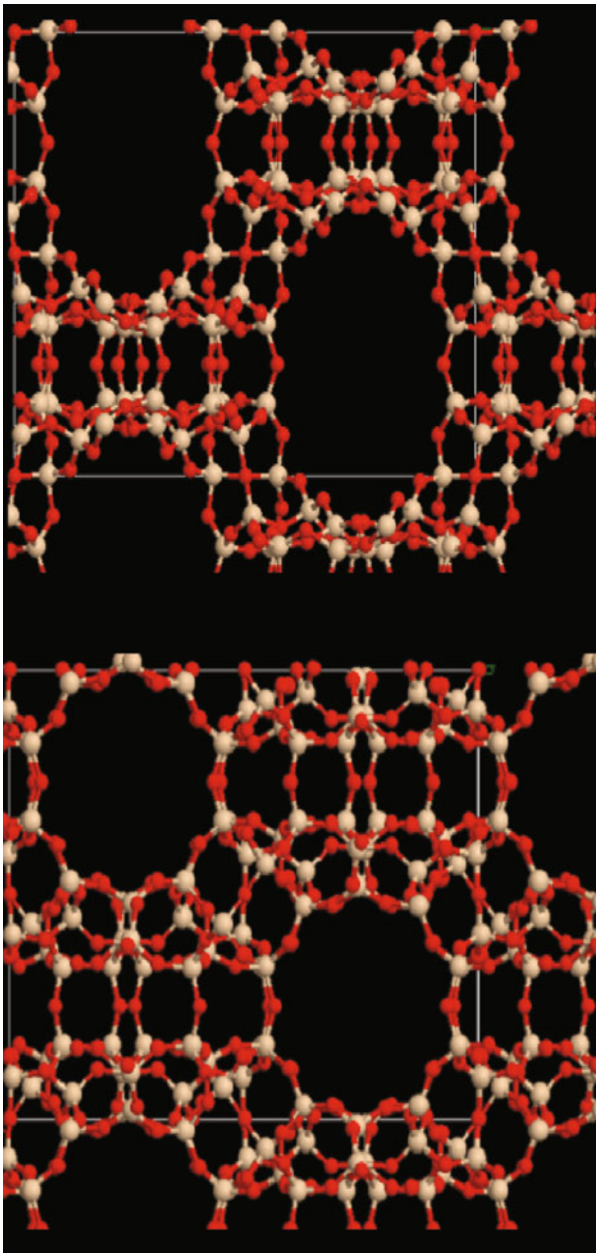


Fig. 6 View of solution #0 with 10T in unit cell C along axis ba (*top*) and solution #5 with 12T in unit cell C along axis ba (*bottom*)

Table 5 Subset of structures for the different unit cells

Unit cell	T_Atoms#	Solution#	Energy	Fitness
A	6	21	-129.43	0.0039
B	6	36	-129.42	0.0039
C	10	0	-128.86	2.488
C	12	5	-129.34	11.722

Table 6 Subset of solutions for unit cell A defined by dimensions $\{a, b, c\} = \{17.868, 13.858, 22.409\}$, angles $\{\alpha, \beta, \gamma\} = \{90, 90, 90\}$ and space group Imma (74)

T_Atoms#	Solution_#	Energy	Fitness
6	*21*	*-129.43*	*0.00391*
6	23	-129.4	0.00391
6	24	-129.25	0.00391
6	26	-129.19	0.00391
6	33	-129.32	0.00391
6	43	-129.31	0.00391
6	46	-129.36	0.00391
6	49	-129.3	0.00391
6	55	-128.91	0.00391
8	2	-129.14	0
8	6	-128.84	0
8	19	-129.09	0

Table 7 Subset of solutions for unit cell B defined by dimensions $\{a, b, c\} = \{13.85862, 17.86852, 22.41018\}$, angles $\{\alpha, \beta, \gamma\} = \{90, 90, 90\}$ and space group Imma (74)

T_Atoms#	Solution_#	Energy	Fitness
6	23	-129.02	0.00391
6	27	-129.41	0.00391
6	33	-129.26	0.00391
6	*36*	*-129.42*	*0.00391*
6	39	-129.29	0.00391
6	40	-129.19	0.00391
6	48	-129.35	0.00391
6	50	-129.41	0.00391
8	1	-129.15	0
8	9	-129.27	0
8	27	-129.02	0

Models may be expressed as nested higher-order functions and realized as the corresponding nested algorithmic skeletons. The default island model implemented by the EASEA language is quite basic. It allows individuals to periodically be sent (and received) to (from) one of several IP addresses listed in a file. The configuration

Table 8 Subset of solutions for unit cell C defined by dimensions $\{a,b,c\} = \{17.86852,\ 22.41018,\ 13.85862\}$, angles $\{\alpha,\beta,\gamma\} = \{90,90,90\}$ and space group Ima2 (46)

T_Atoms#	Solution_#	Energy	Fitness
10	2	-128.95	23.33203
10	3	-128.5	40.45703
10	4	-128.75	44.72266
12	0	-129.31	0.01172
12	3	-128.91	9.97266
12	*5*	*-129.34*	*11.72266*
12	7	-128.92	12.69141
12	13	-129	21.29297
14	0	-129.16	0
14	4	-129.03	15.20312

of the EASEA island model relies on two main parameters, the name of the file containing the IP addresses and the migration probability. Several islands usually run on the same machine according to traditional approaches. But the EASEA island model only lets a single island run per machine and relies on the establishment of connexions with other machines to form a multi-island system. Individuals are exchanged between the islands using a connectionless UDP/IP protocol. No connection implies that the process is asynchronous and therefore robust. As a matter of fact, an island can interrupt its search and then start again later on without disturbing the other islands. In addition, it will receive good immigrants from other machines allowing it to catch up with the other islands and continue the search where it left off to find good spots again. This statement also means that new islands can join the search at any moment without disturbing the overall process while no time is lost in island synchronization. Another point of interest is that EASEA produces code for any OS, and consequently heterogeneity of OSs is supported via the Internet over the entire world enabling very effective problem-solving strategies. Algorithms may be different depending on the machines they run on: slower machines can have an algorithm doing exploration while faster ones can concentrate on exploiting good spots. For this, the IP address can be repeated in the IP file allowing an implementation of weighted edges connecting machines. Note that the fast machines may not send back individuals to the slow machines so as to prevent premature convergence in the cluster of slow machines and spoil the exploration.

There are known effects of magnitude and frequency of migration. For example, limited migration between populations capitalizes on "punctuated equilibria" effect. More migrants or shorter epochs have the effect of precluding isolated evolution on separate islands. Genetic diversity vanishes quickly and the behavior approximates that of a classic EA running on the whole population. On the other hand, insufficient migration keeps the islands too far apart. The genetic richness of the neighboring populations does not have enough chance to spread out. In this regime the parallel run simulates N independent runs with population size N times smaller.

In EASEA, the migration function is called at every generation with a probability *p* set by the user. A destination island is chosen randomly amongst the list of clients. Once a destination is chosen, *n* individuals are selected amongst the population using different selection operators. The selected individuals are then serialized and sent to the destination island. When a run on an island starts, a thread is launched that is in charge of the reception of the incoming immigrants. When a new individual arrives, a selector decides who in the population will be replaced. The algorithm will check at every generation whether new individuals arrived and start the integration process for each of them. Selection may be defined in many different manners. For example, one could send a migrant to that one island whose genetic material is most different or to that island that has maintained the most diversity.

The implementation of the EASEA island model is simple, robust and versatile. Complex topologies can easily be designed using the EASEA island model. For instance, it would be simple to create a ring-shaped island model by just giving each island the address of the following island. The communication rate can be increased or decreased by changing the migration probability: for instance, the center island in a star-shaped topology may require more frequent communications with the other islands. One could also imagine modifying the migration rate based on some more elaborate strategy (increase or decrease migration with the number of generations).

4 Solving MFI Zeolite Thanks to Supra-linear Speedup

When the EASEA island model was implemented (*cf.* chapter "Automatic Parallelization of EC on GPGPUs and Clusters of GPGPU Machines with EASEA and EASEA-CLOUD", Sects. 5 and 6.2), a good challenge was to try it on our chemistry problem (see Fig. 7). We tested the island model using one of the most complicated known zeolites, the MFI framework which contains 96T in the unit cell. Note that in the version used for this test, an improved initialization is integrated. After all atoms have been randomly placed in the asymmetric unit, each atom is iteratively and randomly selected and its position is translated toward the closest atom already present which is not tetra-coordinated. The final position is the optimal distance defined at 3.07A. Also, Wyckoff positions have been integrated, allowing atoms at a distance inferior to 0.8A to be automatically placed at the corresponding special position on the symmetry element defined by the space group before the assessment of the solution. Note that the new position is virtual, i.e. calculated, in order to assess the structures viability as $\{x, y, z\}$ coordinates are not modified in the genome code. The fitness function computes an approximate value of the real energy in the system. The function is still based on geometric terms. The goal of the evolutionary algorithm is to minimize this fitness function.

A simple fully connected island model was not efficient enough to find interesting results for this very complex real-world problem. Two different clusters were created out of the 20 machines (see Fig. 8): one cluster of 16 machines that would do some exploration and one cluster of four machines with an algorithm fit enough

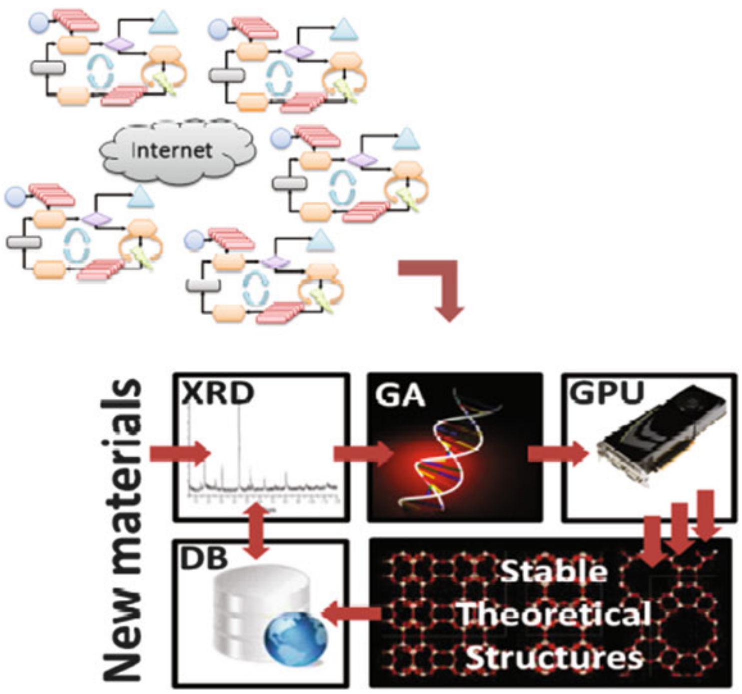


Fig. 7 Heterogeneity of machine and island configuration for efficient optimization

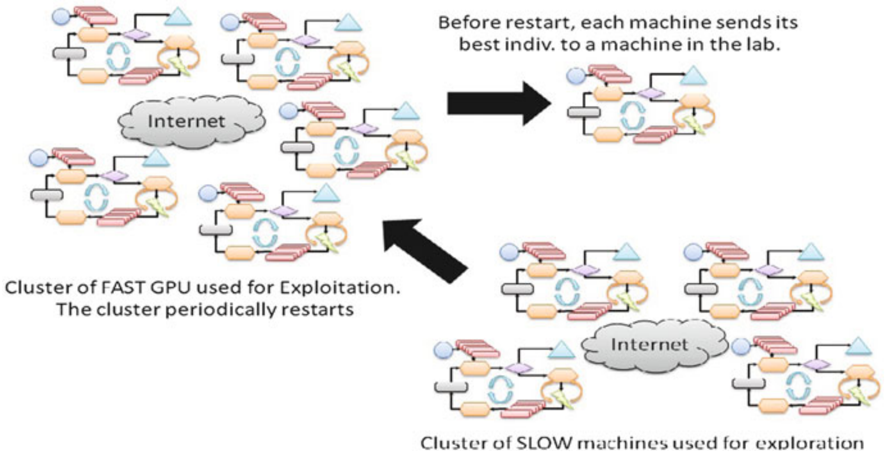

Before restart, each machine sends its best indiv. to a machine in the lab.
Internet
Cluster of FAST GPU used for Exploitation. The cluster periodically restarts
Internet
Cluster of SLOW machines used for exploration

Fig. 8 Heterogeneity of machine and island configuration for efficient optimization

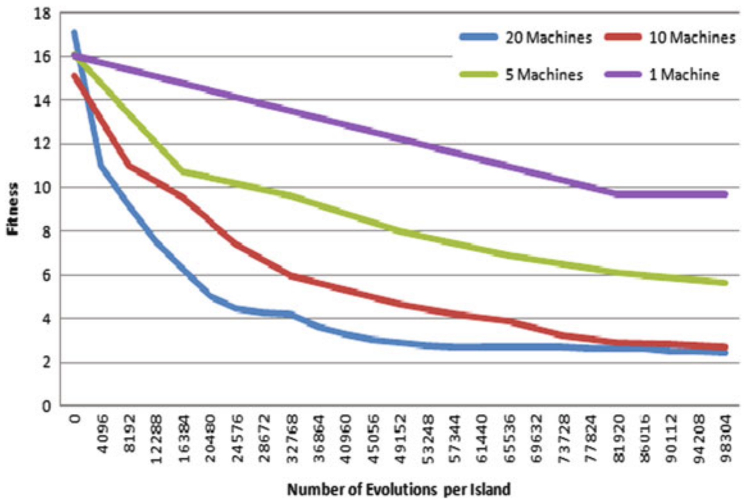


Fig. 9 Evolution of the best fitness for the MFI zeolite

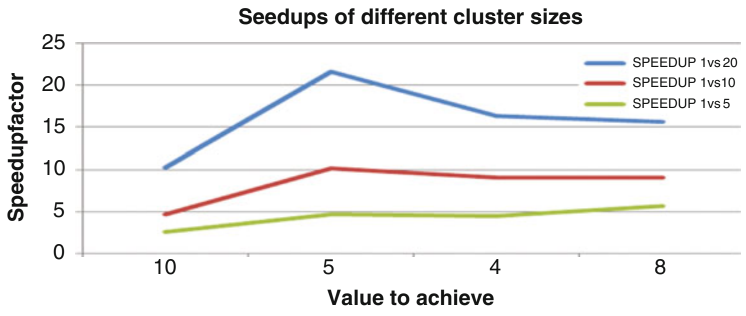


Fig. 10 Observed speedup for the real-world problem

to exploit the best spots. The machines of the first cluster periodically sent their best individuals to the four other machines, while the four machines exclusively exchanged individuals between themselves. Their main goal was to find the best local value around the individuals sent by the cluster of 16 machines. If after a while they did not manage to find any improvements, the four machines simply restarted simultaneously and waited for new suggestions coming from the 16 other machines. Periodically, the four machines sent the best individuals they found to an independent slow machine in our lab that was not part of the optimization process. It simply served as an archive and collected the best individuals found to date.

Figure 9 shows the evolution of the best fitness and Fig. 10 the respective speedup for the crystal structure prediction problem for runs on a cluster of 1, 5, 10 and 20 machines. The 20-machine cluster (and its special topology) outperformed all the others, giving better results for the same computation time. The correct structure which has a fitness value equal to zero is found as the same framework, e.g. connectivity, may be found around 2 due to flexibility and deformation of bonds.

It is interesting to note that where the speedup for 5, 10 and 20 machines is linear for a target value of 5, it decreases after this value. Indeed, apart from five machines, beyond a value of 5, the speedup for ten machines is just about 10, and the speedup for 20 machines stays around 15. This means that down to value 5, the problem is very irregular (as for the Weierstrass function), so the island model is very efficient. Beyond, it seems that the problem is less irregular, so it is not 20 times more efficient to have 20 machines work in parallel. However, if 20 machines are available, they will nevertheless go faster than five only. So, even though the irregularity of the problem seems to match well a five- (or ten-) machine configuration and less well a 20-machine configuration, 20 machines are better to have than ten only. It is only slightly less efficient.

Finally, on this problem, the 20-machine run that found the desired structures took about 40 h, but the benchmarks with 1, 5 and 10 machines was stopped long before. So even though no supra-linear speedup was observed on this problem in Fig. 10, it might well appear further down the path (the curve for one machine in Fig. 9 looks pretty flat below value 10), but there is no evidence of this. To summarize, the idea behind the island model is to split the population into many subpopulations—islands—that alternate periods of extensive isolated evolution with migration; during isolated evolution each process runs on its island a full-blown EA, and at certain times, a few individuals migrate between islands. The advantages of such an approach are that it is uniquely suited for message-passing parallel systems and it is more than a hardware accelerator for EAs. On the other hand, the price to be paid is defined by more complex design decisions due to increased number of parameters as well as the dynamics of multiple EAs running in parallel. EASEA allows automatic parallelization of the evaluation stage of evolutionary algorithms using such a scheme, which nevertheless needs to be controlled by the user.

5 Conclusion

The EASEA platform now integrates a basic yet powerful island model that allows any number of computers connected to the Internet to be turned into a cluster of machines. Experiments show that the published linear and supra-linear performance of EA algorithms can be achieved on both a benchmark and a real-world problem that a run on 20 machines contributed to solve. The architecture used to obtain these results is very close to that of Titan, the current fastest supercomputer that yields a performance of 20 Petaflops thanks to 18,699 AMD Opteron CPU computers and 18,699 NVIDIA TESLA K20X GPGPU cards, showing that evolutionary computation can not only benefit from but also exploit such machines, which prefigure our future desktop computers (Intel has announced that future CPUs will include a large number of GPU cores). It is therefore of the utmost importance to get ready for change and work on platforms such as EASEA, which will allow EC to keep producing generic optimizers for such computers.

Until recently, the huge computational power offered by GPUs was very difficult to harness due to the limitations in hardware architecture and also due to the lack of general-purpose APIs. The field of GPGPU computing is now mature, and thanks to new architectures and software such as CUDA, high performance can be achieved through the exploitation of the tremendous amount of data parallelism inherent to graphics algorithms. The next generation of GPGPU designers face the challenge of striking the correct balance between improved generality and ever-increasing performance. Modelling approaches including ab initio methods should benefit from hardware improvements as computer resources represent the current limitation when macromolecules are tackled. GPGPU clearly opens new horizons to evolutionary computation. All in all, advertized performance can only be reached if a full set of tricks is employed.

References

1. Abraham, N.L., Probert, M.I.J.: Phys. Rev. B. **73**, 224,104 (2006)
2. Akporiaye, D.E., Fjellvag, H., Halvorsen, E.N., Hustveit, J., Karlsson, A., Lillerud, K.: J. Phys. Chem. **100**, 16,641–16,646 (1996)
3. Anderson, J.A., Lorenz, C.D., Travesset, A.: J. Comput. Phys. **227**, 5342–5359 (2008)
4. Baumes, L.A., Moliner, M., Corma, A.: CrystEngComm **10**, 1321–1324 (2008)
5. Baumes, L.A., Moliner, M., Corma, A.: Chem. Eur. J. **15**, 4258–4269 (2009)
6. Boisen, M.B., Gibbs, G.V., Bukowinski, M.S.T.: Phys. Chem. Miner. **21**, 269–284 (1994)
7. Boisen, M.B., Gibbs, G.V., O'Keefe, M., Bartelmehs, K.L.: Micropor. Mesopor. Mater. **29**, 219–266 (1999)
8. Boronat, M., Concepcion, P., Corma, A., Navarro, M.T., Renz, M., Valencia, S.: Phys. Chem. Chem. Phys. **11**, 2876–2884 (2009)
9. Boronat, M., Martinez-Sanchez, C., Law, D., Corma, A.: J. Appl. Comput. Sci. **130**, 16,316–16,323 (2008)
10. Boussie, T.R., Diamond, G.M., Goh, C., Hall, K.A., LaPointeCheryl, A.M., Lund, M., Murphy, V., Shoemaker, J., Tracht, U., Turner, H., Zhang, J., Uno, T., Rosen, R., Stevens, J.: J. Am. Chem. Soc. **125**, 4306–4317 (2003)
11. Corma, A., Moliner, M., Serra, J.M., Serna, P., Diaz-Cabanas, M.J., Baumes, L.A.: Chem. Mater. **18**(14), 3287–3296 (2006)
12. Deem, M.W., Newsam, J.M.: Nature **342**, 260–262 (1989)
13. Deem, N.W., Newsam, J.M.: J. Am. Chem. Soc. **114**, 7189–7198 (1992)
14. Dress, A.W.M., Huson, D.H., Molnar, E.: Acta Crystallogr. A. **49**, 806–817 (1993)
15. Falcioni, M., Deem, M.W.: J. Chem. Phys. **110**, 1754–1766 (1999)
16. Farrusseng, D., Klanner, C., Baumes, L.A., Lengliz, M., Mirodatos, C., Schuth, F.: QSAR & Combin. Sci. **24**, 78–93 (2005)
17. Foster, M.D., Simperler, A., Bell, R.G., Friedrichs, O.D., Almeida-Paz, F.A., Klinowski, J.: Nat. Mater. **3**, 234–238 (2004)
18. Friedrichs, O.D., Dress, A.W.M., Huson, D., Klinowski, J., Mackay, A.L.: Nature **400**, 644–647 (1999)
19. Gorer, A.: U.S. Patent 6.723.678 (Symyx Technologies Inc) (2004)
20. Johnston, R.L.: Dalton Trans. **22**, 4193–4207 (2003)
21. Kirkpatrick, S., Gellat, J.C.D., P.Vecchi, M.: Science **220**, 671–680 (1983)

22. Klanner, C., Farrusseng, D., Baumes, L.A., Mirodatos, C., Schuth, F.: QSAR & Combin. Sci. **22**, 729–736 (2003)
23. Klanner, C., Farrusseng, D., Baumes, L.A., Mirodatos, C., Schuth, F.: Angew. Chem., Int. Ed. **43**(40), 5347–5349 (2004)
24. Kruger, F., Baumes, L.A., Lachiche, N., Collet, P.: In: Int. Conf. EvoStar 2010 (2010)
25. LeBail, A.: J. Appl. Cryst. **38**, 389–395 (2005)
26. Lloyd, L.D., Johnston, R.L., Salhi, S.: J. Comput. Chem. **26**, 1069–1078 (2005)
27. Maitre, O., Lachiche, N., Baumes, L.A., Corma, A., Collet, P.: In: Genetic and Evolutionary Computation Conference, GECCO'09, New York, NY, USA, 2009
28. Maitre, O., Lachiche, N., Clauss, P., Baumes, L.A., Corma, A., Collet, P.: In: 15th International Conference on Parallel and Distributed Computing, pp. 25–28 (2009)
29. Oganov, A.R., Glass, C.W.: J. Chem. Phys. **124**, 244704 (2006)
30. O'Keefe, M.: Acta Crystallogr A. **64**, 425–429 (2008)
31. O'Keefe, M., Hyde, B.G.: Crystal structures I: patterns and symmetry. Mineralogical Society of America, Washington, D.C., 1996
32. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A.E., Purcell, T.J.: Comput. Graph. Forum **26**(1), 80–113 (2007)
33. Pannetier, L., Bassas-Alsina, J., Rodriguez-Carvajal, J., Caignaert, V.: Nature **346**, 343–345 (1990)
34. Pickard, C.J., Needs, R.J.: J. Chem. Phys. **127**, 244503 (2007)
35. Roberts, C., Johnston, R.L., Wilson, N.T.: Theor. Chem. Acc. **104**, 123–130 (2000)
36. Schon, J.C., Jansen, M.: Angew. Chem. Int. Ed. **35**, 1287–1304 (1996)
37. Schon, J.C., Jansen, M.Z.: Kristallogr **216**, 307–325 (2001)
38. Schuth, F., Baumes, L.A., Clerc, F., Demuth, D., Farrusseng, D., Llamas-Galilea, J., Klanner, C., Klein, J., Martinez-Joaristi, A., Procelewska, J., Saupe, M., Schunk, S., Schwickardi, M., Strehlau, W., Zech, T.: Catal. Today **117**, 284–290 (2006)
39. Serna, P., Baumes, L.A., Moliner, M., Corma, A.: J. Catal. **258**(1), 25–34 (2008)
40. Serra, J.M., Baumes, L.A., Moliner, M., Serna, P., Corma, A.: Comb. Chem. High. Throughput Screen. **10**, 13–24 (2007)
41. Smith, J.V.: Am. Mineral. **62**, 703–709 (1977)
42. Smith, J.V.: Am. Mineral. **63**, 960–969 (1978)
43. Smith, J.V.: Am. Mineral. **64**, 551–562 (1979)
44. Sohn, K.S., Seo, S.Y., Park, H.D.: Electrochem. Solid State Lett. **4**, 26–29 (2001)
45. Stone, J.E., Phillips, J.C., Freddolino, P.L., Hardy, D.J., Trabuco, L., Schulten, K.: J. Comput. Chem. **28**, 2618–2640 (2007)
46. Teter, D.M., Gibbs, G.V., Boisen, M.B., Allan, D.C., Teter, M.P.: Phys. Rev. B. **52**, 8064–8073 (1995)
47. Treacy, M.M.J., Randall, K.H., Rao, S., Perry, J.A., Chadi, D.J.: Kristallogr **212**, 768–791 (1997)
48. Treacy, M.M.J., Rivin, I., Balkovsky, E., Randall, K.H., Foster, M.D.: Micropor. Mesopor. Mater **74**, 121–132 (2004)
49. Turner, G.W., Tedesco, E., Harris, K.D.M., Johnston, R.L., Kariuki, B.M.: Chem. Phys. Lett. **321**, 183–190 (2000)
50. Ufimtsev, I.S., Martinez, T.J.: J. Chem. Theor. Comput. **4**, 222–231 (2008)
51. Vogt, L., Olivares-Amaya, R., Kermes, S., Shao, Y., Amador-Bedolla, C., Aspuru-Guzik, A.: J. Phys. Chem. A **112**, 2049–2057 (2008)
52. Wales, D.J., Doyle, J.P.K.: J. Phys. Chem. A. **101**, 5111–5116 (1997)
53. Wales, D.J., Scheraga, H.A.: Science **285**, 1368–1372 (1999)
54. Wells, A.F.: Acta Crystallogr **7**, 535–554, 842–853 (1954)
55. Woodley, S.M.: Phys. Chem. Chem. Phys. **9**, 1070–1077 (2006)
56. Woodley, S.M., Catlow, C., Battle, P.D., Gale, J.D.: Acta Cryst. A. **58**, C196 (2002)
57. Woodley, S.M., Catlow, R.: Nat. Mater. **7**, 937 (2008)
58. Yasuda, K.: J. Chem. Theor. Comput. **4**, 1230–1236 (2008)

Artificial Chemistries on GPU

Lidia Yamamoto, Pierre Collet, and Wolfgang Banzhaf

Abstract An Artificial Chemistry is an abstract model of a chemistry that can be used to model real chemical and biological processes, as well as any natural or artificial phenomena involving interactions among objects and their transformations. It can also be used to perform computations inspired by chemistry, including heuristic optimization algorithms akin to evolutionary algorithms, among other usages.

Artificial chemistries are conceptually parallel computations, and could greatly benefit from parallel computer architectures for their simulation, especially as GPU hardware becomes widespread and affordable. However, in practice it is difficult to parallelize artificial chemistry algorithms efficiently for GPUs, particularly in the case of stochastic simulation algorithms that model individual molecular collisions and take chemical kinetics into account.

This chapter surveys the current state of the art in the techniques for parallelizing artificial chemistries on GPUs, with focus on their stochastic simulation and their applications in the evolutionary computation domain. Since this problem is far from being entirely solved to satisfaction, some suggestions for future research are also outlined.

L. Yamamoto (✉) · P. Collet
ICUBE, University of Strasbourg, Illkirch, France
e-mail: Lidia.Yamamoto@unistra.fr; Pierre.Collet@unistra.fr

W. Banzhaf
Memorial University of Newfoundland, St. John's, Canada
e-mail: banzhaf@cs.mun.ca

1 Introduction

An Artificial Chemistry (AChem) [22] is an abstract model of a chemistry in which molecules of different types (species) interact in chemical reactions, getting converted into new molecular species in the process. Artificial Chemistries can be used to model real chemical and biological systems, to perform computations [19], and as heuristic optimization techniques [7, 37] akin to evolutionary algorithms, among other usages. Moreover AChems can be combined with evolutionary algorithms to evolve chemical reaction networks able to achieve desired goals [9, 40].

In comparison with evolutionary algorithms, the optimization process within an AChem is open to a wider range of dynamics, of which evolutionary dynamics [52] is only a special case. Furthermore, evolutionary behavior may also emerge spontaneously from carefully designed AChems [4, 5, 20, 22, 24]. In this context, AChems have been used to understand the origin of evolution from a pre-evolutionary, random initial state [20, 24], as well as to model evolutionary behavior [34, 49, 64]. The use of AChems for evolutionary optimization follows as a natural step from here: for instance, in [7] machines analogous to enzymes operate on molecules encoding candidate solutions, selecting them according to a fitness criterion, and allowing the selected ones to reproduce. As a result, an evolutionary process guided by enzymes takes place, leading to a parallel optimization algorithm whose amount of parallelism is regulated by the amount of enzymes in the system. Another example of AChems for optimization is [37], in which reaction rules select and modify molecules, driving the system from a higher entropy (more disordered) to a lower entropy (more ordered) state, in which molecules encode increasingly better solutions. A related optimization chemistry has been proposed by some of us in [75], where chemical reactions creating fitter solutions are energetically favored, and thus occur with a higher probability.

Artificial chemistries are naturally parallel: several molecules may move, collide, and react simultaneously. Moreover, algorithms for the simulation of AChems tend to be computationally expensive: a large number of molecules may be present, and many of them may react in a short time interval. Therefore the use of parallel computer architectures for implementing AChem algorithms is reasonable and can be extremely helpful to reduce computation time. In particular, with the advent of affordable Graphics Processing Unit (GPU) hardware, and their general-purpose programming, it has become attractive to implement such algorithms on top of GPUs. However, the parallelization of AChems on GPUs is not straightforward in general, mainly due to the mismatch between the synchronized processing in GPU architecture based on the SIMD (single-instruction, multiple-data) design principle and the typically fluctuating chemical dynamics where myriads of reactions may occur at any time and place.

In this survey we review the current state of the art in techniques for parallelizing AChems on GPUs, with a focus on AChems that can be used for heuristic optimization akin to evolutionary algorithms. We will see that, as the chemistry exhibits more fluctuations, that is, the stronger its stochastic behavior, the more difficult it becomes to parallelize it efficiently on a GPU. However, such stochastic AChems are often necessary in domains where interactions involving a small number of molecules must be modelled (such as gene regulation, cell signalling and other cellular processes), where noise has a qualitative impact on the behavior of the system, and, in our case of heuristic optimization algorithms, where they must rely on some amount of randomness to generate novel solutions and to explore new regions of the search space. So far the challenge of parallelizing stochastic AChems on GPUs remains far from solved to satisfaction though, therefore some suggestions for future research will be outlined at the end.

We start with an introduction to artificial chemistries (Sect. 2) and their (typically sequential) simulation algorithms (Sect. 3). Section 4 provides a brief overview of GPU architecture and programming. Parallel simulation algorithms for AChems are presented in Sect. 5, with focus on GPU algorithms. Finally, Sect. 6 discusses AChems for optimization and their potential parallelization on GPUs, and Sect. 7 concludes the chapter.

2 Artificial Chemistries

An Artificial Chemistry is formally defined in [22] as a tuple (S, R, A), where S is the set of possible molecules in the system, R is the set of reaction rules governing their molecular interactions, and A is an algorithm that determines when and how the reaction rules are applied to the molecules. Numerous algorithms exist for this purpose, as discussed in Sect. 3.

The molecules in an AChem may float in a well-stirred (or well-mixed) tank reactor (spatially homogeneous AChem), may be scattered in different regions of space (spatial AChem), or may be contained within abstract compartments akin to cellular structures. In the first case, since no spatial considerations are taken into account, all molecules share the same opportunities to encounter any other molecules within the tank reactor. In the second case, space is explicitly modelled in the system, for instance, in the form of 2D surfaces or 3D volumes; so the probability of a molecule to collide with other molecules situated in its vicinity is higher than that of bumping into molecules situated far away. In the third case, compartments may contain molecules or other compartments inside, perhaps in a hierarchical manner, but their location may remain abstract, that is, molecules and compartments are not necessarily placed in a spatial structure with an explicit coordinate system. Such differences have a large impact on the dynamics of the system as well as on the algorithms used to simulate the AChem.

When they collide, molecules may react with each other (resulting in an *effective* collision) or not (*elastic* collision). In an artificial chemistry, the reaction rule set R determines which molecular species react with which. When the collision is effective, the chemical reaction rearranges the participating atoms into the products of the reaction. Therefore the quantities of each molecular species change in the process. When the amount of molecules is very large, as happens often in a real chemistry, these quantities are typically measured in terms of concentrations of each species, that is, the amount of (moles of) a given molecular species per unit of volume. Hence, from a macroscopic perspective, a system contained in a fixed volume can be fully described by the concentration dynamics of its molecules in space and time. For a well-stirred vessel containing a sufficiently large number of molecules, such concentration dynamics can be expressed as a system of ordinary differential equations (ODEs), where each equation describes the change in concentration of one particular molecular species. This ODE system can be expressed in matrix notation as follows:

$$\frac{d\mathbf{c}(t)}{dt} = M\mathbf{v}(t), \tag{1}$$

where $d\mathbf{c}(t)/dt$ is the vector of differential equations expressing how the concentration c_i of each of the species C_i changes in time t; M is the *stoichiometric matrix* of the system, which expresses the net changes in number of molecules for each species in each reaction; and $\mathbf{v}(t)$ is a vector of rates for each reaction. The rates typically follow kinetic laws from chemistry, such as the law of mass action, or other laws such as enzyme or Hill kinetics.

In order to remain simple yet close to real chemistry, the law of mass action is often employed. This law states that in a well-stirred tank reactor, the average speed (or rate) of a chemical reaction is proportional to the product of the concentrations of its reactants [3].

For example, consider the following set of chemical reactions, representing the classical Lotka–Volterra model of predator–prey interactions in a simple ecology:

$$A + B \xrightarrow{k_a} 2A \tag{2}$$

$$B \xrightarrow{k_b} 2B \tag{3}$$

$$A \xrightarrow{\mu} \emptyset. \tag{4}$$

These reactions can be interpreted as: "species A (a predator such as a fox) eats prey B (such as a rabbit) and reproduces with speed k_a; prey B reproduces with speed k_b, after eating some nutrient C (such as grass) often assumed to be abundant enough to remain at a constant level; a predator dies with rate μ." When applying the law of mass action, such reactions lead to the following ODE system:

$$\frac{da}{dt} = k_a ab - \mu a \tag{5}$$

$$\frac{db}{dt} = k_b b - k_a ab, \tag{6}$$

where a is the amount or concentration of predator A and B the amount of prey B. Equation (5) states that the population of predators grows when predators reproduce in reaction (2) (and this happens with a rate proportional to the product of concentrations of its reactants A and B) and shrinks when predators die in reaction (4). Conversely, (6) shows that the prey population decreases when they are eaten in reaction (2) and increases when they eat some grass in reaction (3). This example can be generalized to derive the ODEs corresponding to any given set of chemical reactions in an automatic way.

The law of mass action is a simplification that considers molecules as dimensionless particles moving like gas molecules in a bottle. Nevertheless this law is still useful to model the speed of chemical reactions related to many natural phenomena.

In a spatial chemistry, besides chemical reactions, the location and movement of the molecules in space must also be modelled. Sometimes individual molecules are tracked, but more often the quantities of molecules are too large for efficient individual tracking, so the movement of macroscopic amounts of molecules must be simulated instead. Molecules may simply diffuse in space, resulting in a *reaction–diffusion* process [70], they may be dragged by fluid or atmospheric currents (resulting in an advection–reaction–diffusion process [60]), or they may be actively transported by other mechanical, electrical, or chemical forces. For conciseness we focus on reaction–diffusion processes.

In a reaction–diffusion process, molecules not only react but also diffuse in space, and this can be expressed macroscopically by a system of partial differential equations (PDEs) describing the change in concentrations of substances caused by both reaction and diffusion effects combined:

$$\frac{\partial \mathbf{c}(\mathbf{p},t)}{\partial t} = \mathbf{f}(\mathbf{c}(\mathbf{p},t)) + D\nabla^2\mathbf{c}(\mathbf{p},t). \tag{7}$$

The vector $\mathbf{c}(\mathbf{p},t)$ now refers to the concentration level c_i at time t of each chemical C_i at position $\mathbf{p} = (x,y,z)$ in space. The reaction term $\mathbf{f}(\mathbf{c}(\mathbf{p},t))$ describes the reaction kinetics, like in (1), but now expressed for each point in space. The diffusion term $D\nabla^2\mathbf{c}(\mathbf{p},t)$ tells how fast each chemical substance diffuses in space. D is a matrix containing the diffusion coefficients, and ∇^2 is the Laplacian operator.

As an example, consider now that the predators and prey in the Lotka–Volterra model may wander on a two-dimensional surface, with respective speeds D_a and D_b. The corresponding PDEs then become:

$$\frac{\partial a}{\partial t} = k_a ab - \mu a + D_a\nabla^2 a \tag{8}$$

$$\frac{\partial b}{\partial t} = k_b b - k_a ab + D_b\nabla^2 b. \tag{9}$$

The reaction part remains unchanged, while the diffusion part is represented by the last term in (8) and (9). Albeit simple, the Lotka–Volterra model is well known to display a rich set of behaviors, leading sometimes to oscillations in predator and prey concentrations, explosion of prey populations, extinction of both species, waves of predators chasing prey, clustering of species, and so on. It has been studied in a wide variety of settings, including well-mixed and spatial scenarios, as well as deterministic and stochastic settings [2, 50]. Therefore we will use this example to illustrate the various algorithmic aspects discussed throughout this chapter.

In spatial chemistries, the modelled space is often divided into small lattice sites or into larger containers (subvolumes). Lattice sites typically hold one or very few molecules, while subvolumes may potentially contain a larger number of molecules. Each subvolume can be treated as a well-mixed reactor where the law of mass action applies. Diffusion is handled as a flow of molecules between neighboring reactors, with molecules being expelled from one compartment and injected in the neighboring one.

A special case of spatial organization is a multi-compartmental AChem: in such a chemistry, a population of compartments is modelled, each with chemicals and perhaps also other compartments inside, hence allowing hierarchies of compartments to be constructed recursively. A typical example of this case is Membrane Computing or P Systems [54], a formal model of computation inspired by chemistry.

3 Algorithms for Artificial Chemistries

As stated in Sect. 2, an AChem is characterized by the tuple (S, R, A). The algorithm A determines when and how the set of reaction rules R should be applied to a multiset or "soup" of molecules M currently in the system, where each element of M is an instance of an element in S.

A naive way to implement A would be to pick a few random molecules from the soup M (simulating a molecular collision), remove them from M, perform a lookup into the rule table R for a reaction rule $r \in R$ that involves the collected molecules, apply r to these molecules obtaining the reaction products (thus simulating an effective collision), and inject the products into the soup. In case no reaction rule applies, the removed molecules would be reinserted back without change (elastic collision). In case more than one reaction rule applies, the tie could be broken by a priority scheme or simply by random selection. This process would be repeated for the desired number of time steps, or until the system reaches an inert state where none of the reaction rules in R can be applied to the molecules in M. However, when the number of molecule types and/or the number of reaction rules is large, this naive algorithm can be very inefficient, wasting too much computation time on elastic collisions. Moreover, with the naive algorithm, it is difficult to simulate reaction rates accurately in order to follow rate laws such as the law of mass action or others.

For these reasons, various algorithms have been proposed to simulate chemical reactions more efficiently and accurately, by focusing the computation effort on effective reactions. These algorithms can be classified into deterministic and stochastic simulation algorithms. The deterministic algorithms work by numerically integrating the ODEs or PDEs that describe the chemical system. The stochastic algorithms take into account individual molecular collisions and calculate which reaction should occur when scheduling the reactions and updating the molecule counts accordingly.

Our focus is on GPU implementations of AChems, and it turns out that deterministic algorithms tend to be straightforward to parallelize, as will be explained in Sect. 5. The main research challenges for the parallelization of AChems on GPUs, however, lie within the stochastic algorithms. These algorithms can be exact to the level of each individual reaction or approximate in order to trade accuracy for performance. They are reviewed below.

3.1 Stochastic Simulation of Well-Mixed AChems

In order to simulate only the effective collisions, it is useful to observe that the faster a reaction on average, the more likely it is to occur within a given time interval. Moreover, the more molecules there are in the vessel, and the more they can react, the smaller the expected time interval between any two consecutive reactions, that is, the greater the amount of reactions that might be occurring (almost) simultaneously. These two rather intuitive observations are at the heart of the famous Stochastic Simulation Algorithm (SSA) method by Gillespie [29]. This is probably the most well-known algorithm for the stochastic simulation of chemical reactions in well-stirred vessels and the basis for several improved variants that followed.

Gillespie's SSA and its variants are exact methods: they simulate each individual chemical reaction, resulting in a stochastic behavior for the whole system that accurately reflects what would occur at the level of each individual molecule. When the number of molecules is very large, and the stochastic behavior must still be considered, approximate methods are an alternative: they simulate ensembles of reactions, at a granularity that can be controlled as a parameter of the simulation.

We introduce Gillespie's SSA below, together with various related methods that will provide some useful background to discuss parallelization of such algorithms on GPUs in Sect. 5. An overview of the various algorithms in this area can be found in a recent review by Gillespie [31].

3.1.1 Gillespie's SSA

The original SSA by Gillespie as described in [29] is still widely used, since it is at the same time simple, accurate, and sufficiently efficient in many cases. The algorithm is based on the notion of *propensity* (defined formally in [31]): informally, the propensity of a reaction is a value proportional to the probability that the reaction will occur within the next infinitesimal time interval, given the current state of the system.

The pseudo-code for Gillespie's SSA is displayed in Algorithm 1. For each time step iteration, it calculates which reaction should occur and when it should

Algorithm 1 Gillespie SSA (Stochastic Simulation Algorithm) [29]

```
1: multiset of molecules currently in the system: $M$
2: set of possible reactions: $R$
3: number of reactions: $m = |R|$
4: simulation time: $t = t_0$
5: **while** $desired$ **do**
6:     **for all** $r_j \in R$ **do**
7:         calculate the propensity $a_j$ of reaction $r_j$ as: $a_j = c_j h_j$
8:         $c_j$: stochastic reaction constant for $r_j$
9:         $h_j$: number of possible collision combinations leading to $r_j$
10:    **end for**
11:    $a_0 = \sum_{j=1}^{m} a_j$
12:    draw a reaction $r_j \in R$ at random (uniformly) with probability $P(r_j) = a_j/a_0$
13:    draw a random number $p$ uniformly within the unit interval (0,1).
14:    draw time interval $\tau$ from an exponential distribution: $\tau = -\frac{ln(p)}{a_0}$
15:    update current simulation time $t$ as: $t \leftarrow t + \tau$
16:    perform reaction $r_j$ by removing its educts and adding its products to multiset $M$
17: **end while**
```

occur. The next reaction to occur (reaction r_j) is chosen at random from a uniform distribution, with a probability proportional to its propensity a_j. The time interval τ after which the reaction occurs is also drawn at random, but from an exponential distribution with average $1/a_0$, such that the expected interval between reactions is $1/a_0$.

The stochastic constants c_j can be either given or derived from the kinetic rate coefficients using the examples from [29] or the generic formula exposed in [73]. The values h_j count the number of different ways to collide the subset of molecules in M required to perform reaction r_j, and their calculation is also explained in [29] (and generalized in [73]). In a nutshell, when the law of mass action applies, it suffices to take h_j as the product of the numbers of molecules of each type involved in r_j, and adjust c_j by a constant factor that takes into account the number of simultaneous collisions involved in r_j. In practice, collisions involve molecular pairs, and collisions of three or more molecules are very rare. Reactions involving multiple molecules usually combine several reaction steps in a single one for simplification.

Figure 1 shows one run of our implementation of the well-mixed Lotka–Volterra example using SSA with the same parameters as in [29]. As shown in [29], the ODE should remain stable in this case, while the stochastic simulation exhibits oscillations that sometimes amplify themselves.

The runtime for this algorithm scales linearly with the number of possible reactions $|R|$, as can be noticed from lines 6 to 10. Hence, although at each time step the algorithm picks only reactions that effectively occur, when the set R is large, a considerable amount of time can be spent in calculating all their propensities. In order to alleviate this problem, a number of variants of the original SSA and alternative algorithms have been proposed. We summarize them next.

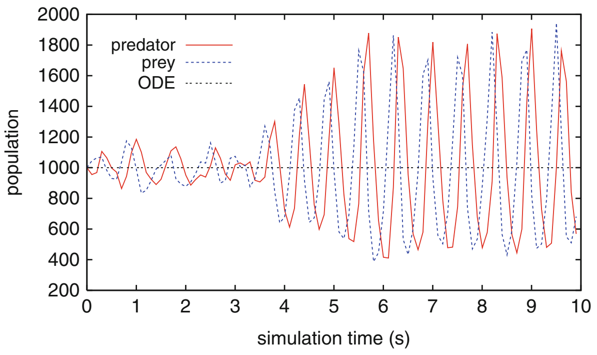


Fig. 1 Lotka–Volterra stochastic simulation using Gillespie's SSA. Reactions (2)–(4) are applied starting from initial concentrations $a_0 = b_0 = 1000$. Stochastic rate constants: $c_a = 0.01$, $c_b = 10$, $\mu = 10$ (parameters from [29])

3.1.2 Other Exact Algorithms

The original Gillespie SSA is sometimes referred to as the direct method (DM). A variant of the DM is Gillespie's First Reaction Method (FRM) [31]: instead of picking a random reaction in a propensity-proportional way, FRM draws one random τ_j interval for each reaction r_j independently, $j = 1, \ldots, m$, and executes the reaction with the smallest τ_j. The value of τ_j is calculated using the formula of line 14 of Algorithm 1 with a_j in the place of a_0. The remaining τ_j values are then discarded (because similarly to the propensity values in DM, they must be updated for the next simulation time step according to the new state of M).

The Next Reaction Method (NRM) [28] goes one step further: it sorts the reactions by increasing τ_j on a waiting list, where they are scheduled to occur at $t + \tau_j$. As the products and educts change the composition of M, only those reactions on the waiting list that were affected by the change need to be rescheduled. The waiting list is kept in the form of a binary tree for efficient lookup and update. If the number of reactions is large, and each reaction changes only a few molecules in M, NRM can significantly outperform DM. However, it is also more difficult to implement.

3.1.3 Approximate Algorithms

Whatever the simulation method chosen, simulating individual molecular reactions does not scale well to a large number of reactions, nor to a large amount of molecules. For these cases, an alternative solution is to rely on approximate algorithms. These algorithms simulate multiple reactions in a single step, trading accuracy for performance.

One of the most well-known algorithms in this category is the τ-leaping method [30]. It assumes that an interval τ can be found such that the propensities of the reactions change by a negligible amount within this interval. This assumption is called the *leap condition*: if it holds, several reactions can be fired within one simulation step (one *leap*), without updating the propensities after each individual

Algorithm 2 τ-leap [30]

```
1: let $M$, $R$, $m$, $t$ as in Algorithm 1
2: while desired do
3:     choose a suitable leap size $\tau$, for instance according to [11]
4:     $t \leftarrow t + \tau$
5:     for all $r_j \in R$ do
6:         calculate propensity $a_j$ as in Algorithm 1
7:         $\lambda = a_j \tau$
8:         draw $k_j$ from a Poisson distribution: $k_j \sim \text{Poisson}(\lambda)$
9:         fire reaction $r_j$   $k_j$ times, and update $M$ accordingly
10:    end for
11: end while
```

reaction. Therefore the algorithm takes a leap forward by several reactions at each time step, resulting in significant savings in computation time. The price to pay is a loss of accuracy, since the assumption of constant propensities over an interval is only an approximation.

The leap size τ plays a crucial role in the τ-leaping algorithm: it must be small enough such that the propensities do not change significantly within this interval and must also be large enough such that several reactions can be fired in a single leap, in order to save simulation time. Therefore a good choice of τ is essential but not always easy to make. A preliminary procedure for adjusting τ automatically during the simulation was sketched in [30]. Several improvements followed, with the method in [11] advised by [31] as being more accurate and faster than earlier attempts. The goal of all τ adjustment methods is to find the largest possible τ that still satisfies the leap condition. The method in [11] does that indirectly, by choosing a τ that leads to a bounded change in the relative amount of each reactant species in the system (this is faster than to calculate the propensities directly). When the leap size found is too small, the algorithm usually falls back to the baseline SSA.

Once τ has been chosen appropriately, the number of firings k_j for each reaction $r_j \in R$ is drawn from a Poisson distribution with mean and variance $\lambda = a_j\tau$. The state vector is then updated accordingly. The pseudo-code for the basic τ-leap algorithm is sketched in Algorithm 2. Figure 2 shows the prey populations for the Lotka–Volterra example now running over the τ-leaping algorithm with various leap sizes set as a function of the global propensity a_0: for instance, "leap 100" means $\tau = 100/a_0$. One can see that leap 10 still displays a good agreement with the expected SSA behavior, while leap 100 excessively amplifies the oscillations, and leap 500 leads to the premature extinction of the species. The run instances shown have been selected arbitrarily out of multiple runs, as instances that look representative of the stochastic behavior of the system under the chosen parameters. For instance, the majority of leap 500 runs lead to premature extinction, while only a few leap 100 runs lead to extinction, the majority of them just displaying a larger than normal oscillatory behavior. This illustrates the importance of setting the τ leap size properly, in order to satisfy the leap condition.

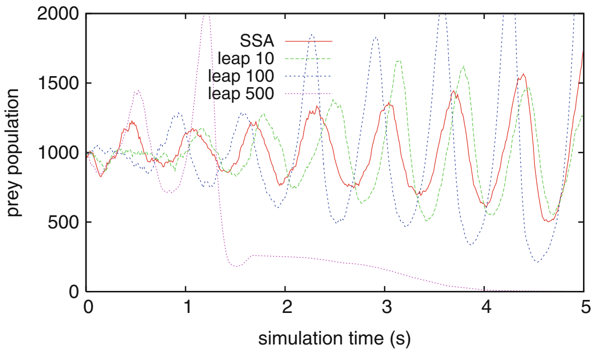


Fig. 2 Prey populations in Lotka–Volterra stochastic simulations using tau-leap with several leap sizes

Apart from setting τ properly, several other extensions of τ-leaping have been proposed. First of all, due to the potentially large leaps, Algorithm 2 may easily produce negative molecule counts if care is not taken. To solve this problem, each k_j may be constrained to a maximum by drawing it from a binomial distribution [14,69], or alternatively, only the number of firings of certain critical reactions (those with reactants approaching extinction) may be constrained [10].

Another important class of τ-leap extensions are those that deal with stiff systems [42,57]. In stiff systems, very fast reactions coexist with very slow reactions, where reaction speeds can differ sometimes by several orders of magnitude. The implicit τ-leaping method [57] deals with stiff systems by extending the implicit Euler method (used to integrate stiff ODEs) to the stochastic simulation domain. More recently, the so-called stochastic projective methods have been proposed [42], also extending upon corresponding ODE methods, with the aim of improving calculation efficiency by including a number of extrapolation steps in between leaps.

Beyond simulation performance, an important feature of τ-leaping and variants is that they can be adjusted to a wide range of behaviors, ranging from one-reaction stochastic exact simulation to a deterministic ODE integration approach: as the number of molecules approaches infinity, the stochastic variations in the Poisson distribution become negligible, and τ-leaping converges to ODE integration.

3.2 *Simulating Spatial and Compartmentalized AChems*

A well-mixed system is a simple but generally poor representation of a real system and does not scale well to a large number of interacting objects. Since the applications of purely well-mixed systems are limited, we now turn our attention to spatial systems. Here again, there are deterministic and stochastic simulation algorithms, and we focus on the stochastic methods.

Like their ODE counterpart, the simplest PDE integration method consists of simply discretizing δt in (7) into small fixed-sized time steps Δt: for each successive integration time step Δt, calculate the change in concentration $\Delta \mathbf{c}$ (in one time unit) for each molecular species in the system at each point in space using (7) and update the concentration vector $\mathbf{c}$ accordingly: $\mathbf{c}(\mathbf{p}, t + \Delta t) = \mathbf{c}(\mathbf{p}, t) + \Delta \mathbf{c} \Delta t$. This method relies on Δt being sufficiently small such that the coarse concentration changes Δc remain close enough to their ideal δc. It is not always easy to choose an appropriate Δt that strikes a good balance between execution time and accuracy, and this is especially problematic in the case of stiff systems. Therefore more sophisticated algorithms are available, and the interested reader is referred to [6] for an overview and further literature pointers.

As for the stochastic algorithms, since it would be too expensive to keep track of the movement of each individual molecule separately, the space is usually partitioned into equally sized *sites* or *subvolumes*, each holding a number of molecules. Each subvolume is treated as a well-mixed vessel with a given volume and a given coordinate in space. Diffusion is implemented as a unimolecular reaction that expels a molecule out of one vessel and injects it in another nearby vessel. The corresponding stochastic diffusion coefficients can be obtained from the deterministic ones by taking into account both the unimolecular reaction case (recall Sect. 3.1.1) and the volume of the compartment, as explained in [23]. Based on this idea, a number of spatial extensions of Gillespie's SSA for the stochastic simulation of reaction–diffusion systems have been proposed [23, 26, 63].

The Next Subvolume Method (NSM) [23] is one of the most well-known algorithms for the stochastic simulation of reaction–diffusion systems. It is a spatial extension of NRM, in which subvolumes are scheduled by event time as were single reactions in NRM. The event time for a subvolume is computed as a function of the total propensity of the reactions (including diffusion as a unimolecular reaction) within the subvolume, in the same way as the event time was computed for a single reaction in NRM as a function of its propensity. In each iteration, the next subvolume is picked from the top of the waiting list. From inside this subvolume, a random reaction (or diffusion instance) is chosen for firing in a propensity-proportional way as in the basic SSA (direct method). The propensities and event times for the concerned subvolumes (that is, the subvolume where the reaction occurred and the one that received a diffused molecule, if any) are updated accordingly, and the algorithm proceeds to the next iteration. Like in NRM, events are kept in a binary tree, to accelerate the search for the compartment within which a reaction or diffusion event should occur. In this way, the execution time for one iteration of the NSM algorithm scales logarithmically with the number of subvolumes and therefore represents a significant advantage over a linear search for subvolumes as would stem from a direct extension of SSA. NSM is implemented in the software package MesoRD [33], and several other algorithms are based on it.

The Binomial τ-leap Spatial Stochastic Simulation Algorithm (Bτ-SSSA) [46] combines binomial τ-leap and NSM in order to simulate longer time spans. In a nutshell, the algorithm operates as follows: subvolumes are scheduled by event time as in NSM. Whenever the propensities allow it, binomial τ-leap is used within a selected subvolume, in order to take a leap of several reactions in a single iteration. Otherwise, a single reaction is chosen within the subvolume using the basic direct SSA.

Rather than considering diffusion as a special kind of reaction, the Gillespie Multiparticle Method (GMP) [58] takes a different approach by splitting diffusion and reaction events in time, as follows: diffusion events advance synchronously in time across all sites, using a multiparticle lattice gas model. Between two diffusion events, reaction events are executed at each site independently, using Gillespie SSA, until the time for the next reaction reaches the time for the next diffusion event. Another diffusion event is then recomputed, and the procedure is repeated in the next iteration. Time between diffusion events is deterministic, but the particles diffuse to neighbors chosen at random. Due to the synchronous nature of the diffusion events, this algorithm is easier to parallelize [71], as will be seen in Sect. 5.

The Multi-compartmental Gillespie's Algorithm (MGA) was presented in [55], and improved variants thereof followed in [27, 59]. MGA is an extension of SSA to multiple compartments following a nested membrane hierarchy or P System [54]). P Systems are artificial chemistries intended as formal models of parallel computation, in which rules akin to chemical reactions are applied to objects akin to molecules enclosed in a membrane. Membranes can be nested, forming a hierarchical structure. Originally, P System rules would execute in a maximally parallel way, with no account for different reaction rates. When applied to systems biology however, a more realistic reaction timing must be taken into account, and MGA seeks to fill this gap. The algorithm is based on NRM: the events occurring in each membrane are ordered by firing time, and at each iteration, the algorithm picks the events with lowest time for firing, updating the affected variables accordingly.

In [65] Membrane Computing is used to evolve populations of artificial cells displaying growth and division. The volume of the compartments is not explicitly modelled in [65]. An extension of Gillespie for compartments with variable volume is introduced in [43], in order to simulate cellular growth and division. Indeed, the authors show that when the volume changes dynamically, the propensities are affected in a non-straightforward way, and adaptations to the original SSA are needed to accurately reflect this.

All the algorithms above assume that molecules are dimensionless particles moving and colliding randomly. However, in reality, intracellular environments are crowded with big macromolecules such as proteins and nucleic acids that fold in complex shapes. In such an environment, the law of mass action no longer applies. Simulations of reaction kinetics in such crowded spaces need different algorithms, such as presented in [61], which also show that fractal-like kinetics arise in such cases.

Recognizing that there is no perfect "one size fits all" algorithm for all possible applications in systems and cell biology, a meta-algorithm was proposed by [67]. The meta-algorithm is part of the E-Cell simulation environment and is able to run several potentially different sub-algorithms inside (such as ODE integration and NRM), in an integrated way. Time synchronization is achieved by taking the sub-algorithm ("stepper") with minimum scheduled time, in a manner similar to NRM. Such a meta-algorithm could also be interesting for simulation of multi-compartmental systems, where each sub-system may be simulated by a different algorithm.

4 GPU Computing in a Nutshell

In recent years, parallel computing on General-Purpose Graphics Processing Unit (GP-GPU) hardware has become an affordable and attractive alternative to traditional large and expensive computer clusters. Originally designed for high-performance image processing in computer graphics, movies, games, and related applications, the popularity of GPUs has reached domains as diverse as scientific computing for physics, astronomy, biology, chemistry, geology, and other areas; optimization and packet switching in computer networks; and genetic programming and evolutionary computation, among other domains [8, 17, 32, 44].

In this section we summarize the GPU architecture and programming very briefly, just enough for the reader to be able to follow the discussion on the parallelization of the AChem algorithms in Sect. 5. See [45] and chapter 2 of this book for a more comprehensive overview of GPU hardware and [17] for a more comprehensive overview of GPU computing applied to the modelling of biochemical systems.

Initially difficult to program due to its specialized internal architecture, GPU cards are now becoming increasingly easier to program, thanks to high-level programming frameworks such as CUDA (Compute Unified Device Architecture, by the GPU card manufacturer NVIDIA) and OpenCL (Open Computing Language, a framework designed to execute over multiple GPU platforms from different manufacturers). However, in many aspects GPU programming still remains rather low level and architecture dependent: in order to fully exploit the parallelism provided by GPUs, the programmer needs to know the internals of the GPU architecture very well and design an efficient program accordingly. Moreover, not all tasks can fully benefit from the parallelism provided by GPUs: GPUs have a SIMD (single-instruction, multiple-data) architecture, in which each single processor is able to process multiple data items in parallel using the same instruction. Therefore, the tasks that are good for GPUs are those that must handle multiple data items using the same flow of instructions.

In a nutshell, the GPU architecture is organized as follows: each GPU device contains a grid of multiprocessors (typically between 15 and 30). Each multiprocessor is organized as a set of SIMD processors. Each SIMD processor can handle a number of data items in parallel, typically 8 or 32. All the processors share a global memory space. Each multiprocessor has a local memory space that is not visible to other multiprocessors, but that can be shared among its own SIMD processors, and is called the shared memory space.

The CUDA framework tries to hide the internal organization of a GPU device, while exposing the aspects that are necessary for the programmers to optimize their algorithms to run efficiently on the GPU. As such, each card is structured as a grid of blocks. Blocks are scheduled to multiprocessors, and each block may run a certain number of threads in parallel (typically 512 or 1,024). These threads are mapped to SIMD processors in a preemptive way: each SIMD processor can handle up to N threads in parallel (N is called the *warp size* and is typically 32), provided that they all run the same instruction on different data items. If a processor gets a group of threads in which half of them is doing something different than the other half (this is called *thread divergence* and typically occurs during conditional branching), then the processor must run one group of threads first, then the second, in sequence, therefore increasing the overall execution time needed to complete the multithreaded task. Therefore, avoiding thread divergence is one technique to help improving the performance of GPU programs.

Other techniques to improve the performance have to do with memory management: the access time to global memory items is much slower than to items placed in shared memory. Therefore, placing frequently used data items on shared memory can improve performance significantly. On the other hand, these local memory items are deallocated when the GPU call returns to the host machine's CPU; therefore, they have to be reinitialized at each GPU iteration call, typically by copying them from global to local memory. Since the global memory space is much larger and data items stored there persist across GPU invocations, its use is often very convenient. A good technique to improve its access time is to retrieve items in groups of contiguous memory positions (coalescent memory access). Another important aspect to consider is the communication cost between CPU and GPU: passing data items from the CPU to the GPU and vice versa is an expensive operation and therefore should be minimized.

The synchronization among threads running on a GPU also deserves attention: threads within the same block can synchronize together, while threads in different blocks cannot. Blocks may be scheduled and preempted at any order; therefore, inter-block synchronization is problematic. It is usually achieved by returning control back to the host and paying a corresponding performance penalty.

In order to run a given program on the GPU, a so-called kernel function must be specified, in which the code for each thread is written, usually as a function of the data items that each thread should handle. After transferring all the necessary data items from the host CPU to the GPU card's global memory, the kernel is then invoked with the grid and block dimension parameters, together with any function parameters needed for the kernel to run, including the locations of the global data items to be processed. After completion, the processed data items are then transferred to the CPU where the computation results can be extracted for further analysis.

The performance of an algorithm running on GPU is usually measured in terms of the speedup of the GPU implementation with respect to a single conventional CPU (single core). In this context, a speedup of $\times 10$ (or $10\times$) means that the parallel algorithm runs ten times faster on the GPU than the corresponding sequential algorithm on a single CPU. Note that speedup figures must be interpreted with care, since they depend on the actual GPU and CPU models used in the measurements.

5 Parallelizing Artificial Chemistries on GPUs

A survey of the use of GPUs for the simulation of biological systems is presented in [17]. Several algorithms are described, including Molecular Dynamics (MD) simulations, lattice-based methods such as cellular automata (CA), multiparticle diffusion models, reaction–diffusion, and P Systems on GPU. A survey of related algorithms for parallel architectures in general can be found in [6], covering ODE integration, as well as the sequential and parallel stochastic simulation of chemical reactions. Here we focus on the chemistry part, and for this reason we do not cover methods that go down to the physics of the system, simulating molecular shapes and movements, such as MD and particle-based methods. We refer the interested reader to [17] for an overview of these other methods in the GPU context.

Among the AChem-related algorithms, numeric PDE integration, cellular automata, and other spatially oriented algorithms are the easiest to parallelize on GPUs, due to their data structure resemblance to those of the image processing tasks for which GPU hardware was originally conceived. Section 5.1 provides a brief overview of some existing approaches to the parallelization of deterministic AChem algorithms on GPU.

The parallelization difficulties increase as we move away from such highly repeated data structures with identical data handling and approach stochastic algorithms in which multiple different reactions may take place at different times. Section 5.2 provides an overview of the existing approaches to parallelize stochastic algorithms on GPU, and Sect. 5.4 points out the main remaining difficulties and potential improvements.

5.1 Deterministic Algorithms

The parallelization of ODE and PDE integration on GPUs is generally straightforward and has been applied to solve problems in systems biology and other domains [51, 60]. This is especially true for the case of numeric PDE integration, where the same differential equations and diffusion rules apply to all the points of the grid, providing a nearly perfect match to the GPU architecture: each GPU thread can take care of one point in space, and the threads in a warp can perform the same computation on different data points.

A GPU parallelization of the numeric integration of reaction–diffusion equations in three-dimensional space is described in [51]. A detailed overview of GPU approaches to parallelize the numeric integration advection–reaction–diffusion equations is presented in [60].

In previous work [77], we used a parallel implementation of reaction–diffusion on a GPU to look at large patterns, and in [76] we complemented such a GPU implementation with an evolutionary algorithm in order to search for reaction–diffusion systems forming desired patterns. In both cases the parallelization of reaction–diffusion and their automatic evolution on GPUs achieved speedups of about two orders of magnitude compared to the single CPU case and was essential to make the experiments run within a feasible duration.

5.2 Stochastic Algorithms

Several efforts to parallelize stochastic algorithms for AChems can be found in the literature, with various degrees of success. This section briefly reviews these efforts, starting with exact methods for well-mixed systems, moving on to approximate methods and spatial and compartmental systems, and finally citing some very recent work.

In [41] multiple instances of Gillespie's SSA are launched in parallel on the GPU, in order to repeat a given experiment several times. A variant of Gillespie's SSA called the logarithmic direct method (LDM) with sparse matrix update is used in order to improve performance. Speedups of up to $\times 200$ are reported. Such good performance is obviously expected given that the SSA algorithm itself is not parallelized, so no global state needs to be maintained. Similar parallelization strategies can be found in software packages such as AESS [35] and CUDA-sim [78].

A parallelization of Gillespie's FRM on GPU is proposed in [18]: the calculation of the smallest τ interval is partitioned among several tasks on the GPU, each of which takes care of a group of chemical reactions and calculates its local minimum τ value accordingly. All the local minima are collected in order to compute the global minimum, which is then used to compute the next state of the algorithm on the CPU. Several GPU-specific technical optimizations are also included in order to improve the performance of the algorithm. Despite the careful design, a weak performance gain of less than $\times 2$ speedup is reported, which can probably be attributed to the excess of synchronization needed between GPU threads and between CPU and GPU in order to compute and maintain the global state of the system.

A parallel implementation of τ-leap on GPUs is presented in [74], extending upon a previous method called parallel Coarse-Grained Monte Carlo (CGMC) for the simulation of spatially distributed phenomena on multiple scales. CGMC partitions the space into cells akin to subvolumes. Only three types of events are considered: diffusion, adsorption and desorption of molecules on cell surfaces. In parallel CGMC, the τ-leap method is extended from a well-mixed to a spatial context. A master-slave configuration is used for this purpose: a master node calculates τ and broadcasts it to the slave (cell) nodes that compute propensities and fire reactions. The locally updated molecular populations are returned to the master who collects all the values and updates the global state for the next iteration. The parallelization of CGMC on GPU [74] also works in the coarse-grained spatial context of CGMC: each GPU thread (slave) takes care of one cell and performs the leaps for the reactions inside the cell, given the interval τ provided by the master node (CPU). Experiments show that simple parallelization strategies, including the use of shared memory for storing local propensities and molecule counts, perform better on large systems than more sophisticated strategies based on a multilayered structure.

An implementation of P Systems on GPU with CUDA is shown in [12]. Membranes are assigned to blocks on the GPU, where threads apply the rules to the objects inside the membrane. Although impressive speedup figures are reported in [12], reaching values in the range of $1,000\times$, the experiments used to obtain these figures included only very simple reaction rules, essentially to refresh and to duplicate objects. These rules were executed in a maximally parallel manner, therefore obviously making full use of the GPU to perform the same operation on multiple data items as fast as possible. A more realistic case study of P Systems on GPUs is presented in [47], where an instance of the N-Queens problem (a well-known NP-hard problem) is solved with the help of a P System running on a GPU card. In [47], only the selection of the rule to be fired is done on the GPU, while the actual rule execution is left for the CPU. In this context, a speedup of about $12\times$ is reported for the selection part on GPU, with respect to the corresponding selection on CPU.

As mentioned in Sect. 3.2, the GMP algorithm is easier to parallelize because diffusion events occur in a synchronous way. Recently, the GMP algorithm was indeed parallelized on a GPU (and GPU cluster), resulting in the GPGMP algorithm [71]. GPGMP consists of a main loop on the CPU, from where three GPU kernels are invoked: first, the Gillespie kernel computes the chemical reactions according to the plain SSA; afterwards, the Diffusion kernel decides which molecules will move in which direction; finally, the Update kernel is invoked to update the molecule counts for each subvolume; and then the loop repeats, with a central update of the simulation time flow on the CPU. Speedups of up to two orders of magnitude are reported but on a GPU cluster and on very simple examples where all processors run the same kind of reaction. Later, the diffusion part of GPGMP was extended to support inhomogeneous diffusion [72], also on a GPU.

At the time of this writing, the most recent algorithm to parallelize stochastic simulations on GPUs is [38]. It parallelizes well-mixed τ-leaping on GPUs with the help of the NVIDIA Thrust library, which provides convenient parallel operations on vectors. The calculation and update of propensities, the random choice of the number of firings k_j for each reaction, and the update of the state vector are done in parallel on the GPU. A speedup of up to $60\times$ is reported with respect to an optimized sequential SSA implementation. A comparison against a sequential implementation of τ-leaping would be useful to assess the actual performance gain of the parallelization procedure more clearly.

The algorithm in [38] also includes a parallel version of SSA (direct method), used when τ-leaping must fall back to SSA. At each time step, this parallel SSA performs the reaction selection procedure, the update of propensities, and the update of the molecule counts in parallel. The computation of τ, followed by the update of the simulation time, is done by the CPU. This method is therefore efficient in systems with large numbers of reactions and molecular species, in cases where

τ-leaping cannot be applied due to a violation of the leap condition. However the performance of the parallel DM part in isolation is not reported in [38].

5.3 A Simple Stochastic AChem on GPU

As an example to illustrate the issues involved in parallelizing AChems on GPUs, we have implemented a simplified version of a spatial stochastic AChem on GPU. The algorithm combines some elements of GPGMP [71] and the GPU-based CGMC [74]. It supports both SSA and τ-leap for the reaction step, while the diffusion step is preferentially stochastic. Diffusion can also be switched off, in order to run multiple independent instances in parallel as in [35, 41, 78].

For simplification, we ignore the most difficult problem with τ-leap so far, namely, the adaptive nature of τ, and adopt a fixed τ interval that is used by all processors during the full duration of the simulation. In multiple-instance mode (without diffusion), the leap size can also be chosen as a multiple of the propensity a_0 for each instance (Sect. 3.1.3). As with the example of Fig. 2, adjusting τ manually allows us to illustrate the trade-off between speed and accuracy in the algorithm.

Algorithm 3 shows how our implementation works. Three kernels are needed: one for the reaction component (lines 5–8) and two for the diffusion component (lines 9–22 and 23–28, respectively). At each iteration, these kernels are invoked sequentially from the CPU, in order to choose the amount of reactions to be fired and of molecules to be diffused during an interval τ. Each kernel launches a number of threads in parallel; each thread takes care of one lattice cell (equal to one subvolume or one compartment). Uniformly distributed random numbers are generated using the CURAND library under CUDA, and Poisson-distributed numbers are derived from these upon demand, using a logarithmic variant of the basic Knuth's method for small lambda ($0 < \lambda \leq 10$) and a Gaussian approximation with continuity correction for $\lambda > 10$. For simplification, no special memory access optimizations are implemented, and the molecule counts for each compartment are stored in global memory.

The first kernel takes care of the reaction part. Each thread simply invokes SSA or τ-leap within its compartment. In the case of SSA, multiple iterations are involved until an interval τ is simulated. In the case of τ-leap, a single iteration with step τ is invoked; negative molecule counts are avoided by simply ignoring reactions that produce them.

The diffusion component is divided into two kernels in order to solve the critical region problem arising from the need to transport molecules from one compartment to another (controlled by another thread). The first diffusion kernel decides how many molecules go to which neighbor positions. This is done by treating diffusion as a unimolecular reaction (as in NSM) and then drawing the number of firings for this pseudo-reaction from a Poisson distribution (as in τ-leap). In this way, each thread computes the amount of molecules to be transported, subtracts this amount

Algorithm 3 Stochastic Reaction–Diffusion on GPU

```
1: $\tau$: time step interval
2: $t = t_0$
3: while *desired* do
4:     $t \leftarrow t + \tau$
5:     for all threads on GPU in parallel do
6:         run $\tau$-leap (Algorithm 2) for compartment
7:         or alternatively, run SSA (Algorithm 1) until $t$ is reached
8:     end for
9:     for all threads on GPU in parallel do
10:        $N$: set of neighbors of this cell in the lattice
11:        $M$: multiset of molecules within this cell
12:        for all $s_i \in M$ do
13:            $B_{i,j} = 0 \; \forall j \in N$
14:            $d_i$: stochastic diffusion coefficient of species $s_i$
15:            if $d_i > 0$ then
16:                $n_i$: number of molecules of type $s_i$ in subvolume
17:                $\lambda = d_i n_i \tau$
18:                draw $k \sim$ Poisson($\lambda$)
19:                $B_{i,j} = k$, where $j$ is a random neighbor of this cell
20:            end if
21:        end for
22:    end for
23:    for all threads on GPU in parallel do
24:        $N$ = set of neighbors of this cell
25:        for all $B_{i,j}$ from $N$ where $j$ is the index of this cell do
26:            $s_i \leftarrow s_i + B_{i,j}$
27:        end for
28:    end for
29: end while
```

from its local count, and writes it to a transport buffer B, indexed by molecular species and destination compartment. After all the threads have completed the first kernel, the second kernel is then launched. During this kernel, each thread inspects the neighboring transport buffers, and increases the molecule counts for each species destined to its position, by the amount given in the corresponding buffer position.

When the population of diffused molecules is large enough, the diffusion step can be easily made deterministic by taking $k = \lambda$ on line 18. A deterministic diffusion step combined with SSA within each compartment would turn Algorithm 3 essentially into GPGMP. The combination of stochastic diffusion with τ-leap leads to a variation of CGMC on GPU where any type of chemical reaction can be supported. In multiple-instance mode, the two diffusion kernels are simply not invoked.

Figure 3 shows some snapshots of the spatially extended stochastic Lotka–Volterra example run on a lattice of 128×120 cells (this is the minimum lattice size for which the GPU card is fully loaded with one cell per thread). The τ-leaping algorithm was used for the reaction step in Algorithm 3, with a leap interval of $\tau = 0.01$ s. For better visibility, the color intensities have been normalized relative

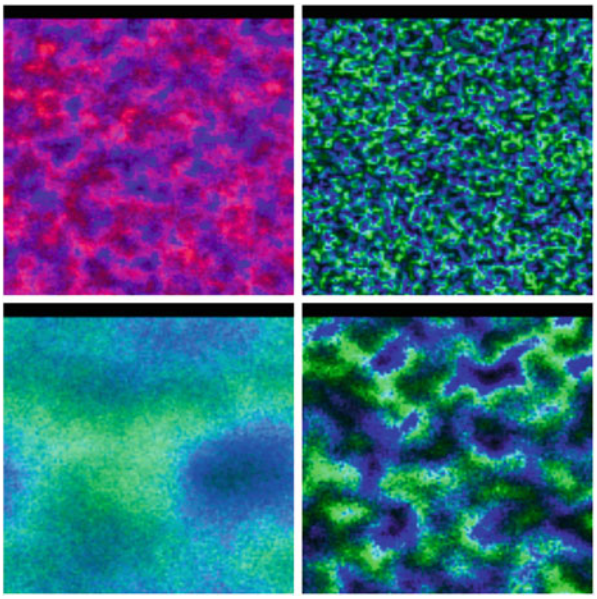


Fig. 3 Snapshots of predator (*red* or *green*) and prey (*blue*) populations in spatial stochastic Lotka–Volterra simulations with varying diffusion coefficients (D parameter) for the predator and prey species. *Top*: $D = 1.0$. *Bottom*: $D = 10.0$. Snapshots taken at the end of the simulation (at $t = 100$ s of simulated time)

to the cell with the highest amount of chemicals (depicting therefore the relative rather than absolute population values). When there is very little or no diffusion (not shown), either the predator or prey populations quickly go extinct, and no patterns are observed. For diffusion coefficients smaller than $D = 1.0$, the prey and predator populations are too scattered (due to extinct regions) or too mixed, so no pattern is formed. For the case of $D = 1.0$ (upper part of Fig. 3), some loose segregation of predator and prey populations in small clusters starts to become apparent. The clustering phenomenon seems to increase with increased diffusion: for instance, for $D = 10.0$ shown in the lower part of Fig. 3, irregular but clearly visible patterns seem to form, including dark areas with scarce populations. The position and shapes of the clustered areas change very quickly in time, but the overall qualitative behavior tends to persist throughout the simulation. However, beyond a certain diffusion rate (for instance, for $D = 100$, not shown), the system reverts back to the high extinction rates observed in the original case without diffusion, indicating that we approach the well-mixed case for the whole lattice.

Figure 4 shows that extinction can be significantly delayed by adding a small amount of diffusion to the system. Extinction here means that the population of either predator or prey gets depleted in a given cell. The fraction of extinct cells is then the fraction of cells that have either population extinct in its local compartment. The diffusion coefficient was set to the same value for both predator

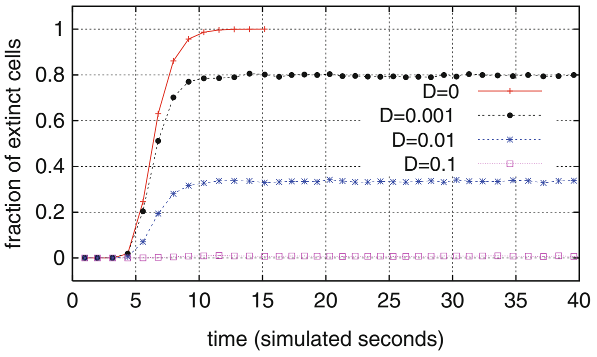


Fig. 4 Fraction of extinct cells in a spatial stochastic Lotka–Volterra simulation, for varying diffusion coefficients

and prey species. When no diffusion is present ($D = 0$), the system is reduced to the multiple-instance mode, without interaction between compartments. In this case, a quick extinction rate is observed. Moreover, the simulation stops at around $t = 15$ s due to the extinction of all cells on the grid. In contrast, by adding an amount of diffusion for predator and prey as low as $D = 0.001$, the total collapse of the simulation is avoided: the global population remains stable and is able to survive till the end of simulation (at $t = 100$ s, although the plot is truncated at $t = 40$ s for better visibility), in spite of a large number of extinct cells. Further increasing diffusion also causes a further drop in extinction rate, until it reaches zero at $D = 1.0$. The extinction rate is kept at zero for $D = 1.0$ until about $D = 10$ (not shown). A further increase in diffusion speed leads again to faster extinction, until a scenario similar to $D = 0$ is achieved (not shown). This return of the danger of extinction with increased diffusion can be delayed to higher diffusion coefficients by decreasing the global time step τ (thus increasing the accuracy of the simulation). Hence these results must be interpreted with care and in a qualitative rather than quantitative way.

Since the focus of this chapter is on the GPU parallelization of stochastic AChems, we will not delve further into the details of this specific Lotka–Volterra example. See [2, 50] for more information about stochastic predator–prey systems and [62] for an early parallel implementation thereof. The results shown in this section seem in line with the known literature in the area; however, the fixed τ step size has been carefully chosen to capture the relevant qualitative aspects of this specific example. For a more general usage, in order to avoid the computational load of adjusting τ globally, an alternative could be for each thread to choose independently whether to apply SSA or τ-leap within the given τ interval, for instance, by falling back to SSA when the leap condition cannot be satisfied in its local compartment. However this could lead to thread divergence when neighboring compartments (executed by the same SIMD processor) choose different algorithms. Once more, such options express the trade-off between simulation accuracy and computational efficiency.

Concerning computational efficiency, Table 1 shows the average speedups achieved for our GPU implementation of SSA and τ-leap, for a leap size of

Table 1 Speedups obtained for our implementations of Gillespie SSA and τ-leap on GPU (for $\tau = 10/a_0$), averaged over 100 runs

	GPU	
CPU↓	SSA	τ-Leap
SSA on CPU	60.7 ± 2.1	120.0 ± 4.0
τ-Leap on CPU	23.1 ± 0.4	45.7 ± 0.8

$\tau = 10/a_0$ in multiple-instance mode. The speedup is calculated over a runtime of 10 simulated seconds of the Lotka–Volterra example under the same initial conditions and parameters as in Fig. 1 but with different seeds for random number generation. We can see that the GPU always outperforms the CPU (speedups are always greater than one), even in the less obvious case of the GPU running SSA against the CPU running τ-leap. Obviously, the best speedup is achieved under the most favorable conditions: when the GPU runs τ-leap and the CPU runs SSA (this was also the comparison used by [38]). Note that in spite of a leap size ten times larger than the average SSA time step interval, τ-leap only reaches twice the speed of SSA. This is mainly due to the higher cost of generating Poisson-distributed random numbers for each reaction, when compared to the two uniform numbers needed at each SSA iteration. Higher speedups can be achieved with τ-leap by increasing the leap size, however, this also makes the algorithm less accurate as discussed in Sect. 3.1.3.

We have also measured the share of the diffusion part of Algorithm 3 over the total simulation time. This was done for the GPU implementation only (due to the slowness of running the spatial algorithm on a single CPU), for the same spatial Lotka–Volterra example and τ-leap with $\tau = 0.01$. Figure 5 shows the runtimes together with the fraction of time spent on the diffusion process alone (consisting of the two diffusion kernels in Algorithm 3). Diffusion takes a considerable share of the total runtime, up to about two thirds, with a peak coinciding with the parameter regions where cells tend to survive until the end of the simulation (at $t = 100$ s) (since cells with extinct populations have nothing to diffuse thus consume no computation time). Part of the responsibility for such a high computation load might be attributed to the required return to the CPU in between the two diffusion kernel invocations, plus the necessity for each cell to inspect neighbor information for local state updates (which increases memory access delay since some neighbors might not be located in nearby global memory positions allowing coalescent memory access). More efficient memory access schemes could be used to optimize the diffusion part, such as those suggested in [38, 74].

5.4 Summary and Discussion

The original Gillespie SSA is known to be hard to parallelize [17]. Indeed, it can be seen in Algorithm 1 that the global state of the system is refreshed at every time step: the propensities of all reactions are summed up to obtain the value a_0 which is then used both to calculate the time τ until the next reaction and to pick a random

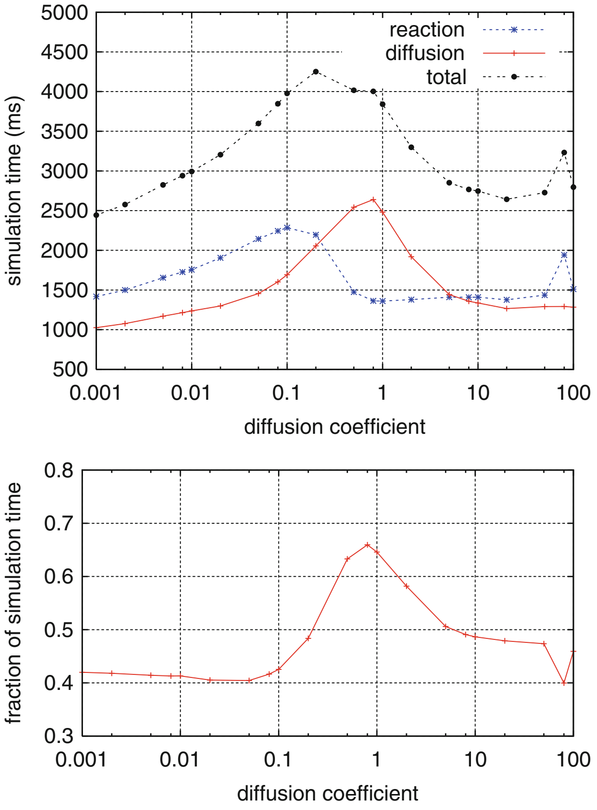


Fig. 5 Measured GPU runtimes for varying diffusion coefficients for predator and prey. *Top*: Runtime for the reaction part, diffusion part, and total runtime. *Bottom*: Fraction of the total runtime spent in the diffusion process alone

reaction to be fired, with a propensity-proportional probability. Here the global state of the system is represented by variables a_0 and τ, and their regular update can be regarded as a compulsory synchronization point in any parallel implementation of the algorithm. Such global state maintenance is what makes the algorithm difficult to parallelize: the best algorithms for parallel implementation are those that can be easily split into independent or loosely connected parts, such that each processor is able to operate alone and correctly without global state information, relying only occasionally on data exchanges with other processors.

Methods to parallelize the original SSA and other exact methods on GPU focus either on the realization of multiple instances of the algorithm in parallel [35, 41, 78] or on the parallelization of the steps to reach the global synchronization points mentioned in the previous paragraph [18, 38]. Such synchronization must be done at every iteration, usually requiring a round trip from the GPU to the CPU for that purpose. The performance of such parallel algorithms is therefore limited by these frequent synchronization events.

The parallelization of approximate methods such as τ-leaping sounds easier: once the step τ is calculated, all the k_j values for the different reactions may be chosen in parallel. In practice, however, the procedure to calculate τ can be computationally expensive, the time to generate poisson or binomial random numbers (needed to obtain the various k_j) may increase with the propensities, and the choice of each k_j value may affect the others, due to substances that participate in multiple reactions. Despite these difficulties, recently new algorithms that parallelize τ-leaping on GPUs have been proposed [38, 74]. Here again, the update of τ represents the global synchronization point that sets a limit on the amount of parallelism that can be achieved.

The parallelization of spatial methods looks even easier: subvolumes can be easily assigned to processors or threads that can run in parallel, occasionally exchanging molecules. However, here again time must be synchronized globally across all processors, since the speed of reactions happening inside each subvolume has an impact on the overall behavior of the system, for instance, on the patterns that may form within it. The GPU implementations of τ-leap [74] and GMP [71] both occur in a spatial chemistry and are examples of this category.

When time constraints are not an issue, impressive speedup figures can be obtained, for instance, when P Systems run on GPU using a maximally parallel way [12]. Since time synchronization is such a critical obstacle to parallelism, some authors [16, 36] have attempted to get around it with techniques from distributed systems applicable to discrete event simulations, essentially based on the rollback of events (undo). This method presents excessive overhead due to the amount of messages that must be exchanged between processors and the amount of events that must be undone. In the case of complex or irreversible chemical reactions with potential side effects, undo can be both computationally expensive and problematic. Needless to say, due to its message passing model, such a technique does not match the GPU architecture very well.

Load balancing is also an issue with any GPU implementation of a stochastic AChem. In order to make full use of the GPU resources, thread divergence must be avoided or at least minimized. Therefore when reactions are executed in parallel, ideally similar reactions must be grouped by thread warps, such that each warp executes a nearly identical code, minimizing divergence. Although promising, algorithms such as GPGMP [71] and the variants of τ-leap on GPU [38, 74] all have the potential problem that thread divergence in reaction execution might occur in the case of complex reaction networks composed of very different reactions. A load balancing strategy that takes into account the GPU architecture is needed. One possible strategy could be to assign groups of reactions to threads based on their similarity and the computation load required to fire them: for instance, one thread warp could receive a group of a few similar but difficult reactions, while another warp would get a group of many similar and easy reactions. How to design an efficient load balancing strategy with minimum computation overhead remains an open issue.

A technical issue of practical importance is the availability of random number generators on GPU. The CURAND library has been recently released, offering a range of pseudorandom number generators on GPUs for CUDA. Before that, researchers used their own homemade random number generator, usually a variant of the well-known Mersenne Twister algorithm [18, 41] and related pseudorandom number generators [39, 68]. Other researchers have exploited the inherent parallelism of the GPU to obtain pseudorandom number generators based on cellular automata [53], as well as true-random number generators that exploit natural sources of randomness on the GPU such as race conditions during concurrent memory access [13]. The computation efficiency of number generators other than uniform (Gaussian, Poisson, binomial) as needed by AChem algorithms still needs to be further assessed and improved on GPUs.

6 AChems for Search and Optimization

Looking at optimization from an Artificial Chemistry perspective is equivalent to explicitly modelling the optimization process as a dynamical system: candidate solutions can be regarded as molecules, and variation can be regarded as a chemical reaction resulting in the transformation of one or more molecules into mutant or recombinant types [7, 37, 75]. Various types of selection pressure may be applied. Two commonly used methods in AChems are inspired by prebiotic evolution: the first one is to kill a random individual whenever a new one is created, and the second one is to apply a dilution flow that randomly discards molecules when the maximum vessel capacity is exceeded [20, 22]. Such non-selective random elimination of individuals leads nevertheless to a selection pressure that favors molecules able to maintain themselves in the population either by self-replication or by being regenerated by others in self-maintaining chemical reaction networks akin to primitive metabolisms [5, 21]. Rather than preprogramming an evolutionary behavior like a genetic algorithm, such AChems favor the emergence of evolution out of molecular interactions in chemical reactions. For instance, the spontaneous emergence of a crossover operator is reported in [20].

Although most of the work in the AChem literature focuses on the dynamics of prebiotically inspired chemistries and their evolutionary potential, without an external objective function to be optimized, such studies are complementary to current effort in evolutionary computation, since they can shed light on the underlying mechanisms of evolution that could potentially be applied to improve or to create new optimization algorithms.

Evolving populations tend to exhibit stiff system dynamics: some mutations might cause waves of change that sweep through the populations, followed by periods of low activity. With some adaptations, the stochastic algorithms discussed in Sect. 3 can also be used to simulate evolutionary dynamics: a hybrid algorithm based on SSA and τ-leap is introduced in [79] and applied to the simulation of evolutionary dynamics of cancer development. The Next Mutation Method [48] is another recent algorithm for simulating evolutionary dynamics. Inspired by NRM and taking into account that mutations are rare events, it aims at reducing computation effort by jumping from one mutation to the next.

As a model of a simple ecology, the Lotka–Volterra example can be naturally extended to an evolutionary context. Indeed, generalized predator–prey systems involving multiple species have been proposed, including cyclic interactions (the predator of one species is the prey for another, and so forth, forming a food chain in an ecosystem), as well as mutations of one or more species, leading to adaptations in individual behavior [1, 25, 66]. In such models, predator and prey species coevolve: for instance, predators may evolve an improved ability to track and capture prey, whereas prey may evolve more efficient escape strategies.

Coevolutionary optimization algorithms [56] have been inspired by the competitive arms race resulting from natural coevolution. Recently, a spatial coevolutionary algorithm inspired by predator–prey interactions has been proposed [15], in which species evolve on a two-dimensional grid in order to solve a function approximation problem. Niches of complementary partial solutions emerge, leading to local specializations for cooperative problem solving, which nevertheless result from competitive predator–prey interactions.

Although promising, the potential of coevolution for optimization remains underexplored, mainly due to the complex dynamics emerging from species interactions. Perhaps this is an example where artificial chemistries running on top of GPUs could help, both to better understand such dynamics and to derive improved algorithms from such knowledge.

To the best of our knowledge, the parallelization of algorithms such as [15, 48, 79] on GPU has not been attempted so far. The simulation of complex ecologies and their evolution seems to be an area where the use of GPU acceleration could bring significant benefits due to the large population sizes involved, their complex interaction patterns, and their potential for an open-ended evolutionary process.

7 Conclusions

The main challenge in parallelizing AChems on GPUs is to parallelize the stochastic algorithms. These algorithms often rely on centralized information such as the total propensity of all reactions in the system and the time interval between reactions, which influence the choice of the next reaction, when it should occur, or how many times it should be fired. Moreover, they require frequent use of random number generators, a facility that only recently became available as a CUDA library.

However, parallelizing such stochastic algorithms is of paramount importance, since these algorithms tend to be computationally intensive and are important when the simulation of reactions is needed, which is often the case in artificial chemistry studies related to synthetic biology, artificial life, and evolution.

In this survey we have shown the state of the art in artificial chemistries on GPUs, discussed applications, and exemplified the usability of recently proposed GPU algorithms for stochastic spatial predator–prey systems. We have highlighted the main issues involved in the efficient parallelization of such algorithms, with attention to their application in the optimization domain. Although many problems remain to be solved, as GPU programming becomes increasingly easier, it is expected to contribute to significant advancements in the understanding of evolutionary and coevolutionary processes in models of natural ecologies or for devising new optimization algorithms able to tackle large and complex problems.

Acknowledgements We would like to thank the University of Strasbourg and the Region d'Alsace for funding a postdoctoral stay of L. Yamamoto and a visiting stay of W. Banzhaf under the EVOL grant. W. Banzhaf also acknowledges support through the NSERC Discovery Grant program under RGPIN 283304-12.

References

1. Abrams, P.A.: The evolution of predator–prey interactions: theory and evidence. Annu. Rev. Ecol. Systemat. **31**, 79–105 (2000)
2. Andrews, S.S., Bray, D.: Stochastic simulation of chemical reactions with spatial resolution and single molecule detail. Phys. Biol. **1**(3), 137–151 (2004)
3. Atkins, P., de Paula, J.: Physical Chemistry. Oxford University Press, Oxford (2002)
4. Bagley, R., Farmer, J., Fontana, W.: Evolution of a Metabolism. In: Artificial Life II, pp. 141–158. Addison-Wesley, Reading (1991)
5. Bagley, R.J., Farmer, J.: Spontaneous Emergence of a Metabolism. In: Artificial Life II, pp. 93–140. Addison-Wesley, Reading (1991)
6. Ballarini, P., Guido, R., Mazza, T., Prandi, D.: Taming the complexity of biological pathways through parallel computing. Brief. Bioinform. **10**(3), 278–288 (2009)
7. Banzhaf, W.: The "molecular" traveling salesman. Biol. Cybern. **64**, 7–14 (1990)
8. Banzhaf, W., Harding, H., Langdon, W.B., Wilson, G.: Accelerating genetic programming on graphics processing units. In: Riolo, R., Soule, T., Worzel, B. (eds.) Genetic Programming Theory and Practice VI, GEC Series, pp. 229–248. Springer, New York (2009)
9. Banzhaf, W., Lasarczyk, C.: Genetic programming of an algorithmic chemistry. In: O'Reilly, et al. (eds.) Genetic Programming Theory and Practice II, Chap. 11, vol. 8, pp. 175–190. Kluwer/Springer, Dordrecht/Berlin (2004)
10. Cao, Y., Gillespie, D.T., Petzold, L.R.: Avoiding negative populations in explicit Poisson tau-leaping. J. Chem. Phys. **123**, 054104 1–8 (2005)
11. Cao, Y., Gillespie, D., Petzold, L.: Efficient step size selection for the tau-leaping simulation method. J. Chem. Phys. **124**, 044109 1–11 (2006)
12. Cecilia, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulation of P Systems with active membranes on CUDA. In: IEEE International Workshop on High Performance Computational Systems Biology (HIBI), pp. 61–70 (2009)
13. Chan, J.J.M., Sharma, B., Lv, J., Thomas, G., Thulasiram, R., Thulasiraman, P.: True random number generator using GPUs and histogram equalization techniques. In: Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications (HPCC), pp. 161–170. IEEE Computer Society, Washington (2011)
14. Chatterjee, A., Vlachos, D.G., Katsoulakis, M.A.: Binomial distribution based τ-leap accelerated stochastic simulation. J. Chem. Phys. **122**, 024112 1–7 (2005)
15. de Boer, F.K., Hogeweg, P.: Co-evolution and ecosystem based problem solving. Ecol. Informat. **9**, 47–58 (2012)
16. Dematté, L., Mazza, T.: On parallel stochastic simulation of diffusive systems. In: Computational Methods in Systems Biology. Lecture Notes in Computer Science, vol. 5307, pp. 191–210. Springer, Berlin (2008)
17. Dematté, L., Prandi, D.: GPU computing for systems biology. Brief. Bioinform. **11**(3), 323–333 (2010)

18. Dittamo, C., Cangelosi, D.: Optimized parallel implementation of Gillespie's first reaction method on graphics processing units. In: IEEE International Conference on Computer Modeling and Simulation (ICCMS), pp. 156–161. IEEE Computer Society, Los Alamitos (2009)
19. Dittrich, P.: Chemical computing. In: Unconventional Programming Paradigms (UPP 2004). Lecture Notes in Computer Science, vol. 3566, pp. 19–32. Springer, Berlin (2005)
20. Dittrich, P., Banzhaf, W.: Self-evolution in a constructive binary string system. Artif. Life **4**, 203–220 (1998)
21. Dittrich, P., Speroni di Fenizio, P.: Chemical organization theory. Bull. Math. Biol. **69**(4), 1199–1231 (2007)
22. Dittrich, P., Ziegler, J., Banzhaf, W.: Artificial chemistries—a review. Artif. Life **7**(3), 225–275 (2001)
23. Elf, J., Ehrenberg, M.: Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases, supplementary material: next subvolume method. Proc. IEE Syst. Biol. **1**(2), 230–236 (2004)
24. Fontana, W., Buss, L.W.: 'The arrival of the fittest': toward a theory of biological organization. Bull. Math. Biol. **56**, 1–64 (1994)
25. Frachebourg, L., Krapivsky, P.L., Ben-Naim, E.: Spatial organization in cyclic Lotka–Volterra systems. Phys. Rev. E **54**, 6186–6200 (1996)
26. Fricke, T., Schnakenberg, J.: Monte-Carlo simulation of an inhomogeneous reaction–diffusion system in the biophysics of receptor cells. Z. Phys. B Condens. Matter **83**(2), 277–284 (1991)
27. García-Quismondo, M., Gutiérrez-Escudero, R., Martínez-del-Amor, M.A., Orejuela-Pinedo, E., Pérez-Hurtado, I.: P-Lingua 2.0: a software framework for cell-like P systems. Int. J. Comput. Commun. Control **IV**(3), 234–243 (2009)
28. Gibson, M.A., Bruck, J.: Efficient exact stochastic simulation of chemical systems with many species and many channels. J. Phys. Chem. A **104**(9), 1876–1889 (2000)
29. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. J. Phys. Chem. **81**(25), 2340–2361 (1977)
30. Gillespie, D.T.: Approximate accelerated stochastic simulation of chemically reacting systems. J. Chem. Phys. **115**(4), 1716–1733 (2001)
31. Gillespie, D.T.: Stochastic simulation of chemical kinetics. Ann. Rev. Phys. Chem. **58**, 35–55 (2007)
32. Han, S., Jang, K., Park, K., Moon, S.: PacketShader: a GPU-accelerated software router. SIGCOMM Comput. Commun. Rev. **40**(4), 195–206 (2010)
33. Hattne, J., Fange, D., Elf, J.: Stochastic reaction–diffusion simulation with MesoRD. Bioinformatics **21**(12), 2923–2924 (2005)
34. Hutton, T.J.: Evolvable self-reproducing cells in a two-dimensional artificial chemistry. Artif. Life **13**(1), 11–30 (2007)
35. Jenkins, D., Peterson, G.: AESS: accelerated exact stochastic simulation. Comput. Phys. Commun. **182**(12), 2580–2586 (2011)
36. Jeschke, M., Park, A., Ewald, R., Fujimoto, R., Uhrmacher, A.M.: Parallel and distributed spatial simulation of chemical reactions. In: 22nd Workshop on Principles of Advanced and Distributed Simulation, pp. 51–59. IEEE Computer Society, Washington (2008)
37. Kanada, Y.: Combinatorial problem solving sing randomized dynamic composition of production rules. In: IEEE International Conference on Evolutionary Computation, pp. 467–472 (1995)
38. Komarov, I., D'Souza, R.M., Tapia, J.-J.: Accelerating the Gillespie τ-leaping method using graphics processing units. PLoS ONE **7**(6) (2012)
39. Langdon, W.B.: A fast high quality pseudo random number generator for nVidia CUDA. In: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference (GECCO): Late Breaking Papers, pp. 2511–2514. ACM, New York (2009)
40. Lenser, T., Hinze, T., Ibrahim, B., Dittrich, P.: Towards evolutionary network reconstruction tools for systems biology. In: Proceedings of EvoBio. Lecture Notes in Computer Science, vol. 4447, pp. 132–142. Springer, Berlin (2007)

41. Li, H., Petzold, L.: Efficient parallelization of the stochastic simulation algorithm for chemically reacting systems on the graphics processing unit. Int. J. High Perform. Comput. Appl. **24**, 107–116 (2010)
42. Lu, H., Li, P.: Stochastic projective methods for simulating stiff chemical reacting systems. Comput. Phys. Commun. **183**, 1427–1442 (2012)
43. Lu, T., Volfson, D., Tsimring, L., Hasty, J.: Cellular growth and division in the Gillespie algorithm. Syst. Biol. **1**(1), 121–128 (2004)
44. Lu, P.J.: Orders-of-magnitude performance increases in GPU-accelerated correlation of images from the International Space Station. J. Real-Time Image Process. **5**(3), 179–193 (2010)
45. Maitre, O.: Understanding NVIDIA GPGPU Hardware. In: Tsutsui, S., Collet, P. (eds.) Massively Parallel Evolutionary Computation on GPGPUs. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37959-8
46. Marquez-Lago, T.T., Burrage, K.: Binomial tau-leap spatial stochastic simulation algorithm for applications in chemical kinetics. J. Chem. Phys. **127**(10) (2007)
47. Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Cecilia, J.M., Guerrero, G.D., García, J.M.: Simulation of recognizer P Systems by using manycore GPUs. In: RGNC REPORT 2/2009, Seventh Brainstorming Week on Membrane Computing, vol. II, pp. 45–57, February 2009
48. Mather, W.H., Hasty, J., Tsimring, L.S.: Fast stochastic algorithm for simulating evolutionary population dynamics. Bioinformatics **28**(9), 1230–1238 (2012)
49. McKinley, P., Cheng, B., Ofria, C., Knoester, D., Beckmann, B., Goldsby, H.: Harnessing digital evolution. IEEE Comput. **41**(1), 54–63 (2008)
50. Mobilia, M., Georgiev, I.T., Täuber, U.C.: Phase transitions and spatio-temporal fluctuations in stochastic lattice Lotka–Volterra models. J. Stat. Phys. **128**(1–2), 447–483 (2007)
51. Molnár Jr., F., Izsák, F., Mészáros, R., Lagzi, I.: Simulation of reaction-diffusion processes in three dimensions using CUDA. ArXiv e-prints, April 2010
52. Nowak, M.A.: Evolutionary Dynamics, Exploring the Equations of Life. The Belknap Press of Harvard University Press, Cambridge (2006)
53. Pang, W.-M., Wong, T.-T., Heng, P.-A.: Generating massive high-quality random numbers using GPU. In: IEEE Congress on Evolutionary Computation (CEC), IEEE World Congress on Computational Intelligence, pp. 841–847 (June 2008)
54. Paun, G.: Computing with membranes. J. Comput. Syst. Sci. **61**(1), 108–143 (2000)
55. Pérez-Jiménez, M.J., Romero-Campero, F.J.: P Systems, a new computational modelling tool for systems biology. In: Transactions on Computational Systems Biology VI. Lecture Notes in Bioinformatics, vol. 4220, pp. 176–197. Springer (2006)
56. Popovici, E., Bucci, A., Wiegand, R.P., de Jong, E.D: Coevolutionary principles. In: Handbook of Natural Computing. Springer, Berlin (2010)
57. Rathinam, M., Petzold, L.R., Cao, Y., Gillespie, D.T.: Stiffness in stochastic chemically reacting systems: the implicit tau-leaping method. J. Chem. Phys. **119**(24), 12784–12794 (2003)
58. Rodríguez, J.V., Kaandorp, J.A., Dobrzynski, M., Blom, J.G.: Spatial stochastic modelling of the phosphoenolpyruvate-dependent phosphotransferase (PTS) pathway in *Escherichia coli*. Bioinformatics **22**(15), 1895–1901 (2006)
59. Romero-Campero, F.J., Twycross, J., Camara, M., Bennett, M., Gheorghe, M., Krasnogor, N.: Modular assembly of cell systems biology models using P systems. Int. J. Found. Comput. Sci. **20**(3), 427–442 (2009)
60. Sanderson, A.R., Meyer, M.D., Kirby, R.M., Johnson, C.R.: A framework for exploring numerical solutions of advection–reaction–diffusion equations using a GPU-based approach. Comput. Vis. Sci. **12**(4), 155–170 (2009)
61. Schnell, S., Turner, T.E.: Reaction kinetics in intracellular environments with macromolecular crowding: simulations and rate laws. Prog. Biophys. Mol. Biol. **85**(2–3), 235–260 (2004)
62. Smith, M.: Using massively-parallel supercomputers to model stochastic spatial predator–prey systems. Ecol. Model. **58**(1–4), 347–367 (1991)

63. Stundzia, A.B., Lumsden, C.J.: Stochastic simulation of coupled reaction-diffusion processes. J. Comput. Phys. **127**(1), 196–207 (1996)
64. Suzuki, H.: An example of design optimization for high evolvability: string rewriting grammar. BioSystems **69**(2–3), 211–221 (2003)
65. Suzuki, Y., Fujiwara, Y., Takabayashi, J., Tanaka, H.: Artificial life applications of a class of P Systems: abstract rewriting systems on multisets. In: Workshop on Multiset Processing (WMP), pp. 299–346. Springer, London (2001)
66. Szabó, G., Czárán, T.: Phase transition in a spatial Lotka–Volterra model. Phys. Rev. E **63**, 061904 (2001)
67. Takahashi, K., Kaizu, K., Hu, B., Tomita, M.: A multi-algorithm, multi-timescale method for cell simulation. Bioinformatics **20**(4), 538–546 (2004)
68. Thomas, D.B., Howes, L., Luk, W.: A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), pp. 63–72. ACM, New York (2009)
69. Tian, T., Burrage, K.: Binomial leap methods for simulating stochastic chemical kinetics. J. Chem. Phys. **121**(21), 10356–10364 (2004)
70. Turing, A.M.: The chemical basis of morphogenesis. Philos. Trans. R. Soc. Lond. B **327**, 37–72 (1952)
71. Vigelius, M., Lane, A., Meyer, B.: Accelerating reaction–diffusion simulations with general-purpose graphics processing units. Bioinformatics **27**(2), 288–290 (2011)
72. Vigelius, M., Meyer, B.: Multi-dimensional, mesoscopic Monte Carlo simulations of inhomogeneous reaction-drift-diffusion systems on graphics-processing units. PLoS ONE, 7(4) (2012)
73. Wolkenhauer, O., Ullah, M., Kolch, W., Cho, K.-H.: Modelling and simulation of intracellular dynamics: choosing an appropriate framework. IEEE Trans. Nano-Biosci. **3**(3), 200–207 (2004)
74. Xu, L., Taufer, M., Collins, S., Vlachos, D.: Parallelization of tau-leap coarse-grained Monte Carlo simulations on GPUs. In: IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–9 (April 2010)
75. Yamamoto, L., Banzhaf, W.: Catalytic search in dynamic environments. In: Artificial Life XII, Proceedings of the Twelfth International Conference on the Synthesis and Simulation of Living Systems, pp. 277–285. MIT Press, Cambridge (August 2010)
76. Yamamoto, L., Banzhaf, W., Collet, P.: Evolving reaction–diffusion systems on GPU. In: Proceedings of XV Portuguese Conference on Artificial Intelligence (EPIA), Thematic Track on Artificial Life and Evolutionary Algorithms (ALEA). Lecture Notes in Artificial Intelligence, vol. 7026, pp. 208–223. Springer, Berlin (2011)
77. Yamamoto, L., Miorandi, D., Collet, P., Banzhaf, W.: Recovery properties of distributed cluster head election using reaction–diffusion. Swarm Intell. **5**(3–4), 225–255 (2011)
78. Zhou, Y., Liepe, J., Sheng, X., Stumpf, M., Barnes, C.: GPU accelerated biochemical network simulation. Bioinformatics **27**(6), 874–876 (2011) [Applications Note].
79. Zhu, T., Hu, Y., Ma, Z.-M., Zhang, D.-X., Li, T., Yang, Z.: Efficient simulation under a population genetics model of carcinogenesis. Bioinformatics **27**(6), 837–843 (2011)

Acceleration of Genetic Algorithms for Sudoku Solution on Many-Core Processors

Yuji Sato, Naohiro Hasegawa, and Mikiko Sato

Abstract In this chapter, we use the problem of solving Sudoku puzzles to demonstrate the possibility of achieving practical processing time through the use of many-core processors for parallel processing in the application of evolutionary computation. To increase accuracy, we propose a genetic operation that takes building-block linkage into account. As a parallel processing model for higher performance, we use a multiple-population coarse-grained genetic algorithm (GA) model to counter initial value dependence under the condition of a limited number of individuals. The genetic manipulation is also accelerated by the parallel processing of threads. In an evaluation using even very difficult problems, we show that execution times of several tens of seconds and several seconds can be obtained by parallel processing with the Intel Core i7 and NVIDIA GTX 460, respectively, and that a correct solution rate of 100 % can be achieved in either case. In addition, genetic operations that take linkage into account are suited to fine-grained parallelization and thus may result in an even higher performance.

Y. Sato (✉)
Faculty of Computer and Information Sciences, Hosei University, 3-7-2 Kajino-cho, Koganei-shi, Tokyo 184-8584, Japan
e-mail: yuji@k.hosei.ac.jp

N. Hasegawa
Graduate School of Computer and Information Sciences, Hosei University, 3-7-2 Kajino-cho, Koganei-shi, Tokyo 184-8584, Japan
e-mail: naohiro.hasegawa.af@stu.hosei.ac.jp

M. Sato
Faculty of Engineering, Tokyo University of Agriculture and Technology, 2-24-16 Naka-cho Koganei-shi, Tokyo 184-8588, Japan
e-mail: mikiko@namikilab.tuat.ac.jp

1 Introduction

Research on the implementation of evolutionary computation on massively parallel computing systems to attain faster processing [1–5] has been going on since about 1990, but it has not come into widespread use. On the other hand, multi-core processors, graphics processing units (GPU), and other such many-core processors have been coming into wide use in ordinary personal computers in recent years. The features of many-core processors include suitability for small- and medium-scale parallelization from several to several hundreds of nodes and low-cost compared to massively parallel computing systems. This environment has stimulated research on parallelization of evolutionary computation on many-core processors [6–10]. These current reports, however, focus on benchmark tests for evolutionary computation using typical GPUs. The objective of our research is to use an actual and practical problem to demonstrate that practical processing time is possible through the use of a GPU for parallelization of evolutionary computation, even for problems for which the use of evolutionary computation has not been investigated previously because of the processing time problem.

As the first step towards that objective, we take the problem of solving Sudoku puzzles [11] and investigate acceleration of the processing with a GPU. The reasons for this approach are listed below:

1. Sudoku puzzles are popular throughout the world.
2. Assuming a single-core processor, the processing time for genetic algorithms is much higher than for backtracking algorithms [12]. On the other hand, backtracking algorithms pose problems for parallelization, whereas evolutionary computation is suitable for parallelization. Therefore, increasing the number of GPU cores may make the processing time for genetic algorithms equal to or less than that for backtracking algorithms.
3. The use of multi-core processors has recently expanded to familiar environments like desktop PCs and laptop computers, and it has become easy to experiment with parallelization programs on multi-core processors through thread programming. GPUs are designed for the processing of computer graphics in games. But research on general-purpose computation on graphics processing units (GPGPU) has begun, and GPUs can be used to support solving a logical game.
4. Although the processing time for the backtracking algorithm increases exponentially as the puzzle size increases from 9×9 to 16×16, the fact that a genetic algorithm (GA) is a stochastic search algorithm opens up the possibility of reversing the processing time ratio.

High-speed evolutionary computation by a GPU requires the design of a parallel program that is dependent on the number of cores and memory capacity in the GPU. This means adjusting the degree of parallelization and the amount of processing allocated to each task according to GPU specifications. In the case of Core i7, however, we focus only on the degree of parallelization afforded by the number of cores in a multi-core processor and propose a method for speeding up Sudoku problem solving by general-purpose thread programming conforming to POSIX specifications [13].

In this chapter, we show the possibility of a large reduction in processing time for evolutionary computation by parallelization using a many-core processor. In the following section (Sect. 2), we explain the rule of Sudoku. In Sect. 3, we show an improvement in the accuracy of Sudoku puzzle solution by using a genetic operation that takes building-block linkage into account [14]. In Sect. 4, we propose an implementation of a parallel genetic algorithm on a GPU. Section 5 describes a comparative evaluation of the solution of a difficult Sudoku puzzle executed on a CPU and on a many-core processor. Section 6 presents a discussion and Sect. 7 concludes this chapter.

2 The Rules of Sudoku

The Sudoku rules are explained in Fig. 1. General Sudoku puzzles consist of a 9×9 matrix of square cells, some of which already contain a numeral from 1 to 9. The arrangement of given numerals when the puzzle is presented is called the starting point. In Fig. 1, it contains 24 nonsymmetrical given numbers, and the correct number for the other 57 points should be solved. The degree of difficulty varies with the number of given numerals and their placement. Basically, fewer given numerals means a higher number of combinations among which the solution must be found and so raise the degree of difficulty. But there are about 15–20 factors that have an effect on difficulty rating [11]. A Sudoku puzzle is completed by filling in all of the empty cells with numerals 1–9, but no row or column and no 3×3 sub-block (the sub-blocks are bound by heavy lines in Fig. 1) may contain more than one of any numeral. An example solution to the example Sudoku puzzle given in Fig. 1 is shown in Fig. 2. In this figure, the given numbers are marked in boldface.

For these basic puzzles, methods such as backtracking, which counts up all possible combinations in the solution, and the meta-heuristic approach [15] are effective. There also exist some effective algorithms to solve Sudoku puzzles [16–18] and that are faster than GA. On the other hand, there are also many variations of Sudoku. Some are puzzles of larger size, such as 16×16 or 25×25. Others impose additional constraints, such as not permitting the same numeral to appear more than once in diagonals or in special sets of 9 cells that have the same color. For larger puzzles such as 16×16 or 25×25, GA or other stochastic search methods may be effective. Methods for speeding up evolutionary computations through implementations on graphics processing units (GPU) may be also effective.

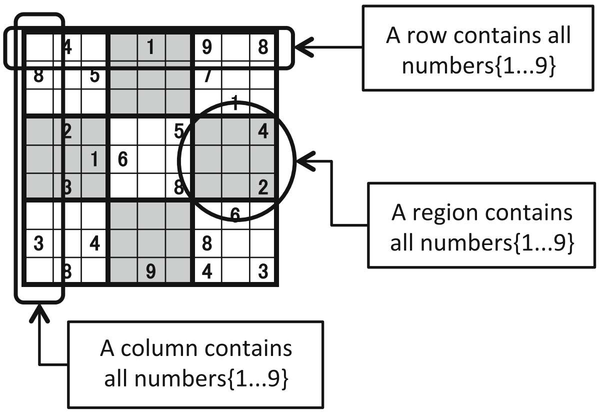
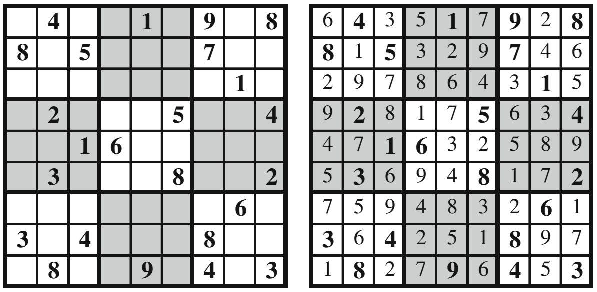

A row contains all numbers{1...9}
A region contains all numbers{1...9}
A column contains all numbers{1...9}

Fig. 1 An example of Sudoku puzzles. Each cell of 24 positions contains a given number; the other position should be solved



	4			1		9		8
8		5				7		
							1	
	2				5			4
		1	6					
	3				8			2
							6	
3		4				8		
	8			9		4		3

6	**4**	3	5	**1**	7	**9**	2	**8**
8	1	**5**	3	2	9	**7**	4	6
2	9	7	8	6	4	3	**1**	5
9	**2**	8	1	7	**5**	6	3	**4**
4	7	**1**	**6**	3	2	5	8	9
5	**3**	6	9	4	**8**	1	7	**2**
7	5	9	4	8	3	2	**6**	1
3	6	**4**	2	5	1	**8**	9	7
1	**8**	2	7	**9**	6	**4**	5	**3**

Fig. 2 A solution for the Sudoku puzzles given in Fig 1. The given numbers are marked in *boldface*

3 Improved Accuracy in Sudoku Solution Using Genetic Operation That Takes Linkage into Account

3.1 Genetic Operations That Takes Linkage into Account

A number of studies on application of GA to solving Sudoku have already been made. On the other hand, there seem to be relatively few scientific papers. References [19, 20], for example, define a one-dimensional chromosome that has a total length of 81 integer numbers and consists of linked sub-chromosomes for each 3×3 sub-block of the puzzle and applies uniform crossover in which the crossover positions are limited to the links between sub-blocks. References [21, 22] compare the effectiveness for different crossover methods, including one-point crossover that limit crossover points to links between sub-blocks, two-point crossover, crossover

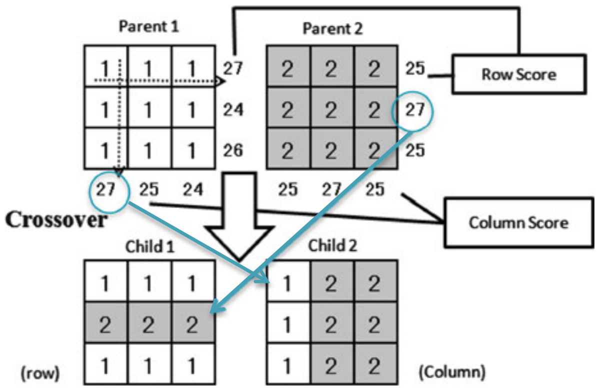


Fig. 3 An example of the crossover considering the rows or the columns that constitute the sub-blocks

in units of row or column, and permutation-oriented crossover operation. In these examples, optimum solutions to simple puzzles are easily found, but the optimum solutions for difficult puzzles in which the starting point has few givens are often not obtainable in realistic time. We believe the reason for the failure of this design is that the main GA operation, crossover, tends to destroy highly fit schemata (BB) [23]. To avoid that problem, we defined 9×9 two-dimensional arrays as the GA chromosome and proposed a crossover operation [14] that takes building-block linkage into account. The fitness function, Eq. (1), is based on the rule that there can be no more than one of any numeral in a row or column. The score of a row (column) is the number of different elements in the row (column). The score f of a given individual x is defined as

$$f(x) = \sum_{i=1}^{9} |g_i| + \sum_{j=1}^{9} |h_j| \tag{1}$$

where $|g_i|$ ($|h_j|$) denotes the number of different numerals in the ith row (jth column). Therefore, the maximum score of the fitness function $f(x)$ becomes 162.

An example of this crossover is shown in Fig. 3. In this figure, we assumed that the highest score of each row or column is 9. Therefore, the highest score of each row or column in sub-blocks becomes 27. Child 1 inherits the row information from parent 1, parent 2, and parent 1 in order from top to bottom. Child 2 inherits the column information from parent 1, parent 2, and parent 2 in order from left to right. Mutations are performed for each sub-block. Two numerals within a sub-block that are not given in the starting point are selected randomly, and their positions are swapped. We added a simple local search function in which multiple child candidates are generated when mutation occurs, and the candidate that has the highest score is selected as the child. These experiments use tournament selection.

a

No. 1

		9				1		
2	1	7				3	6	8
			2		7			
	6	4	1		3	5	8	
	7						3	
1	5		4	2	8		7	9
			5	8	9			
4	8	5				2	9	3
		6	3		2	8		

b

No. 11

2	9		7		1			
5	3			6		1		
		6	3				4	
			5	9				4
	1	5			4	6	8	9
			1	8				3
		2	6				9	
3	6			4		7		
9	4		8		5			

c

No. 27

		1		8				
			3		4	7	5	
	6			5				
8		6			2	3	4	9
		9						
3		4			8	1	7	2
	3			7				
			8		1	5	6	
		2		3				

d

No. 29

	1		5		6		2	
3								6
		9	1		4	5		
	9			1			4	
	7		3		2		5	
	3			8			6	
		3	2		7	1		
9								2
	5		6		1		8	

e

No. 77

5								9
9			8		5			6
3			9		7			5
				9				
	9			1			2	
	3	8				9	4	
4								2
		3	5		9	6		
		2	4		1	3		

f

No. 106

			4		7			
		1				7		
4								3
	2		3		9		4	
	4			1			9	
		6				8		
5								8
	8	4		6		5	3	
3								2

Fig. 4 The puzzles used to investigate the effectiveness of the proposed genetic operations. (**a**) Easy level Sudoku (Givens: 38). (**b**) Easy level Sudoku (Givens: 34). (**c**) Medium level Sudoku (Givens: 30). (**d**) Medium level Sudoku (Givens: 29). (**e**) Difficult level Sudoku (Givens: 28). (**f**) Difficult level Sudoku (Givens: 24)

3.2 *Sudoku Solution Accuracy by GA*

For the puzzles used to investigate the effectiveness of the genetic operations proposed in [14], we selected two puzzles from each level of difficulty in the puzzle set from a book [14]: puzzles 1 and 11 from the easy level, 29 and 27 from the intermediate level, and 77 and 106 from the difficult level, for a total of six puzzles. We also used the particularly super-difficult Sudoku puzzles introduced in [24]. An example of the puzzles used in the experiment is shown in Figs. 4 and 5, respectively. The experimental parameters are population size, 150; number of child candidates/parents, 2; crossover rate, 0.3; mutation rate, 0.3; and tournament size, 3.

The relation between the number of givens in the starting point and the number of generations required to reach the optimum solution is shown in Table 1 and Fig. 6.

For the three cases in which only mutation is applied (a kind of random search), when mutation and the proposed crossover method are applied (mut+cross), and when the local search improvement measure is applied in addition to mutation and crossover (mut+cross+LS), the tests were run 100 times and the averages

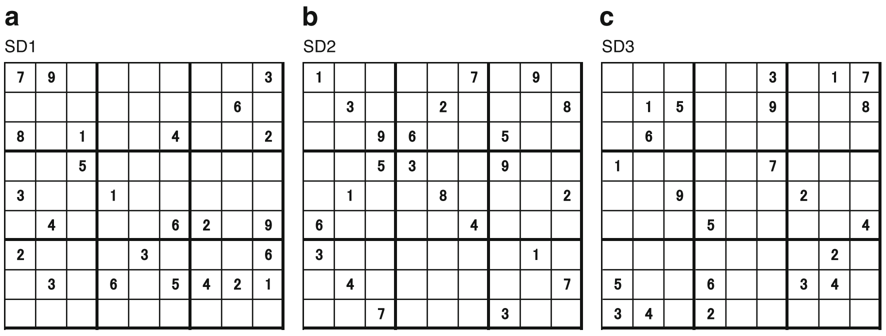


Fig. 5 The puzzles used for the particularly super-difficult Sudoku puzzles. (**a**) GA generated Super difficult Sudoku (Givens: 24). (**b**) AI Escarcot claim to be the most difficult Sudoku (Givens: 23). (**c**) Super difficult Sudoku from www.sudoku.com (Givens: 22)

Table 1 The comparison of how effectively GA finds solutions for the Sudoku puzzles with different difficulty ratings

Difficulty rating	Givens	mut + cross + LS		mut + cross		Swap mutation	
		Count	Average	Count	Average	Count	Average
Easy(#1)	38	100	62	100	105	100	105
Easy(#11)	34	100	137	100	247	100	247
Medium(#27)	30	100	910	100	2,247	100	2,247
Medium(#29)	29	100	3,193	100	6,609	100	6,609
Difficult(#77)	28	100	9,482	100	20,658	100	20,658
Difficult(#106)	24	96	2,6825	74	56,428	74	56,428

of the results were compared. The termination point for the search was 100,000 generations. If a solution was not obtained before 100,000 generations, the result was displayed as 100,000 generations. When the search is terminated at 100,000 generations, the proportion of obtaining an optimum solution for a difficult puzzle was clearly improved by adding the proposed crossover technique to the mutation and improved even further by adding the local search function. The mean number of generations until a solution is obtained is also reduced.

In Table 2, the number of times the optimum solution was obtained in 100 test runs using the super-difficult Sudoku puzzles is shown [24]. Without any trial limitation, the method was solved every time. On the other hand, when using a limit of 100,000 trials, our method was solved 99 times, 83 times, and 74 times out of 100 test runs for three super-difficult problems, respectively.

On the other hand, Fig. 7 shows the relation of the average number of generations and dispersion. For puzzles that have the same number of initial givens, there is a dependence on the locations of the givens, and a large variance is seen in the mean number of generations needed to obtain the optimum solution. Furthermore, for difficult puzzles that provide few initial givens, there were cases in which a solution was not obtained even when the search termination point was set to 100,000

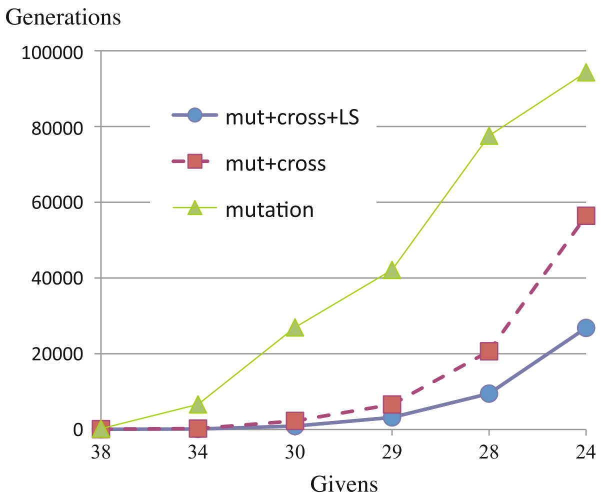


Fig. 6 Relationship between givens and the average number of GA generations needed to find the solution

Table 2 The number of times the optimum solution was obtained in 100 test runs using super-difficult problems. The numbers represents how many times out of 100 test runs each method reached the optimum

Sudoku puzzle	SD1	SD2	SD3
Count	99	83	74

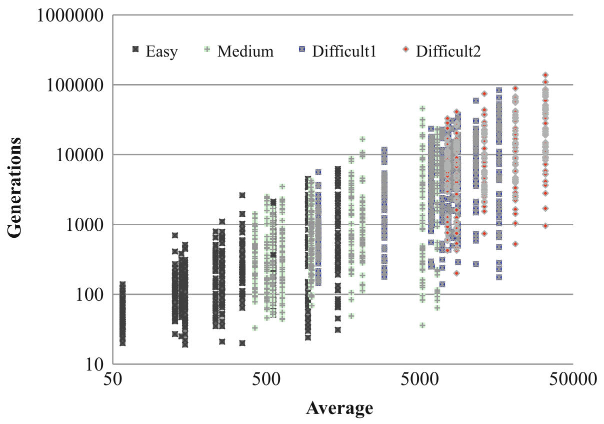


Fig. 7 The difficulty order of tested Sudoku. The minimum and maximum generations needed to solve each Sudoku from 100 test runs as a function of generations needed

generations. The reason for that result is considered to be that the search scope for the solution to a difficult puzzle is large and there exist many high-scored local solutions that are far from the optimum solution. Another possibility is that there are puzzles for which the search scope is too broad and there is a dependence on the initial values. The processing time was still very poor compared to the backtracking algorithm.

4 Accelerating Evolutionary Computation with Many-Core Architecture

4.1 System Architecture for Sudoku Solution

4.1.1 GTX 460 and CUDA Programming

Here we describe parallel processing of a program implemented on the GeForce GTX 460, a commercial GPU from the NVIDIA Corporation that uses the CUDA architecture. The GTX 460 comprises seven streaming multiprocessors (SMs). Each SM has 48 CUDA cores and comprises up to 48 KB of shared memory. Data reading and writing between SMs is accomplished via a large-capacity global memory (1 GB). A part of the global memory is a constant read-only memory of 64 KB. The processors within SM can read from and write to the shared memory at high speed, but the data reading and writing between SMs and the global memory is slow. Therefore, the parallelization of evolutionary computation must be implemented with full consideration given to that feature. The basic CUDA operations are broadly grouped into the four classes: (1) reserving GPU memory, (2) data transfer from CPU to GPU, (3) parallel execution on the cores of the GPU, and (4) data transfer from the GPU to the CPU.

CUDA has three units of processing: A thread corresponds to a single process, a block is a number of threads, and a number of blocks of the same size constitute a grid. In CUDA, a thread array of up to three dimensions can be made into a block, and a grid can include an array of blocks of up to two dimensions. The unit for the execute instruction from the host is the grid. All of the threads in a grid are executed by the same program, which is called the kernel. The CUDA programming model is a kind of multi-thread model. Each thread is allocated an element in a data array, and the data array serves in the management of the parallel execution of those threads. Threads within the same block share the shared memory inside the SM, so the number of threads within a block is limited to 1,024.

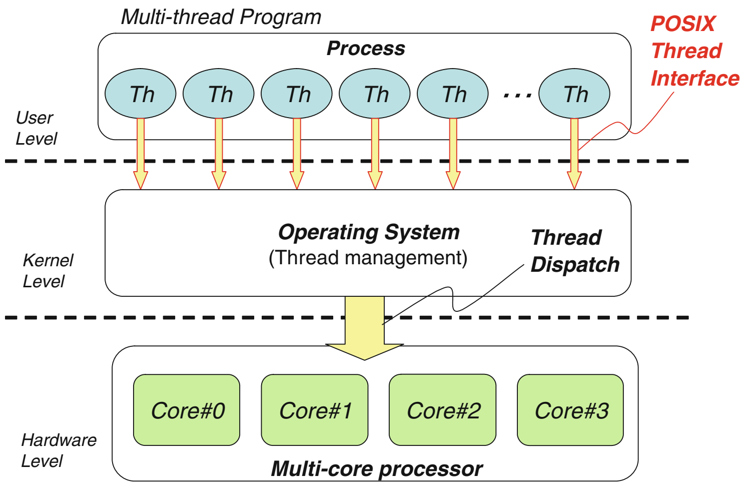


Fig. 8 The system architecture for multi-core processors

4.1.2 System Architecture for Core i7

A conceptual diagram of the homogeneous multi-core processor architecture and system software targeted by this research is shown in Fig. 8. Intel, AMD, and other semiconductor companies have recently been marketing quad-core products for a wide range of computers from PCs to servers. In this research, we target a homogeneous multi-core processor that has recently come to be used in PCs and attempt to speed up Sudoku puzzle solving on a commonly available system. The OS (Ubuntu 10.04) used in our research provides a POSIX thread interface as an application programming interface (API) and can execute a multi-thread program on a multi-core processor.

4.2 Parallel GA Model and Implementation for Many-Core Architecture

4.2.1 Parallel GA Model and Implementation for GPU Computation

Figure 9 shows the parallel GA model for GPU computation. Because the grid is the unit of execution for instructions from the host, we conducted experiments with seven blocks in a grid to match the number of SM and with the number of threads in a block equal to three times the number of individuals ($3 \times N$) for parallel processing in units of rows or columns that consist of Sudoku region blocks. We allocate the population pools PP and WP and some working pools to the shared memory of each

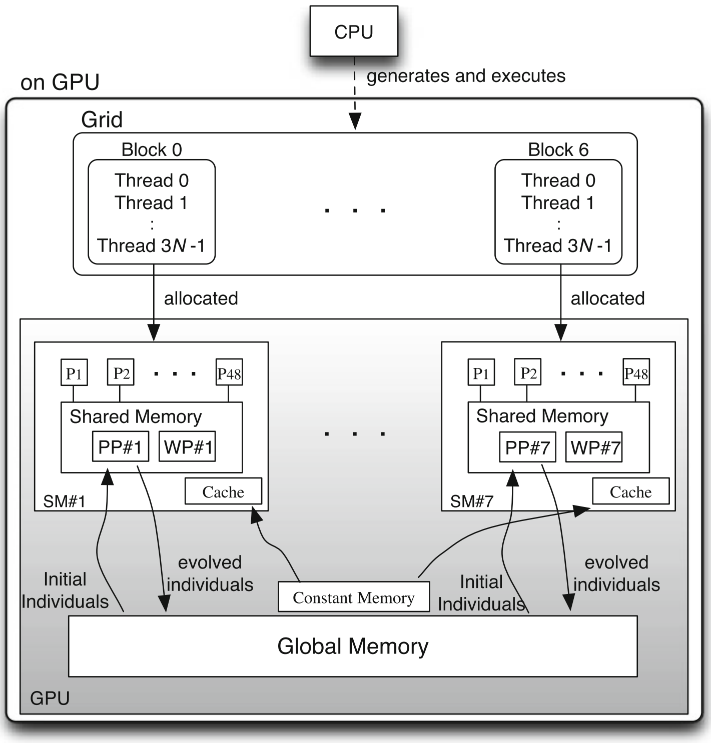


Fig. 9 Parallel GA model for GPU computation

SM. Here, PP is the population pool to keep individuals of the current population, and WP is a working pool to keep newly generated offspring individuals until they are selected to update PP for the next generation.

The procedure of the parallel GA model for GPU computation is as follows.

1. In the host machine, all individuals are randomly generated and then sent to the global memory of the GPU.
2. Each SM copies the corresponding individuals from global memory to its shared memory, and the generational process is repeated until the termination criteria are satisfied.
3. Finally, each SM copies the evolved individuals from its shared memory to global memory.

When applying the GA to the solution of Sudoku, the general procedure is to define an 81-bit one-dimensional chromosome that consists of the joined sub-chromosomes of the various puzzle regions and then perform crossover with the crossover points limited to only the joints between regions. Crossover of this type, however, is believed to easily destroy building blocks. As a one way to solve that problem, we define a two-dimensional chromosome, taking building-block linkage into account. Crossover is then performed by assigning a score to each row or column, each of which consists of region blocks, comparing the scores for the two parent individuals, and then passing the row or columns that have the highest scores on to the child.

Generally, local search functions are effective for constraint satisfaction problems such as Sudoku. Our objective, on the other hand, remains as the solution of Sudoku puzzles in a practical time within the framework of evolutionary computation. Accordingly, we added only a simple local search function in which multiple child candidates are generated when mutation occurs and the candidate that has the highest score is selected as the child. This is equivalent to the selection function in (μ, λ)-ES and is an operation in the evolutionary computation framework.

Implementation that Takes Measures Against the Initial Value Dependency Problem into Account

From Fig. 7, we can see that when the number of individuals is 150, the number of generations until the optimum solution is found depends on the initial values. If we do not consider parallelization, the processing time is considered to be determined by the product of the number of individuals and the number of generations. Accordingly, from the relation of the proportion of correct solutions obtained to the processing time, we set the number of individuals to 150 on the basis of preliminary experiments. We could conjecture that the effect of parallelization using the GPU would be that increasing the number of individuals would not greatly affect the processing time.

On the other hand, increasing the parallelism requires that the data on individuals and other data required for evolutionary computation be stored in the shared memory of the SM rather than in the global memory, but the shared-memory capacity is low and may not hold the data for a sufficient number of individuals. Furthermore, the data transfer speed between SM in the GPU is more than 100 times slower than the communication within SM, so an implementation that requires frequent communication between SMs is not suitable. Accordingly, we adopt an implementation method in which each SM runs the same evolutionary computation program, changing only the initial values of the individual data, etc., and whichever SM finds a solution terminates. In other words, the evolutionary computation programs running in the SMs using threads are executed in parallel, and the execution of the same program in each SM with different initial values is considered to serve as a measure against initial value dependency. This can be considered to be SIMD-type parallel processing.

Sparing Use of High-Speed Shared Memory

In previous experiments that involved the evaluation of benchmark tests implemented on multi-core processors [10], tables for random number generation were

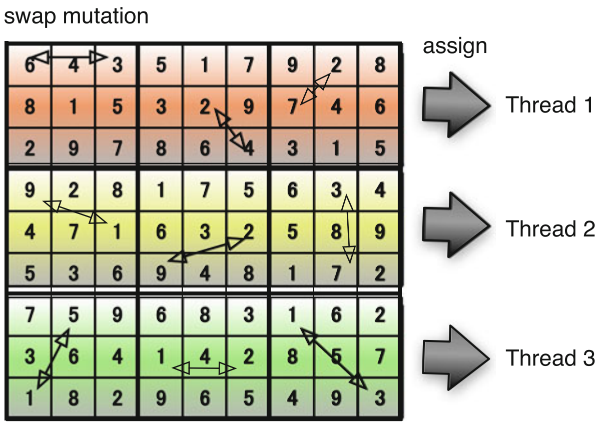


Fig. 10 An example of the swap mutation within a sub-block and the thread assignment

placed on each core to reduce the time required for random number generation. On the other hand, the shared memory of the GTX 460 is small, with a maximum of 48KB, so if random number tables are placed in each block, the required number of individuals cannot be defined. Therefore, the CURAND library function is used instead of the random number tables. Random number generation with CURAND is slower than using random number tables on each core, but it is much faster than generating the random numbers on the host and transferring the values to each SM. Furthermore, because the 64KB read-only constant memory can be accessed at high speed, the initial arrangement of the Sudoku puzzle (four bytes (int)$\times$81 $=$ 324 bytes) is stored in that cache memory. The data allocated to the shared memory is as follows:

- Area for storing individual data: 1 byte (char) $\times 81 \times N \times 2$
- Work area for tournament selection: 4 bytes (int) $\times N$
- Work area for crossover: 4 bytes (float) $\times N/2$
- Work area for mutation: 1 byte (char) $\times$ 81 $\times N$

Parallel Processing in Units of Sub-chromosome

Figure 10 shows an example of the swap mutation within a sub-block and the thread assignment. For a Sudoku puzzle that comprises 3$\times$3 region blocks, the genetic manipulation for each region block is performed in parallel, so nine threads are allocated to the processing for one individual. Because of the limited shared-memory capacity, however, we assign here three threads to the processing for one individual. The processing for crossover, mutation, or other such purpose for each line or column that consists of three region blocks is accelerated by the parallel processing of three threads.

4.2.2 Parallel GA Model and Implementation for Multi-core Processors

It can be seen from Fig. 7 that the number of generations needed to find an optimal solution for the same Sudoku problem is highly dispersed. This indicates that a Sudoku solution using evolutionary computations is dependent on the initial value when a sufficient number of individuals cannot be set due to insufficient memory capacity or other constraints. With this in mind, we generate the same number of threads as cores in the target processor and propose a method that executes genetic operations with each core having a different initial value. We also adopt the value of the core that finds a Sudoku solution first.

The procedure of the parallel GA model at each core processor is as follows:

1. All individuals are randomly generated.
2. The generational process is repeated until the termination criteria are satisfied.
3. The core that finds a Sudoku solution first cancels the operations in the other cores.

5 Evaluation Experiments

5.1 Execution Platform

The specifications of the execution platform used in experiments for Intel Core i7 and GTX 460 are listed in Tables 3 and 4, respectively. The specifications of the GTX 460 GPU used in these experiments are listed in Table 5.

5.2 Scalability

The system described here has scalability with respect to the number of cores. Increasing the number of cores in an SM is also considered to improve robustness against the dependence on initial values. For this reason, in the case of Core i7, we varied the number of threads to be executed in parallel from 1 to 8 and surveyed (1) solution rate, (2) average number of generations until the correct solution was obtained, and (3) average execution time. The number of cores in the multi-core processor is 4, but since 2 threads can be executed in parallel in one core by hyper-threading technology, we performed the experiment by executing a maximum of 8 threads in parallel. On the other hand, in the case of GPU, we varied the number of SM ranging from one to seven.

The results are presented in Tables 6, 7, and 8 for Core i7 and in Tables 9, 10, and 11 for GTX 460. The values shown in the results are the averages for 100 experiments that were conducted with 150 individuals and a cutoff of 100,000 generations.

Table 3 Multi-core processor execution environment

CPU	Intel Core i7 920 (2.67GHz, 4 cores)
OS	Ubuntu 10.04
C compiler	gcc 4.4.3 (optimization “O3”)

Table 4 GPU execution environment

CPU	Phenom II X4 945 (3GHz, 4 cores)
OS	Ubuntu 10.04
C compiler	gcc 4.4.3 (optimization “O3”)
CUDA Toolkit	3.2 RC

Table 5 GTX 460 specifications

Board	ELSA GLADIAC GTX 460
#Core	336 (7 SM×48 Core/SM)
Clock	675MHz
Memory	1GB (GDDR5 256 bits)
Shared memory/SM	48KB
#Register/SM	32,768
#Thread/SM	1,536

Table 6 The rate of correct answers, the number of average generations, and the average execution time (Intel Core i7: SD1)

#Threads	Count [%]	Average gen.	Exec. time
1	94	32,858	22s 19
2	100	15,268	13s 87
4	100	7,694	13s 11
8	100	3,527	7s 39

Table 7 The rate of correct answers, the number of average generations, and the average execution time (Intel Core i7: SD2)

#Threads	Count [%]	Average gen.	Exec. time
1	82	42,276	28s 41
2	98	25,580	22s 48
4	100	13,261	21s 47
8	100	5,992	12s 12

Table 8 The rate of correct answers, the number of average generations, and the average execution time (Intel Core i7: SD3)

#Threads	Count [%]	Average gen.	Exec. time
1	69	60,157	39s 88
2	93	46,999	40s 43
4	100	19,982	30s 79
8	100	8,795	17s 13

Table 9 The rate of correct answers, the number of average generations, and the average execution time (GTX 460: SD1)

#SM	Count [%]	Average gen.	CPU time
1	62	57,687	16s 728
2	80	40,820	11s 845
4	98	19,020	5s 527
8	100	10,014	2s 906

Table 10 The rate of correct answers, the number of average generations, and the average execution time (GTX 460: SD2)

#SM	Count [%]	Average gen.	CPU time
1	50	70,067	20s 199
2	69	58,786	16s 958
4	93	31,254	9s 260
8	97	22,142	6s 391

Table 11 The rate of correct answers, the number of average generations, and the average execution time (GTX 460: SD3)

#SM	Count [%]	Average gen.	CPU time
1	32	82,742	23s 958
2	59	68,050	19s 722
4	77	47,811	13s 879
8	95	30,107	8s 727

5.3 Experiments on Increasing the Number of Individuals

For a large number of individuals, initial values in a Sudoku solution are highly diverse. To investigate the relationship between diversity in initial values and the results of a Sudoku solution, we varied the number of individuals on Intel Core i7 from 150 to 400 and surveyed (1) solution rate, (2) average number of generations until the correct solution was obtained, (3) average execution time, and (4) minimum number of generations.

Table 12 The result on increasing the number of individuals (SD2)

#Individuals	Count [%]	Average gen.	Exec. time	Best gen.
100	100	8,641	11s 63	644
150	100	5,992	12s 12	243
200	100	7,115	19s 20	229
300	100	9,441	38s 29	123
400	98	15,441	84s 76	86

Table 13 The minimum numbers of generations and the execution times required to solving SD1 through SD3

Sudoku	Minimum gen.	Exec. time
SD1	83	25 ms
SD2	158	47 ms
SD3	198	76 ms

The result for SD2 is presented in Table 12. The values shown in the results are the averages for 100 experiments. In these experiments, we limited the number of threads to 4 and 8 and set the search termination point to 100,000 generations.

5.4 Minimum Number of Generations

To estimate the performance in the case that the initial value dependence problem has been solved, we determined the minimum numbers of generations and the execution times required to solve SD1 through SD3 (Table 13).

6 Discussion

6.1 Scalability and the Dependence on Initial Values

From Tables 6 through 8, we can see that increasing the number of threads reduces the execution time and increases the correct solution rate. Furthermore, we can see that the reduction rate of the average number of generations and average execution time with respect to execution by one thread decreases as the number of threads increases. In other words, the problem of initial value dependence tends to be eliminated as the number of threads is increased for both the processing time and the correct solution rate.

On the other hand, in the case of GPU, from Tables 9–11, we can see that increasing the number of SM reduces the execution time and increases the correct solution rate. In other words, the problem of initial value dependence tends to be eliminated as the number of SM is increased for both the processing time and the correct solution rate. From Table 13, increasing the number of SM or any other means of solving the initial value dependence problem makes it fully possible to solve super-difficult Sudoku puzzles within one second in a stable manner by parallelization of evolutionary computation using a GPU.

In other words, we consider that the performance of a multi-core processor is scalable in relation to number of threads and that the performance of a GPU is scalable in relation to the number of SMs. Parallelization using the GTX 460 GPU finds solutions faster than that using Core i7 multi-core processor, but we consider this to be due simply to a difference in number of cores. On the other hand, Core i7 exhibits higher solution rates, which we think are due to the fact that random numbers in GTX 460 are generated using the CURAND library function.

6.2 Setting the Number of Individuals

Generally, we can consider that the effect of parallelization will become large as the number of individuals increases. From Table 12, in case of the Core i7, increasing the total number of individuals increased the number of individuals that came closer to the correct Sudoku solution but also increased the number that deviated from the correct solution. This is considered to be the reason why the average number of generations until the correct solution was obtained also increased. Increasing the number of individuals also increased the processing time for one generation thereby increasing the average execution time until the correct solution was obtained. The value of best generations also decreased. Accordingly, if the number of cores in the processor can be increased and processing performance by parallelization increased in future multi-core processors, we can expect processing to accelerate to the point where it will be possible to derive correct solutions for even super-difficult Sudoku problems in less than a few seconds.

In the processing-acceleration technique by GPU that we have been developing in parallel with the above technique, programming must take into account upper limits such as task parallelization and memory capacity based on the hardware specifications of the GPU to be used. There is therefore a limit as to how far the number of individuals can be increased to solve the problem of initial value dependence. On the other hand, the technique introduced in this chapter, while inferior to the GPU technique in terms of parallelization, enables a parallel program to be executed without limitations in number of threads or memory capacity by virtue of using general-purpose multi-core processors. Looking forward, we believe that accelerating evolutionary computations for solving Sudoku puzzles by a highly parallel system should be effective for either GPUs or multi-core processors while solving the problem of initial value dependence by increasing the number of individuals.

Table 14 The execution time and the correct solution rates for when the number of individuals is set to 192

Sudoku	Count [%]	Average gen.	CPU time
SD1	100	9,072	2s 751
SD2	100	13,481	4s 530
SD3	100	22,799	6s 862

On the other hand, in the case of GTX 460, the amount of data allocated to the shared memory as described in Sect. 4.2.1 is equal to 249N, so the maximum number of individuals for which data can be stored in the 48 KB shared memory is 192. Furthermore, considering that 192 is a multiple of the number processors within the SM and also a multiple of the CUDA thread processing unit, the execution time and the correct solution rates for when the number of individuals is set to 192 are presented in Table 14.

Compared with the case in which the number of individuals is set to 150, the processing time was reduced by approximately 5 % while the correct solution rate remained at 100 %. For SD2, the correct solution rate increased from 97 % to 100 % and the processing time decreased by approximately 29 %. For SD3, the correct solution rate increased from 90 % to 100 % and the processing time decreased by approximately 21 %. From this data, we can see that setting the number of individuals to an appropriate value for parallel execution of an evolutionary computation program written in C on a GPU accelerates the processing relative to processing on a CPU by a factor of 25.5 for the SD1 problem, by a factor of 19.1 for SD2, and by a factor of 16.2 for SD3. We can also see that a correct solution rate of 100 % can be attained, even for problems in all three of the super-hard categories.

It can therefore be seen that super-difficult Sudoku problems can be solved in realistic times by the parallelization of evolutionary computation using the Core i7 multi-core processor commonly used in desktop PCs or the inexpensive, commercially available GTX 460 GPU.

These experiments also show that the GPU can find solutions faster than the multi-core processor by making use of a higher degree of parallelization. As new GPUs with a higher number of SMs and a higher degree of integration come to be developed, we can expect even faster execution times. At the same time, the GPU suffers from limitations such as the need for programming that must consider upper limits in task parallelization and in the memory allocated to each task due to hardware constraints. In other words, it is more difficult to use a GPU than a multi-core processor which can execute programs in parallel without having to worry about limitations in number of threads or memory capacity. Furthermore, when using a library function such as CURAND for random number generation due to limitations in shared-memory capacity within an SM, problems with the cycle length of random numbers generated in this way must be taken into account.

Thus, when trying to decide whether to use a GPU or a multi-core processor when attempting to accelerate evolutionary computations by parallelization, due consideration must be given to the target application problem.

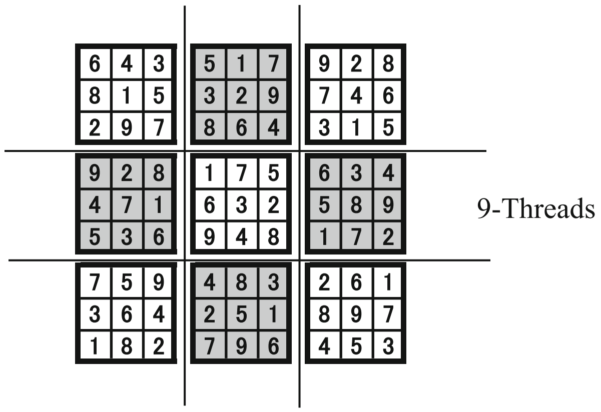


Fig. 11 Relationship between mutation and thread allocation

6.3 Assessment of Fine-Grained Parallelization on a Sub-chromosome Unit

The effect of acceleration on the proposed crossover and mutation methods using fine-grained parallelization of a sub-chromosome unit was assessed. Since assessment is made on an individual without parallel processing, there is allowance in the number of utilizable threads. Figure 11 shows the relationship between mutation and thread allocation.

As shown in Fig. 11, in order to increase parallel processing speed, nine threads were allocated to the processing of one individual. Two numbers were randomly chosen within a region, and swapping of the two within the nine region blocks was processed in parallel.

Figure 12 shows the relationship between crossovers and thread allocation. Regarding crossovers, to prevent the destruction of the valid part of the solution (building blocks), three rows or three columns consisting of region blocks were compared and the higher scores were passed onto the offspring. Since each row and column independently undergoes this process three times in parallel, three threads can be allocated to each individual and thus parallel processing is achieved.

The parameters for the genetic algorithm were the same as those of the previous experiment, with the level of difficulty set at Super Difficult 3. The machine specifications used for the experiment were shown in Table 15.

Table 16 shows the fine-grained parallel processing results. Processing time equals the average generation number required to obtain the answer divided by execution time (sec). In addition, the degree of the increase in processing performance is a value relative to execution without parallel processing of individuals on a CPU.

Table 16 shows that fine-grained parallel processing of the proposed crossover method increased performance by 23 %, while fine-grained parallelization of mutations without parallel processing of individuals on a GPU increased process

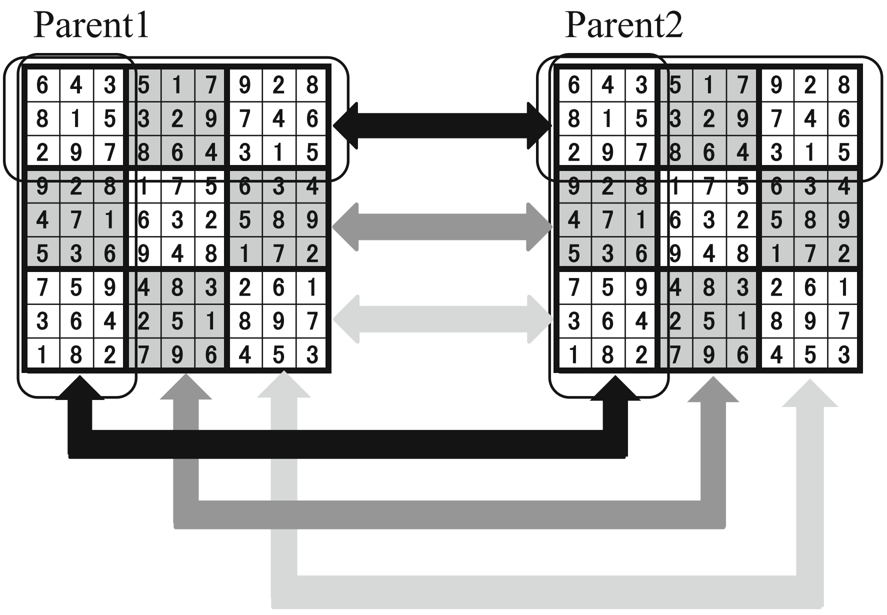


Fig. 12 Relationship between crossovers and thread allocation

Table 15 GTX 580 GPU execution environment (2)

OS	Ubuntu 10.04
CUDA Toolkit	4.0
GPU	GeForce GTX 580
CPU	Phenom Ⅱ X4 945 (3GHz, 4 cores)
Main memory	DDR2-800 4GB

Table 16 Comparison of processing time with fine-grained parallel processing

	Processing time [sec]	Processing performance (generation/sec)	Increase in processing performance
No parallel processing of individuals (CPU)	42.600	1,282	1
No parallel processing of individuals (GPU)	17.110	3,766	2.93
Parallel processing of crossovers (GPU)	14.313	4,054	3.16
Parallel processing of mutations and crossovers (GPU)	10.730	5,018	3.91

performance by 75 % when compared to execution without parallel processing of individuals on a GPU. In addition, process performance increased by approximately four times with parallel processing of proposed mutation and crossover methods on a GPU compared to execution without parallel processing on a CPU. The performance increase is lower compared to parallel processing on an individual level because the overhead required for thread generation is large. Further, the increase in process performance is larger with the mutation method compared to the crossover method because the number of threads, that is, the degree of parallel processing, is set to nine and the number of processes for mutations is large. Performance increased by approximately three times for processing on a GPU compared to processing on CPU because a number of the same processes with different initial values are executed by 16 streaming multiprocessors simultaneously; hence, there is less of a problem of dependence on initial values.

Our results show a modest increase in performance with fine-grained parallel processing of genetic operations such as crossovers and mutations on a sub-chromosome unit compared to parallel processing on an individual level. On the other hand, in an ideal environment, two parallel processes that are run simultaneously may be compounded to produce even faster processing. In our experiment using GTX 460, each performance increase was not compounded. The reason is the problem of computing system resources which are limited by the executable number of threads in each block. Improvement of the GPU endeavors to eliminate the lack of system resources, which will allow the execution of the two parallel processors to be multiplied, resulting in much faster processing.

7 Conclusion

We have used the problem of solving Sudoku puzzles as an actual and practical problem to demonstrate that practical processing time is possible through the use of many-core processors for parallel processing of evolutionary computation. Specifically, we implemented parallel evolutionary computation on the NVIDIA GTX 460, a commercially available GPU, or the commercially available Core i7 multi-core processor from Intel. Evaluation results showed that execution acceleration factors from 10 to 25 relative to execution of a C program on a CPU are attained and a correct solution rate of 100 % can be achieved, even for super-difficult problems. In short, we showed that the parallelization of evolutionary computation using a multi-core processor commonly used in desktop PCs or a GPU that can be purchased at low cost can be used to solve problems in realistic times, even in the case of problems for which the application of genetic algorithms has not been studied in the past because of excessive processing times.

Furthermore, fine-grained parallelization of genetic operations that take linkage into account accelerated processing by a factor of 4 relative to processing on a CPU. An increase in GPU resources will diminish the conflict of thread usage between coarse-grained parallelization on an individual level and will enable a faster processing speed.

References

1. Gordon, V.S., Whitley, D.: Serial and parallel genetic algorithms as function optimizers. In: Proceedings of the 5th International Conference on Genetic Algorithms, pp. 177–183. Morgan Kaufmann Publishers Inc., San Franciso, CA USA (1993)
2. Mühlenbein, H.: Parallel genetic algorithms, population genetics and combinatorial optimization. In: Proceedings of the 3rd International Conference on Genetic Algorithms, pp. 416–421 (1989)
3. Mühlenbein, H.: Evolution in time and space - the parallel genetic algorithm. In: Foundations of Genetic Algorithms, pp. 316–337. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1991)
4. Shonkwiler, R.: Parallel genetic algorithm. In: Proceedings of the 5th International Conference on Genetic Algorithms, pp. 199–205 (1993)
5. Cantu-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers, Norwell, MA,USA (2000)
6. Byun, J.H., Datta, K., Ravindran, A., Mukherjee, A., Joshi, B.: Performance analysis of coarse-grained parallel genetic algorithms on the multi-core sun UltraSPARC T1. In: SOUTHEASTCON'09, pp. 301–306. IEEE, Danvers, MA, USA (2009)
7. Serrano, R., Tapia, J., Montiel, O., Sepúlveda, R., Melin, P.: High performance parallel programming of a GA using multi-core technology. In: Castillo, O., Melin, P., Kacprzyk, J., Pedrycz, W., eds.: Soft Computing for Hybrid Intelligent Systems. Volume 154 of Studies in Computational Intelligence, pp. 307–314. Springer, Berlin/Heidelberg (2008)
8. Tsutsui, S., Fujimoto, N.: Solving quadratic assignment problems by genetic algorithms with GPU computation: a case study. In: GECCO '09: Proc. 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference, pp. 2523–2530 (2009)
9. Munawar, A., Wahib, M., Munetomo, M., Akama, K.: Theoretical and empirical analysis of a GPU based parallel Bayesian optimization algorithm. In: Proceedings of the 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies. PDCAT '09, pp. 457–462 (2009)
10. Sato, M., Sato, Y., Namiki, M.: Proposal of a multi-core processor from the viewpoint of evolutionary computation. In: Proceedings of the IEEE Congress on Evolutionary Computation 2010, pp. 3868–3875 (July 2010)
11. Wikipedia: Sudoku. Available via WWW: http://en.wikipedia.org/wiki/Sudoku (cited 8.3.2010)
12. Wikipedia: Backtracking. Available via WWW: http://en.wikipedia.org/wiki/Backtracking (cited 1.11.2011)
13. IEEE: ISO/IEC 9945-1 ANSI/IEEE Std 1003.1. (1996)
14. Sato, Y., Inoue, H.: Solving Sudoku with genetic operations that preserve building blocks. In: Proceedings of the IEEE Conference on Computational Intelligence in Game, pp. 23–29 (2010)
15. Lewis, R.: Metaheuristics can solve Sudoku puzzles. J. Heuristics **13** 387–401 (2007)
16. Simonis, H.: Sudoku as a constraint problem. In: Proc. of the 4th Int. Workshop Modelling and Reformulating Constraint Satisfaction Problems International Conference on Genetic Algorithms, pp. 13–27 (2005)
17. Lynce, I., Ouaknine, J.: Sudoku as a SAT problem. In: Proceedings of the 9 th International Symposium on Artificial Intelligence and Mathematics, AIMATH 2006 (2006)
18. Moon, T., Gunther, J.: Multiple constraint satisfaction by belief propagation: An example using Sudoku. In: Proceedings of the 2006 IEEE Mountain Workshop on Adaptive and Learning Systems (SMCals 2006), pp. 122–126. IEEE, Los Alamitos, CA, USA (2006)
19. Mantere, T., Koljonen, J.: Solving and ranking Sudoku puzzles with genetic algorithms. In: Proceedings of the 12th Finnish Artificial Conference STeP 2006, pp. 86–92 (October 2006)
20. Mantere, T., Koljnen, J.: Solving, rating and generating Sudoku puzzles with GA. In: Proceedings of the IEEE Congress on Evolutionary Computation 2007, pp. 1382–1389 (July 2007)

21. Moraglio, A., Togelius, J., Lucas, S.: Product geometric crossover for the Sudoku puzzle. In: Proceedings of IEEE Congress on Evolutionary Computation 2006, pp. 470–476 (July 2006)
22. Galvan-Lopez, E., O'Neill, M.: On the effects of locality in a permutation problem: The Sudoku puzzle. In: Proceedings of IEEE Symposium on Computational Intelligence and Games (CIG 2009), pp. 80–87 (September 2009)
23. Goldberg, D.E., Sastry, K.: A practical schema theorem for genetic algorithm design and tuning. In: Proceedings of the 2001 Genetic and Evolutionary Computation Conference, pp. 328–335 (2001)
24. Super Difficult Sudoku's. Available via WWW: http://lipas.uwasa.fi/~timan/sudoku/EA_ht_2008.\pdf#search='CT20A6300%20Alternative%20Project%20work%202008' (cited 8.3.2010)

Index