J Supercomput (2013) 63:773–789
DOI 10.1007/s11227-012-0770-1

Enhancing GPU parallelism in nature-inspired algorithms

José M. Cecilia · Andy Nisbet · Martyn Amos · José M. García · Manuel Ujaldón

Published online: 3 May 2012

Abstract We present GPU implementations of two different nature-inspired optimization methods for well-known optimization problems. Ant Colony Optimization (ACO) is a two-stage population-based method modelled on the foraging behaviour of ants, while P systems provide a high-level computational modelling framework that combines the structure and dynamic aspects of biological systems (in particular, their parallel and non-deterministic nature). Our methods focus on exploiting data parallelism and memory hierarchy to obtain GPU factor gains surpassing 20x for any of the two stages of the ACO algorithm, and 16x for P systems when compared to sequential versions running on a single-threaded high-end CPU. Additionally, we

J.M. Cecilia (✉)
Departamento de Informática, Escuela politécnica, Universidad Católica San Antonio Murcia, Campus de los Jerónimos S/N, Guadalupe 30107, Murcia, Spain
e-mail: jmcecilia@ucam.edu

A. Nisbet · M. Amos
School of Computing, Mathematics and Digital Technology, Manchester Metropolitan University, Chester Street, Manchester M1 5GD, UK

A. Nisbet
e-mail: a.nisbet@mmu.ac.uk

M. Amos
e-mail: m.amos@mmu.ac.uk

J.M. García
Facultad de Informática, Universidad de Murcia, Campus de Espinardo, 30080 Murcia, Spain
e-mail: jmgarcia@ditec.um.es

M. Ujaldón
Computer Architecture Department, ETSI Informática, University of Málaga, Campus Teatinos, Bulevar Louis Pasteur, s/n, 29071 Málaga, Spain
e-mail: ujaldon@uma.es

compare performance between GPU generations to validate hardware enhancements introduced by Nvidia's Fermi architecture.

Keywords GPUs · HPC · ACO · P systems · Bioinspired methods

1 Introduction

The use of *Graphics Processing Units* (GPUs) has become increasingly popular in high performance computing applications [13], to the extent that GPUs are now a key resource for parallel applications [12]. The scalable CUDA programming model (for details, see [19]) has evolved to support the new hardware features progressively added by NVIDIA to their range of GPUs, with the *Fermi* architecture [21] being the most recent milestone in this path. In this article, we evaluate the GPU performance optimisation of two commonly-used bio-inspired computational methods; Ant Colony Optimisation (ACO) [10] and Membrane Systems (or P systems) [22, 23]. We begin by giving a brief overview of these methods, before describing how each may be tuned for the GPU. We then describe the results of performance evaluation experiments, and conclude with a brief discussion.

ACO is a population-based search method inspired by the foraging behaviour of ants. It has been applied to a wide range of problems [1, 6], many of which are graph-theoretic in nature. It was first applied to the Traveling Salesman Problem (TSP) [17] by Dorigo et al. [8, 9]. Membrane Computing (of which P systems are a central part) is a paradigm inspired by living cells, introduced by Paŭn [23]. Membrane computing models the *biochemical processes* taking place inside living cells. Simulated cells have several syntactic ingredients: a membrane structure, consisting of a hierarchical arrangement of membranes embedded in a skin, and delimiting regions (or *compartments*) where multi-sets of objects and sets of evolution rules are placed. A number of different models of P systems have been developed, and many of them are computationally universal [4].

A common computational feature shared by both ACO and P systems, is their inherent *massive parallelism*. ACO algorithms are population-based, that is, a collection of agents "collaborate" to find an optimal (or at least a satisfactory) solution. In a P system, thousands of "cells" run in parallel to search for a solution. Both approaches raise new computational challenges, depending on the nature of the particular problem and the underlying features of a parallel target architecture. In the following sections, we analyze and optimize the new computational patterns emerging from these methods on two different GPUs; an ACO algorithm for the Traveling Salesman Problem (TSP), and a simulation of a recognizer P system with active membranes for the Satisfiability (SAT) problem.

2 Tuning the ACO algorithm for solving TSP on GPUs

The Travelling Salesman Problem (TSP) [17] involves finding the shortest (or "cheapest") round-trip route that visits each "city" exactly once. The symmetric TSP on n

cities is represented as a complete weighted graph, G, of n nodes, with each weighted edge $e_{i,j}$ representing the inter-city distance $d_{i,j} = d_{j,i}$ between cities i and j. TSP is a well-known NP-hard optimisation problem, and is used as a standard benchmark for many heuristic algorithms [15].

The ACO TSP solution in [5, 9] uses a number of simulated "ants" (or *agents*) to perform distributed search on a graph. Each ant moves through the graph until it completes a tour, and then offers this tour as its suggested solution. Each ant drops "pheromone" on the edges that it visits, and the amount of pheromone dropped, if any, is determined by the *quality* of an ant's solution relative to those obtained by other ants. Each ant probabilistically chooses the next city to visit based on *heuristic information* obtained from inter-city distances and the net pheromone trail. Although such heuristic information drives ants towards an optimal solution, a process of "evaporation" is also applied in order to prevent the process stalling in a local minimum.

This computation is divided into two main stages: *Tour construction* and *Pheromone update* [10]. In essence, simulated ants construct solutions to the TSP in the form of *tours*. Pheromone trails are a fundamental component of the algorithm, since they facilitate indirect communication between agents via their *environment*, a process known as *stigmergy* [7]. For additional details about these processes, please see [10].

In this section, we discuss several GPU implementations of the Ant System (an early variant of ACO, first proposed by Dorigo [5]) as applied to the TSP. We propose different computational patterns, introduce two different design approaches for the tour construction stage (based on either task or data parallelism), and describe several GPU techniques for increasing data bandwidth during the pheromone update stage.

2.1 The sequential baseline

Algorithm 1 The sequential AS version for the TSP problem

```
1: InitializeData()
2: while ¬Convergence() do
3:     TourConstruction()
4:     PheromoneUpdate()
5: end while
```

Algorithm 1 shows the sequential pseudo-code of the Ant System. The algorithm first initializes all data structures, and then proceeds with the two main stages: *Tour construction* and *Pheromone update*, until the convergence criteria are met.

The *Tour construction stage* is divided into two stages: *Initialization* and *ASDecisionRule*. In the former, all data structures are initialized by each ant, and ants are randomly assigned to a city. Algorithm 2 shows the latter stage, which is divided into two sub-stages. First, each ant calculates heuristic information, in order to inform the decision on whether or not to visit city j from city i. As explained in [10], it is computationally expensive to repeatedly calculate those values for each computational step of each ant, k, and an additional data structure, namely *choice_info*, is used to store heuristic values through an adjacency matrix [10] in order to avoid unnecessary computation. Notice that each entry of this structure can be calculated independently. Secondly, the probabilistic choice of the next city to visit by each ant is performed by using roulette wheel selection [10, 14] (see Algorithm 2).

Algorithm 2 *ASDecisionRule* for the Tour construct stage. m is the number of ants, and n is the number of cities of the TSP instance

```
1: sum_prob ← 0.0;
2: curr_city ← ant[k].tour[step − 1];
3: for j = 1 to n do
4:     if ant[k].visited[j] then
5:         selection_prob[j] ← 0.0;
6:     else
7:         curr_prob ← choice_info[curr_city][j];
8:         selection_prob[j] ← curr_prob;
9:         sum_probs ← sum_probs + curr_prob;
10:    end if
11: end for
    {Roulette Wheel Selection Process}
12: r ← random(1..sum_probs);
13: j ← 1;
14: p ← selection_prob[j];
15: while p < r do
16:     j ← j + 1;
17:     p ← p + selection_prob[j];
18: end while
19: ant[k].tour[step] ← j;
20: ant[k].visited[j] ← true;
```

Finally the pheromone update stage involves two main tasks: *evaporation* and *deposition*. Evaporation lowers the pheromone value on all edges by a constant factor. In deposition, each ant adds a quantity of pheromone on the edges that it has crossed in its tour, the amount being based on the relative quality of the tour.

2.2 Tour construction on GPUs

The "traditional" task parallelism approach to tour construction is based on the observation that ants run in parallel looking for the best tour [11, 18, 29]. Therefore, any inherent parallelism exists at the level of individual ants. In CUDA, each ant can be identified as a thread, and threads are equally distributed amongst thread blocks. To improve the kernel memory bandwidth, some data structures are placed in on-chip shared memory. The *visited* and *selection_prob* list are the best candidates, as they are accessed many times with an irregular access pattern. Shared memory is a limited resource in CUDA [19], which is allocated and shared at CUDA thread block level. Ants assigned to the same thread block directly share memory.

2.2.1 Strategies to increase data parallelism

The previous task parallelism approach offers several challenges for the GPU. Firstly, it requires a relatively low number of threads on the GPU (roughly the number of cities in the TSP instance [10]). Secondly, it presents unpredictable memory access patterns, due to its execution being guided by a stochastic process. Finally, checking the list of cities visited contains many warp divergences (threads taking different control-flow paths), leading to serialisation [19].

To increase parallelism, we calculate the *choice_info* data structure as a *separate kernel*, which is executed prior to tour construction. Here, a CUDA thread is assigned to compute each entry in the *choice_info* structure, and threads are evenly grouped into CUDA thread blocks. We also increase *data-parallelism*, and avoid warp divergence in the tour construction kernel, by associating a *thread-block* to each ant, such that each thread within a block represents a city (or cities) that those ants may visit. Now, all CUDA threads fully cooperate to obtain a solution, increasing the

data-parallelism by a factor of *1:w*, where w is the number of CUDA threads per thread-block.

A thread loads the heuristic value linked with its associated city (or cities), and checks if the city has been visited. To avoid conditional statements (and, thus, warp divergences), the tabu list is represented in shared memory as one integer value per city. A city's value is 0 if it has been visited, and 1 otherwise. Finally, these values are multiplied and stored in a shared memory array, which is then prepared for roulette wheel selection. We note that the shared memory requirements are drastically reduced, as the tabu and probabilistic lists are now stored once per thread-block, instead of once per thread.

The number of threads per thread-block on CUDA is a hardware limiting factor. Thus, the cities should be distributed among threads to allow for a flexible implementation. A *tiling* technique is proposed to deal with this issue. Cities are divided into blocks (i.e. tiles). For each tile, a city is selected stochastically from the set of unvisited cities on that tile. When this process is over, we have a set of "partial best" cities. Finally, the city with the best absolute heuristic value is selected from this partial best set.

The tabu list can be placed in the register file (as it now represents thread-private information). However, the tabu list cannot be represented by a single integer register per thread in the tiled version, because a thread now represents more than one city. The 32-bit registers may be used on a bitwise basis for managing the list. For example, the first city represented by each thread, i.e. on the first tile, is managed by bit 0 on the register that represents the tabu list, and subsequent cities are represented by subsequent bits. A roulette wheel algorithm is implemented in ACO to provide a fully sequential stochastic selection process [10]. In CUDA, a particular thread is identified to proceed sequentially with the selection $n - 1$ times, where n equals the number of cities. An alternative approach consists of generating a random number for each city in the interval [0, 1] to feed into the stochastic simulation. Thus, three values are multiplied and stored in the shared memory array per city, i.e. the heuristic value associated with a city, a value showing whether the city has been visited or not, and the random number associated with a city.

2.3 Pheromone update stage: evaporation and deposition

Evaporation is quite straightforward to implement, as a single thread can independently lower each entry of the pheromone matrix by a constant factor. Deposition is more problematic, as each ant generates its own private tour in parallel, and will eventually visit the same edge as another ant. Consequently, atomic operations are required (with an associated negative impact on performance) to prevent race conditions when accessing the pheromone matrix. An alternative approach can use *scatter gather* transformations [27] in order to avoid expensive atomic operations. The scatter to gather pattern is implemented by creating as many threads as there are cells in the pheromone matrix, and then distributing those threads evenly amongst thread blocks. As each thread represents the coordinates of a single entry in the pheromone matrix, it is assigned the task of checking whether the cell represented by a pheromone matrix element has been visited by any ant. At this point, we have a tradeoff between the

pressure in device memory for avoiding a design based on atomic operations, and the number of atomic operations involved. Therefore, this approach allows us to perform computation *removing* atomic operations, but at the expense of drastically increasing the pressure on device memory.

A tiling technique is proposed to increase application bandwidth. All threads cooperate to load data from global memory to shared memory, but they still access edges in an ant's tour. The number of memory accesses is reduced, although it remains a similar order of magnitude. An ant's tour length $(n + 1)$ may, however, be larger than the maximum number of threads that each block can support. Our algorithm prevents this situation by setting our empirically demonstrated optimum thread block layout (θ), and dividing the tour into tiles of this length. This raises another issue when $n + 1$ is not divisible by θ. We solve this by applying padding to the ants tour array, in order to avoid warp divergence. Unnecessary loads to device memory can be avoided by taking advantage of the symmetric version of the TSP; this halves both the number of threads and the number of device memory accesses.

3 Tuning P systems for solving SAT on GPUs

The Satisfiability `SAT` problem determines if, for a given Boolean formula in Conjunctive Normal Form (CNF), there exists an assignment to its variables such that it evaluates to *true*. SAT problems occur in areas such as model checking, automatic test pattern generation for VLSI, theorem proving and software verification. Recognizer P systems with active membranes are one framework for solving such decision problems. In [24], a family of recognizer (*deterministic*) P systems with active membranes for solving SAT in linear time (but requiring exponential space) is described. From an algorithmic point of view, the simulation of a given P system may be enhanced by removing general model constraints concerning communication and global synchronization, which may be not required for a particular design. Moreover, P-system computation can be adapted to better exploit the underlying architecture. Here, we show how to enhance P system simulation for solving SAT on GPUs. For details of the implementation on other parallel architectures, we refer the reader to [3].

3.1 The sequential baseline

The P system simulation to solve the SAT problem is based on the P system computation described in [24], which can be summarized thus: (1) *Generation*. Membranes are structured within a rooted tree with a single branch. The root node is the *skin membrane*, and the second node is the *internal membrane*. All possible truth assignments to variables are generated using division rules encoded in internal membranes that are executed step by step, as described in [24]. In this way, 2^n internal membranes are created, where each one encodes a truth assignment to n CNF formula variables. (2) *Synchronization*. The objects encoding a true clause (a partial solution to the CNF formula) are unified in the membrane. (3) *Check out*. This step determines how many (and which) clauses are *true* in every internal membrane (that is, by the assignment that they represent). (4) *Output*. Internal membranes encoding a solution

send an object to the skin. If the skin has such an object from some membrane, then the object *Yes* is sent to the environment. Otherwise, the object *No* is sent.

The *Generation* and *Synchronization* stages create an exponential workspace of membranes in a synchronous way, and unify objects that codify a partial solution. Note that each membrane runs in parallel at each iteration of *Generation*, but a global synchronization is required by different iterations. Once the workspace is created, the *Check out* and *Output* stages are performed. They first determine which clauses are true in every internal membrane, and then check whether there exists a solution for the SAT problem.

3.2 A generic simulator

We use the generic simulator of recognizer P systems with active membranes from [2] as a starting point. The tool is applicable to a wide range of recognizer P systems, but its generality can lead to performance problems. In this simulator, the P system computation is only known at run-time once rules have been selected. Due to non-determinism, any rule may be selected and any object in the alphabet O may be present in a membrane at a given time. Clearly, this represents a worst case scenario, where all the objects of the alphabet O belong to the multi-set of a membrane, forcing the simulator to pre-allocate in GPU memory *all* potentially required resources (the GPU lacks true dynamic memory allocation). However, two levels of parallelism exist. One level resides in membranes, and is similar to that of CUDA blocks, and the other exists with the objects within each membrane, comparable to CUDA threads. The generality provided by this simulator penalizes performance significantly, and achieves low memory usage. At a given time, many threads are idle, because they deal with non-existent objects in each membrane. Furthermore, a considerable part of memory is allocated without being used at all, in order to cover the worst case scenario explained earlier.

3.3 Baseline simulator: adapting the simulator for the SAT problem

General conditions are removed from our baseline P-system simulator in order to reduce memory requirements. In this new version, the P system computation for any given SAT problem is still fully reproduced by the simulation algorithm. The new simulator is still decomposed into the same four main stages as described in Sect. 3.1. We use two CUDA kernels and one CPU stage; (1)a kernel that generates membranes and evolves objects within each membrane; (2), the Synchronization and *Check Out* stages are deployed in a single kernel; this determines the membranes that codify a solution (where all the clauses are true) for an instance of an SAT problem; (3) the *Output* stage runs on the CPU for an entire replication of P system behaviour.

To guarantee parallelism amongst membranes, a CUDA thread block is assigned to each membrane, as in [2]. However, a thread is now assigned to an object of the input multi-set, instead of each object in the alphabet O. Each object represents a literal of the CNF formula of the input SAT instance. Thus, a thread is created if and only if there is a literal of the CNF formula associated with it. The number of objects considered by each thread block is therefore widely reduced. However, parallelism is limited by global synchronization amongst thread blocks which update objects controlling the P system's Π_φ computation.

3.4 Enhanced simulator: increasing parallelism

Parallelism is increased and all computation is performed on the GPU by using two kernels. New approaches are taken to the computation of the *Check out* and *Output* stages, and heuristics are used to allow theoretical computation to be adapted to the simulator, while preserving the nature of the P system Π_φ, i.e. without loss of generality.

Algorithm 3 shows the Single Program Multiple Data (SPMD) pseudo-code for the Enhanced simulator. The CUDA kernels formerly belonging to the *Generation* stage are grouped into a single kernel, dividing membranes, and evolving objects concurrently. Furthermore, the evolution of control objects (R and D) [24] is replaced by a fixed number of iterations in the loop, as the main goal of these objects is to control the timing of execution in the theoretical P-system computation (which is known in advance).

Algorithm 3 SPMD pseudocode of the P system simulation algorithm for the SAT problem with n variables

Require: $n \geq 0$
{Start Generation and Synchronization stages}
repeat
Generation
until n
{Start Check out and Output stages}
Checkout

The *Generation* and *Synchronization* stages synchronously create an exponential workspace of membranes, that unifies objects codifying a partial solution within each membrane. The unification of objects is performed by each thread independently from each other after generation. Thus, there are no global synchronization requirements between both stages, and they are merged into a single *Generation* stage. Note that (1) each membrane is always generated by the *same* membrane in the same computational step, and (2) each membrane runs in *parallel* at each step, but a global synchronization is required *between* steps. The computation fits into a pattern of *dynamic memory generation*. Due to the static nature of the device memory allocation in CUDA, we solve this by creating a static queue, and setting kernel parameters at each iteration.

Once membranes have been created, the *Check out* and *Output* stages are performed (see Algorithm 3). First, they determine the "true" clauses in every internal membrane, and then check if a solution exists. We refer to these two stages as the *Check out* stage. In this kernel, each thread block loads a membrane from global memory, and then each thread checks rules associated with this stage. Finally, each block returns whether its associated membrane makes the CNF formula true/false. This kernel also includes the *Output* stage.

Initially, the simulation starts with a single CUDA thread block (representing the membrane received as input) that barely exploits GPU resources. However, the number of CUDA thread blocks grows exponentially in the *Generation* stage along with

the number of membranes, and GPU resources are fully utilized at relatively early stages of the simulation. Although it is possible to create a larger set of initial membranes on the CPU to avoid this problem, we have verified that this initial low usage of GPU resources has a negligible impact on performance, even on tiny benchmarks. The simulator does not take advantage of the data locality. First of all, a dynamic data set is generated and stored in global memory, due to the synchronization issues previously highlighted. Then the *Check out* stage again loads the data from global memory, incurring a high number of loads/writes from/to device memory.

3.5 Tiled simulator: exploiting data locality

Tiling is a technique for improving data locality in memory hierarchies [16]. The tiled simulator augments the *Generation* kernel to include a *Block Preprocessing* (BP) step, where a set of membranes are partially created and placed at block size intervals (see Fig. 1) in order to improve memory locality.

The second kernel performs the rest of the *Generation* locally at each block, followed by the *Check out* stage. Each thread within a block cooperates for an efficient load from *global* memory to *shared* memory of the initial membrane generated by the BP step (represented by a black square in Fig. 1). The *Generation* stage then interacts with shared memory, saving expensive loads/writes from/to global memory, which are around 400 times slower. Finally, the *Check out* stage is performed over the data stored in shared memory after a block-level synchronization. This checks whether a clause makes the CNF formula true, and writes its result into device memory.

4 Experimental setup

The performance of both algorithms was evaluated on two different NVidia GPU Tesla architectures: C1060 and C2050 (GT200 and Fermi architectures respectively [21]). Both GPUs are connected to the same motherboard with a dual-socket 2.40 GHz quad-core Intel Xeon E5620 for single-threaded sequential experiments. We use gcc 4.3.4 with the -O3 flag to compile our CPU implementations, and CUDA compilation tools release 3.2 for GPUs.

For TSP, we use a set of benchmark instances from the well-known TSPLIB library [26, 28]. We compare our implementations to the sequential code, written in ANSI C, provided by Stüzle in [10]. Performance figures are given for single-precision (float) numbers and a single iteration run averaged over 100 iterations. We set the ACO parameters according to the values recommended in [10]; $\alpha = 1$, $\beta = 2$, and $\rho = 0.5$, and $m = n$, which means the number of ants, m, is equal to the number of cities, n.

For SAT, a set of benchmarks generated by the WinSAT [25] program are used. WinSAT generates random SAT problems in DIMACS CNF file format by configuring several parameters: the number of variables (n), the number of clauses (m), and the number of literals per clause (k). The number of membranes in our P system depends on the number of CNF variables, n ($Membranes = 2^n$). We vary this parameter from $n = 13$ up to $n = 25$ whenever possible, whereas the number of literals ($l = m \times k$) is kept constant ($l = 256$ when $n < 22$ and $l = 200$ when $n \geq 22$) in order to remain within GPU device memory limits.

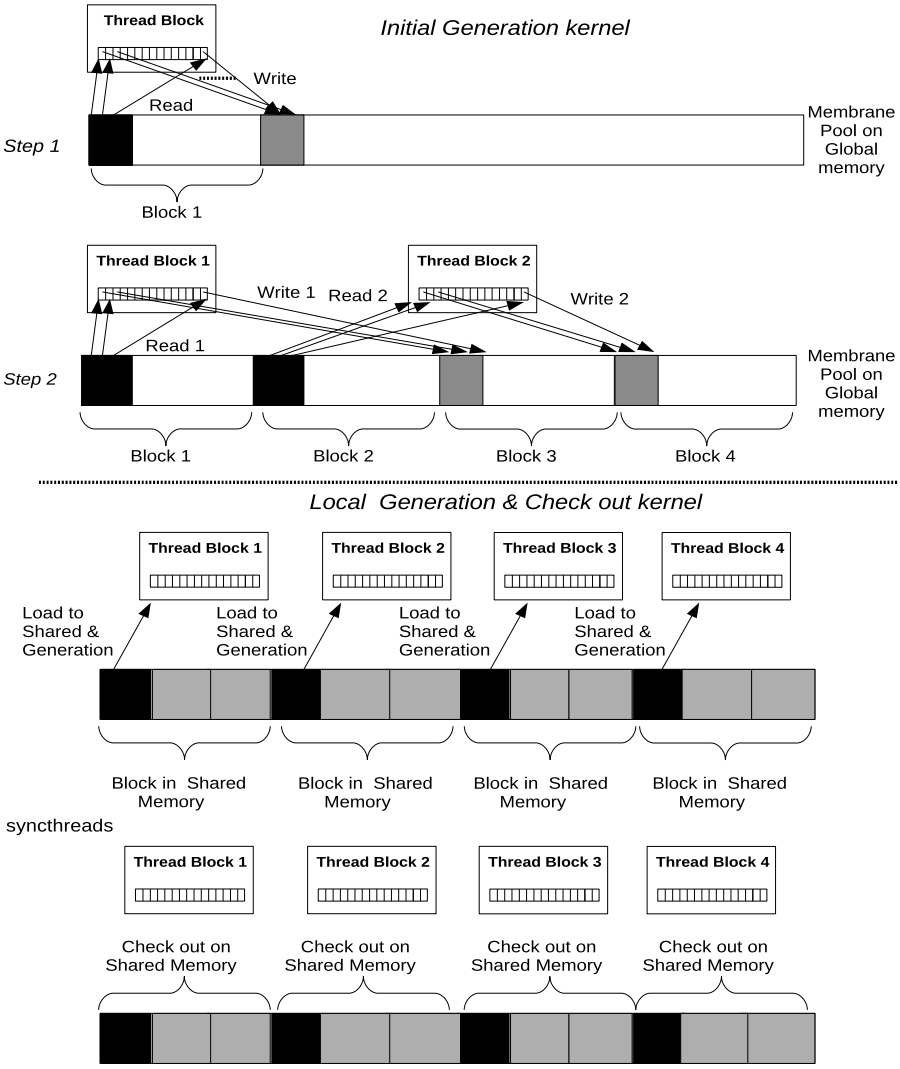

Fig. 1 Tiled simulator on a single GPU

5 Performance evaluation

In this section, we compare and contrast the performance of our ACO and P system implementations on the benchmark programs and problem instances.

5.1 ACO solving TSP

Our initial focus investigates floating-point math library calls to *powf()* in the choice_info kernel in ACO tour construction. The analogous CUDA function, *__powf()*, maps directly to hardware [20] and can be used a replacement with neg-

Table 1 Execution times (milliseconds) on the GPU system for the choice_info kernel, using both CUDA instructions: *powf* and *__powf*. Execution times (ms.) of task and data-based parallelism for our tour construction stage of the ACO algorithm on a Tesla C2050 GPU. We vary the TSPLIB benchmark instance to increase the number of cities

TSPLIB instance	choice_info kernel				Tour construction parallelism GPU Tesla C2050		
	GPU Tesla C1060		GPU Tesla C2050		Task	Data	Data speedup
	powf()	*__powf*()	*powf*()	*__powf*()			
d198	0.06	0.03 (1.9x)	0.03	0.01 (2.8x)	29.37	4.32	6.79x
a280	0.10	0.05 (2.1x)	0.06	0.02 (3.1x)	70.52	11.94	5.90x
lin318	0.13	0.06 (2.2x)	0.08	0.02 (3.3x)	153.33	15.18	10.10x
pcb442	0.23	0.10 (2.2x)	0.14	0.04 (3.4x)	301.66	38.73	7.78x
rat783	0.71	0.30 (2.3x)	0.44	0.11 (3.7x)	1375.38	207.08	6.64x
pr1002	1.15	0.49 (2.3x)	0.71	0.18 (3.8x)	2437.34	392.56	6.20x
pr2392	6.50	2.73 (2.5x)	4.04	1.03 (3.9x)	29792.03	5092.27	5.85x

ligible accuracy losses, as its omission of special arithmetic cases is irrelevant for our computational problem. The performance gains from using CUDA *__powf()* are shown in Table 1 for a range of TSP benchmark instances, giving up to 3.9x and 2.5x, respective performance gains for Tesla C2050 and C1060.

5.1.1 Tour construction evaluation

Table 1 additionally presents the performance benefit of data versus task based parallelism for the tour construction stage of the ACO algorithm on the C2050 GPU. For task parallelism, we use 16 CUDA threads with 16 ants running in parallel per thread-block, in order to maximize performance. This produces a low GPU resource usage per SM, and is not well suited for developing high-throughput GPU applications. The heavy-weight threads of this design need resources in order to execute their tasks independently and to avoid large serialization phases. In CUDA, this is obtained by distributing those threads amongst SMs, by increasing the number of thread-blocks during execution.

For data parallelism, the number of threads in each CUDA block is under programmer control, and we have empirically demonstrated that: (1) the 64 thread-block configuration reaches peak performance on an Tesla C1060, except for the 2392 cities benchmark instance, where this block-size is not allowed due to register constraints, and (2) the 128 thread-block configuration is optimal for all benchmark instances on an Tesla C2050.

Classic *Roulette Wheel* (RW) selection compromises GPU parallelism, but our *Independent Roulette* (I-R) selection method gives up to a 2.12x factor gain compared to classic roulette wheel on an Tesla C1060, and even higher gains on an Tesla C2050 (2.36x). The average performance gain is 2.00x and 1.87x on C1060 and C2050, respectively, because algorithm parallelism is increased at the cost of generating additional random numbers. Better performance is obtained by the older Tesla C1060 in tour construction on the pr1002 and pr2392 TSP benchmarks, and this difference

Table 2 Execution times (ms) for the tour construction stage of the ACO algorithm on different hardware platforms (single core CPU vs. GPUs) and enabling data parallelism on the GPU

TSPLIB instance	CPU Xeon E5620	GPU Tesla C1060		GPU Tesla C2050	
		Execution time	Speed-up vs. CPU	Execution time	Speed-up vs. CPU
d198	43.01	3.34	12.88x	4.32	9.95x
a280	151.99	10.81	14.06x	11.94	12.72x
lin318	223.78	12.62	17.67x	15.18	14.69x
pcb442	618.07	26.98	22.90x	38.73	15.95x
rat783	3539.01	168.27	21.03x	207.08	17.08x
pr1002	7965.17	334.05	23.84x	392.56	20.33x
pr2392	110573.22	4264.46	25.93x	5092.27	21.71x

widens by using roulette wheel (RW) as a selection process instead of our method. In the Tesla C1060, the 64 thread-block configuration produces better results, as thread-level parallelism is severely affected by sequential parts of the RW algorithm. This is partially solved by our selection procedure I-R, increasing the best configuration thread-block layout to 128 thread-block on Tesla C2050 architecture. The 64 thread-block configuration remains the best thread block layout on the Tesla C1060 case. This fact highlights that the Fermi architecture requires more thread-level parallelism to fully exploit the pipeline on each SM, and thus hide and amortize operation latency. Although this is partially fulfilled by our selection approach, the tour construction stage still has some sequential parts that compromise parallelism, such as the final decision to the next city to visit by each thread-block, marking this city as visited, and so on.

Finally, Table 2 shows execution times for our ACO algorithm. Notable gains are found on the GPU side, which are also favored with the problem size, as expected. Paradoxically, we note the unexpected higher performance on the Tesla C1060 versus the newer GPU, C2050. After a thorough study of architectural features on those two GPUs, we find that the amount of parallelism in the ACO algorithm can be compromised by several parts which are inherently sequential, and may produce long-stall instructions. Those instructions should be hidden by other parallel warps, but if we analyze the number of in-flight threads that each GPU is able to run at a given time, for the C2050 we have 21504 (a Fermi architecture with 1536 threads/SM and 14 SMs), whereas the C1060 can run up to 30720 (a GT200 architecture with 1024 threads/SM and 30 SMs). Therefore, 9216 threads more can be executed in-flight on the Tesla C1060, which means having 288 warps ready to prevent severe stalls during execution.

5.1.2 Pheromone update evaluation

Figure 2 shows the performance evaluation of the pheromone kernel strategies we have developed in this work. The baseline code is our optimal kernel version, which uses atomic instructions and shared memory, along with the known tradeoff between the number of accesses to global memory for avoiding costly atomic operations and the number of those operations. The "scatter to gather" computation pattern exhibits a

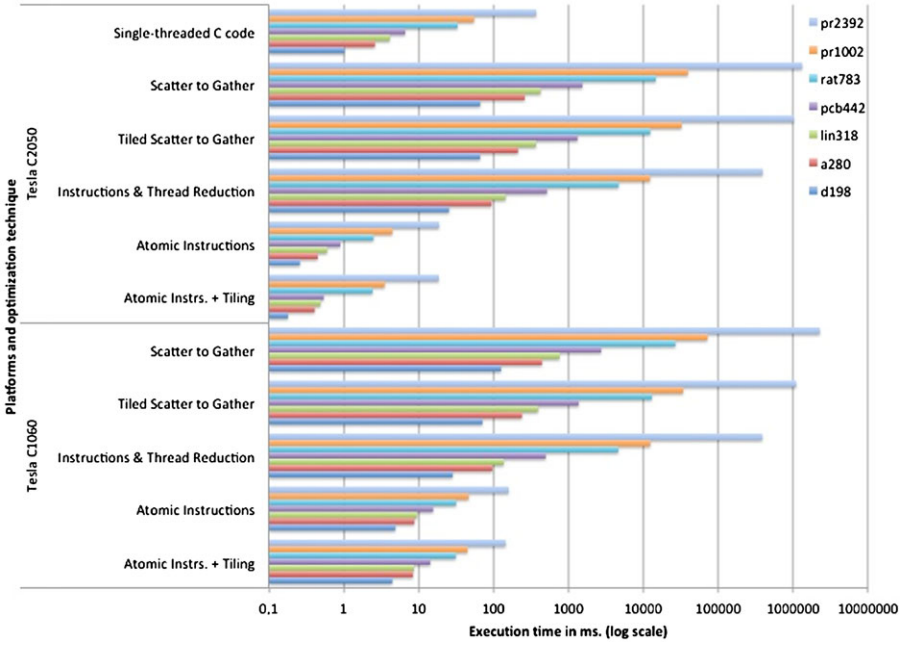

Fig. 2 Execution times (ms) on Teslas C1060, C2050, and single-threaded C version for the ACO pheromone update implementation. Different optimizations techniques are represented on y-axis

major imbalance between these two parameters, which is reflected by an exponential performance degradation as the problem size increases. Tiling improves application bandwidth in the scatter to gather approach. Reduction diminishes the total number of accesses to either shared or device memory by halving the number of threads with respect to versions 4 and 5 (and also using tiling to alleviate device memory use). Even though the number of loads per thread remains the same, the overall number of loads in the *application* is reduced.

Figure 2 also shows on the lower side the best version of the pheromone update kernel compared to a single-threaded sequential code executed on a CPU. The computational pattern for this kernel is based on data-parallelism, showing a linear speed-up along with the problem size. However, the lack of atomic operation support on an Tesla C1060 for floating point operations results in better performance for the sequential kernel on small benchmark instances. Whenever the level of parallelism increases, performance rises, reaching up to 2.56x speed-up for Tesla C1060 and 20.04x for Tesla C2050.

5.2 P systems solving SAT

Table 3 presents execution times for SAT experiments on the two high-end Tesla GPUs. The GPUs are consistently at least two orders of magnitude faster if we compare the baseline version against a single-threaded CPU, and three orders faster when we consider the *GPU Enhanced* version, which reaches a 15–16x speed-up factor

Table 3 Execution times (ms) for SAT on different hardware platforms. The number of literals is $k = 256$. The best configuration for our GPU tiled simulator is shown on each case (n.a. means "not available" due to shared memory constraints)

Number of membranes	CPU Xeon E5530	GPU generic		GPU baseline		GPU enhanced	
		Tesla C1060	Tesla C2050	Tesla C1060	Tesla C2050	Tesla C1060	Tesla C2050
2^{13}	800.47	3130.93	2348.20	10.09	7.61	0.82	0.62
2^{14}	1659.92	6678.89	5009.16	20.15	15.18	1.55	1.20
2^{15}	3382.49	16948.77	12711.58	39.99	29.59	2.90	2.30
2^{16}	6888.05	37841.52	28381.14	80.40	58.93	5.65	4.37
2^{17}	14211.80	69581.05	52185.78	163.06	119.36	11.16	8.71
2^{18}	28995.10	n.a.	n.a.	327.38	255.71	22.06	17.15
2^{19}	59521.80	n.a.	n.a.	657.97	474.39	44.69	33.16
2^{20}	121199.67	n.a.	n.a.	1328.58	955.24	88.48	69.03
2^{21}	247467.00	n.a.	n.a.	2675.85	2782.58	171.04	127.85

versus the baseline implementation. This is mainly because it saves 50 % of the synchronization time on average per iteration, and avoids duplicating a number of accesses to device memory, thus reducing the total number of operations carried out.

Considering the slowest GPU time (on Tesla C1060) for the optimized version, speed-up is 976x when the simulation covers 2^{13} membranes, and reaches up to 1446x when it is extended to 2^{21} membranes. The main reason for this concerns memory bandwidth, which is much higher on GPUs. On small data sets, memory latency plays its role, but when the working data set size grows exponentially, as in this benchmark, Table 3 also shows the performance degradation of the Generic simulator (see Sect. 3.2), even with the single-threaded CPU. The Generic version benefits from adapting the P system computation solving the `SAT` problem, and also from using the blocking technique. Besides, a thread per object (or set of objects) is defined in the alphabet Γ for each block to enable a wide range of recognizer P systems to be simulated. This generality prevents race conditions and synchronization issues, which can be solved on the targeted P system, but which causes a performance degradation, leading to a slower code execution on GPUs compared to the single threaded CPU (C++) version. We note that no more than 17 CNF formula variables may be used by the Generic version, in order to stay within available GPU memory limits.

5.2.1 P systems solving SAT: Improvements through tiling

Table 4 quantifies the benefit of using tiling in our P system. Up to 1.4x speed-up gains are achieved versus the non-tiled counterpart on GPUs, with the best performance coming on the newer Tesla C2050. Tiling becomes more effective with caches, and the C2050 enables a 16 KB L1 cache plus a 768 KB L2 cache together with 3x more shared memory space in which to allocate membranes. GPUs can significantly improve the achieved memory bandwidth through shared memory usage under explicit programmer control on memory bounded applications such as presented in this paper.

Table 4 Execution times (ms) on different GPU platforms and enabling tiling on P systems solving the SAT problem. The number of literals is $k = 256$

Number of membranes	GPU Enhanced		GPU Tiled	
	Tesla C1060	Tesla C2050	Tesla C1060	Tesla C2050
2^{13}	0.82	0.62	0.64	0.37
2^{14}	1.55	1.20	1.15	0.66
2^{15}	2.90	2.30	2.17	1.24
2^{16}	5.65	4.37	4.23	2.37
2^{17}	11.16	8.71	8.29	4.65
2^{18}	22.06	17.15	16.46	9.19
2^{19}	44.69	33.16	32.79	18.27
2^{20}	88.48	69.03	65.51	36.65
2^{21}	171.04	127.85	130.96	73.23

Table 5 Execution times (ms) on a single GPU for the particular case of a P system composed of 2^{21} membranes. Communication and initialization times (runtime overhead) are not accounted for (n.a. means "not available" due to shared memory constraints)

Tesla archit.	Algorithm stage(s)	CUDA Block size (in membranes)				
		2	4	8	16	32
C1060	Block Preprocessing (BP)	83.59	41.56	20.68	N.a.	N.a.
	Generation and Check out	113.12	104.02	103.61	N.a.	N.a.
	All (total execution time)	**196.71**	**145.58**	**124.29**	N.a.	N.a.
C2050	Block Preprocessing (BP)	32.51	16.17	8.10	4.13	2.02
	Generation and Check out	85.71	75.90	65.12	65.63	101.24
	All (total execution time)	**118.22**	**92.07**	**73.22**	**69.76**	**103.26**

5.2.2 P systems solving SAT: Block size and shared memory use

Table 5 shows the breakdown of the total execution time for a single GPU executing with $n = 21$ variables and tiling. These numbers also evaluate the impact of the data block size, which is limited by the on-chip shared memory space (16 KB for Tesla C1060 and 48 KB for Tesla C2050). Considering these constraints, we are able to measure performance for 2, 4, and 8 membranes per block on the Tesla C1060, and for 2, 4, 8, 16 and 32 membranes per block on the Tesla C2050. Note that the number of global memory accesses and the number of iterations in the *Block Preprocessing* kernel depend intrinsically on the block size. In particular, eight membranes per block require half of the memory accesses and computations (that is, iterations) compared to the four membranes per block case. Similarly, four membranes cut down to a half those required by the two membranes per block case. Furthermore, memory accesses in the *Generation* and *Check out* stages are similarly reduced as long as the block size increases. However, GPU resource occupancy worsens for the eight membranes per block case, because shared memory usage per block prevents the allocation of

more than one block per GPU. As a result, the overall improvement is barely 14 % versus the four membranes per block case on the Tesla C2050, and then the situation worsens if the parameter is increased further (shared memory constraints limits this to 8 on the Tesla C1060).

6 Summary and conclusions

This paper provides insights into the behaviour of nature-inspired Ant Colony Optimisation (ACO) and P system methods on GPUs. The main inefficiencies of baseline implementations are identified, and significant performance improvements of several orders of magnitude are obtained through various optimizations. Our key insights/contributions are: (1) Floating point arithmetic is the flagship of GPU computing. However, some mathematical operators (such as powf) can drastically affect the overall performance of GPUs when implemented at software level. (2) At a higher level, the task parallelism used by early algorithm implementations does not fit well with GPU architecture, and a *data parallelism* approach is proposed exploiting SIMD and CUDA. Algorithmic modifications for SIMD and the exploitation of GPU resources are presented. (3) A set of strategies aimed at avoiding atomic instructions, and tradeoffs for increasing application bandwidth is presented.

Overall, the gains achieved by our implementations reach impressive speed-up factors of 4–5 orders of magnitude for our largest benchmarks on Teslas C1060 and 2050 GPUs. However, our most valuable contribution is to demonstrate the research potential for nature-inspired algorithms on GPU platforms.

Acknowledgements Partially supported by a travel grant from project EU FP7 NoE HiPEAC IST-217068, and by the Spanish MEC and European Commission FEDER funds under grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2009-14475-C04-02", and also by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 15290/PI/2010. We also thank NVIDIA for hardware donation under Professor Partnership 2008–2010, CUDA Teaching Center 2011–2013, CUDA Research Center 2012 and CUDA Fellow 2012 Awards.

References

1. Blum C (2005) Ant colony optimization: introduction and recent trends. Phys Life Rev 2(4):353–373
2. Cecilia JM, García JM, Guerrero GD, Martínez-del-Amor MA, Pérez-Hurtado I, Pérez-Jiménez MJ (2010) Simulation of P systems with active membranes on CUDA. Brief Bioinform 11(3):313–322
3. Cecilia JM, García JM, Guerrero GD, Martínez-del-Amor MA, Pérez-Jiménez MJ, Ujaldón M (2010) P systems simulations on massively parallel architectures. In: Third international workshop on parallel architectures and bioinspired algorithms (WPABA' 10), in conjunction with the nineteenth international conference on parallel architectures and compilations techniques (PACT' 10), Vienna, Austria
4. Díaz-Pernil D, Pérez-Hurtado I, Pérez-Jiménez MJ, Riscos-Núñez A (2008) P-lingua: a programming language for membrane computing. In: Proceedings of the 6th brainstorming week on membrane computing, Sevilla, Spain
5. Dorigo M (1992) Optimization, learning and natural algorithms. Dissertation, Politecnico di Milano
6. Dorigo M, Birattari M, Stützle T (2006) Ant colony optimization. IEEE Comput Intell Mag 1(4):28–39
7. Dorigo M, Bonabeau E, Theraulaz G (2000) Ant algorithms and stigmergy. Future Gener Comput Syst 16:851–871

8. Dorigo M, Colorni A, Maniezzo V (1991) Positive feedback as a search strategy. Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy, Tech Rep 91-016
9. Dorigo M, Maniezzo V, Colorni A (1996) The ant system: optimization by a colony of cooperating agents. IEEE Trans Syst Man Cybern, Part B 26:29–41
10. Dorigo M, Stützle T (2004) Ant colony optimization. Bradford Company, Scituate
11. Fu J, Lei L, Zhou G (2010) A parallel ant colony optimization algorithm with GPU-acceleration based on all-in-roulette selection. In: Proceedings of the third international workshop on advanced computational intelligence (IWACI), Suzhou, China
12. Garland M, Le Grand S, Nickolls J, et al (2008) Parallel computing experiences with CUDA. IEEE Micro 28:13–27
13. Garland M, Kirk DB (2010) Understanding throughput-oriented architectures. Commun ACM 53:58–66
14. Goldberg DE (1989) Genetic algorithms in search, optimization and machine learning. Addison-Wesley, Longman, Reading, Harlow
15. Johnson DS, Mcgeoch LA (1997) The traveling salesman problem: a case study in local optimization. In: Aarts EHL, Lenstra JK (eds) Local search in combinatorial optimization. Wiley, London, pp 215–310
16. Lam MD, Rothberg EE, Wolf ME (1991) The cache performance and optimizations of blocked algorithms. ACM SIGPLAN Not 26(4):63–74
17. Lawler E, Lenstra J, Kan A, Shmoys D (1987) The traveling salesman problem. Wiley, New York
18. Li J, Hu X, Pang Z, Qian K (2009) A parallel ant colony optimization algorithm based on fine-grained model with GPU acceleration. Int J Innov Comput, Inf Control 5(11):3707–3715
19. NVIDIA CUDA C Programming Guide 4.0 (2011)
20. NVIDIA CUDA C Best Practices Guide 4.0 (2011)
21. NVIDIA, Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi (2009)
22. Păun G (2002) Membrane computing: An introduction. Springer, Berlin
23. Păun G (2000) Computing with membranes. J Comput Syst Sci 61:108–143. TUCS Report No 208
24. Pérez-Jiménez MJ, Romero-Jiménez Á, Sancho-Caparrini F (2003) Complexity classes in models of cellular computing with membranes. Nat Comput 2(3):265–285
25. Qasem M (2009) WinSAT website. http://users.ecs.soton.ac.uk/mqq06r/winsat
26. Reinelt G (1991) TSPLIB: A traveling salesman problem library. ORSA J Comput 3(4):376–384
27. Scavo T (2010) Scatter-to-gather transformation for scalability
28. TSPLIB Webpage (2011) http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/
29. You YS (2009) Parallel ant system for traveling salesman problem on GPUs. In: GECCO 2009—GPUs for genetic and evolutionary computation, pp 1–2