

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 multi-core 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.

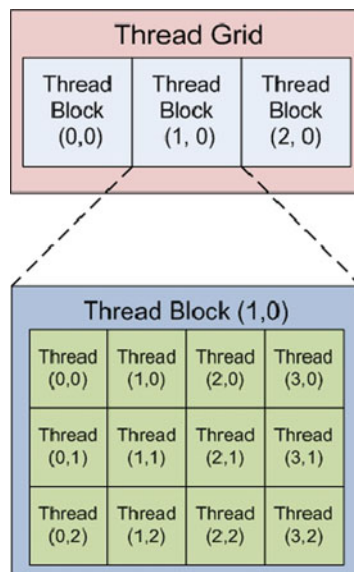
Flynn's taxonomy [3] splits up parallel computing systems into three categories based on their execution capabilities (Flynn actually describes four total categories of computing systems; however, the Single Instruction Single Data category is not a parallel system but a uniprocessing system). The two 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:

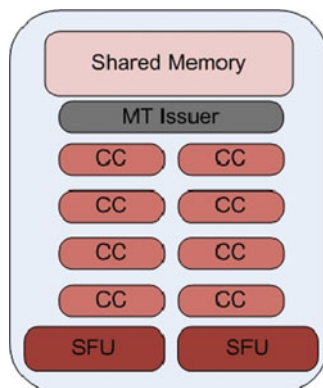
1. Launching another kernel—after invoking one kernel, attempting to launch another will result in the CPU application halting until the previous kernel has completed execution (effectively “synchronizing” all threads in the thread grid, as they must all have completed execution of the first kernel).
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

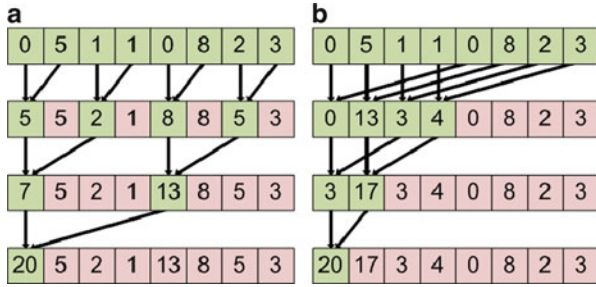


Fig. 3 Two styles of parallel add reduction on an array of elements

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, \dots, v_n , we apply some associative operator, \oplus , to the elements, $v_1 \oplus v_2 \oplus \dots \oplus v_n$, 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

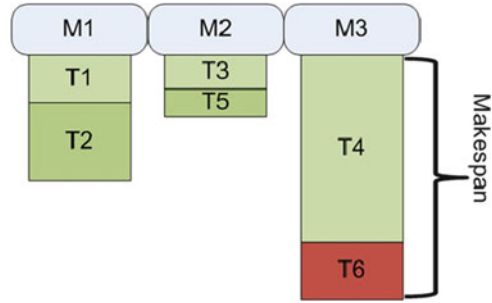
The task matching problem represents a significant problem in heterogeneous, distributed computing environments, such as grid or cloud computing. The problem involves determining the optimal assignment, or *matching*, of tasks to machines such that the total execution time is minimized. More specifically, 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:

$$V_{i+1} = w * V_i + c_1 * \text{rnd}() * (X_{\text{Pbest}} - X_i) + c_2 * \text{rnd}() * (X_{\text{Gbest}} - X_i) \quad (1)$$

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 $\text{rnd}()$ 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  $i = 0 \rightarrow \text{numIterations}$  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:

$$V_{i+1} = V_{i+1} + c_3 * \text{rnd}() * f(X_{\text{FGbest}}, X_{\text{Gbest}}, X_i) \quad (2)$$

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, $V = (t_0, t_1, \dots, t_{|T|-1})$, 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 $|M| - 1$.

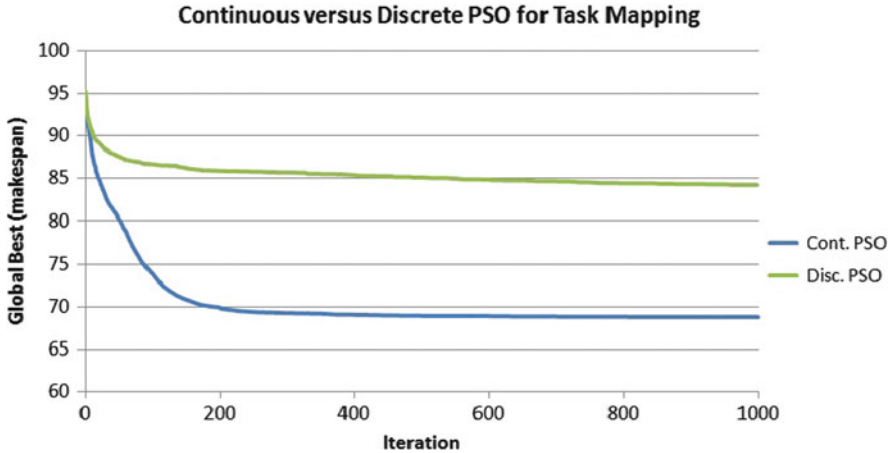


Fig. 5 Global best results for continuous and discrete PSO by iteration

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

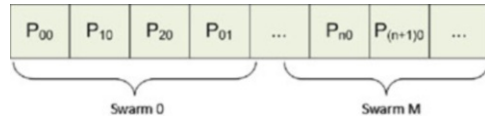


Fig. 6 Global memory layout of position, velocity, and particle best positions (P_{xy} refers to particle x 's value along dimension y)



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 n th 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.

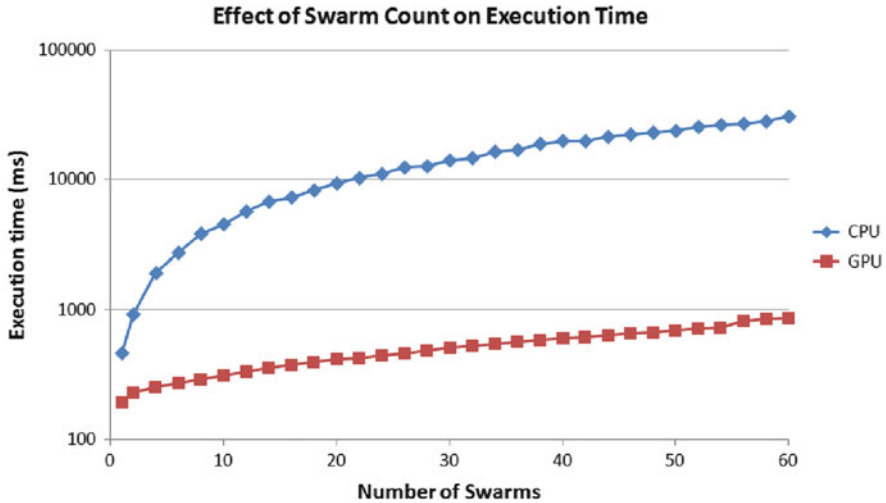


Fig. 8 Comparison between sequential CPU and GPU algorithm as swarm count increases

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.

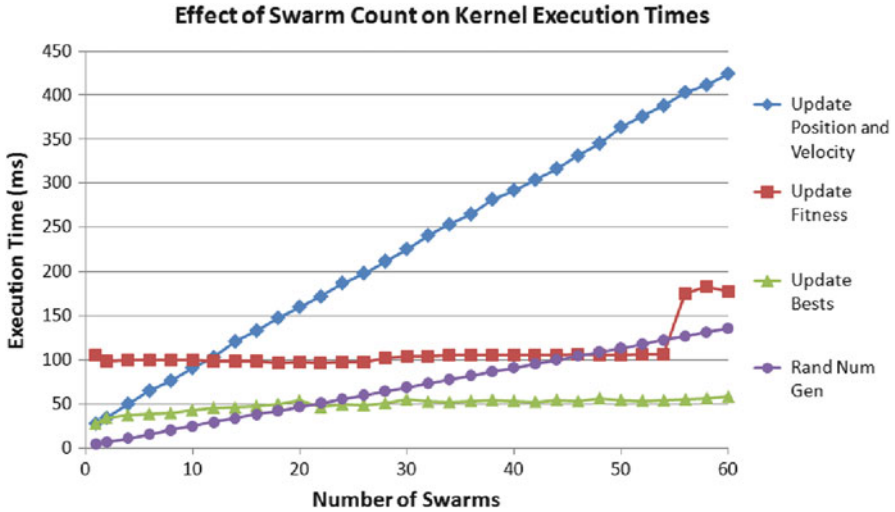


Fig. 9 Total execution time for the various GPU kernels as the swarm count increases

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-dimensional 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

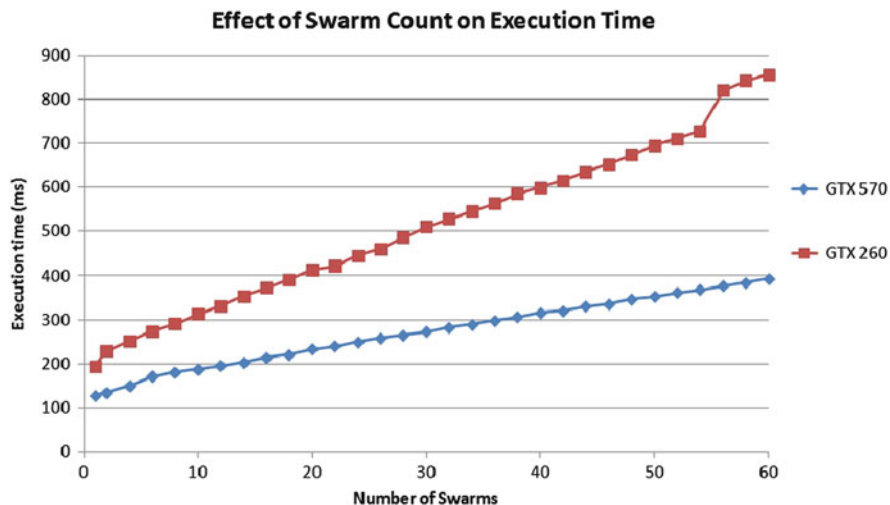


Fig. 10 Comparison between GTX 260 and 570 GPUs as swarm count increases

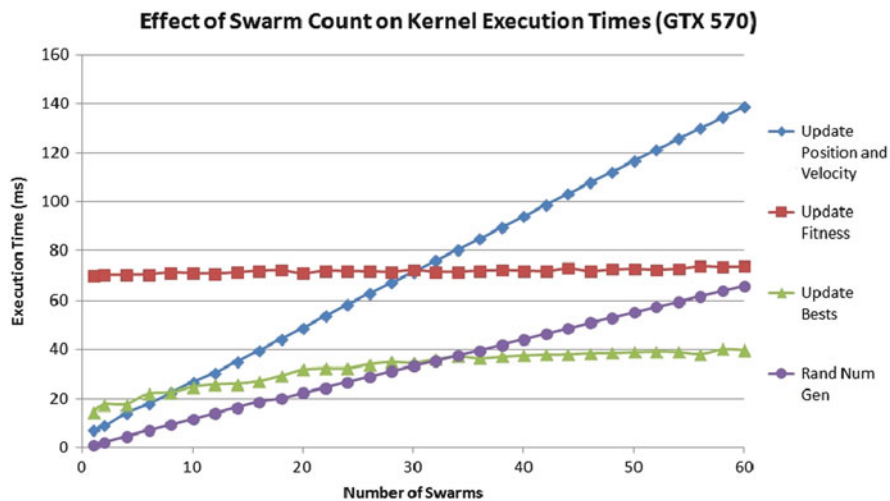


Fig. 11 Total execution time for the various GPU kernels as the swarm count increases for a GTX 570 GPU

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

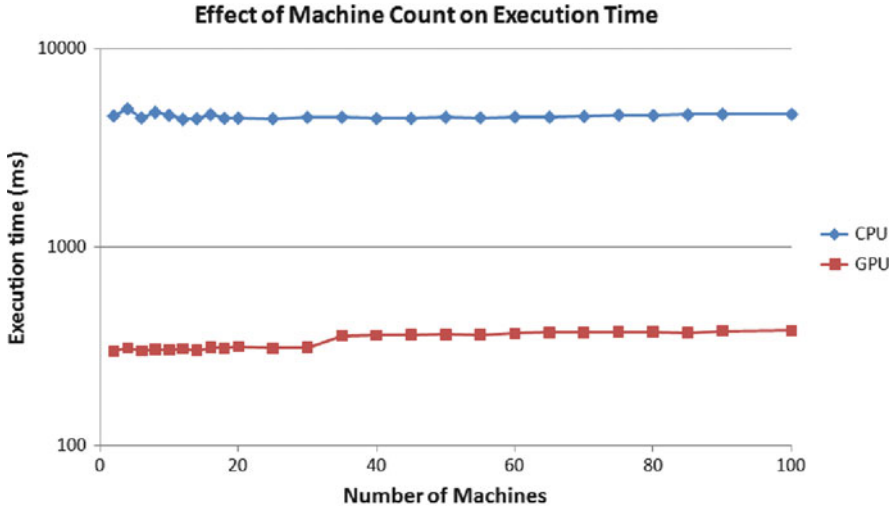


Fig. 12 Comparison between sequential CPU and GPU algorithm as machine count increases

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.

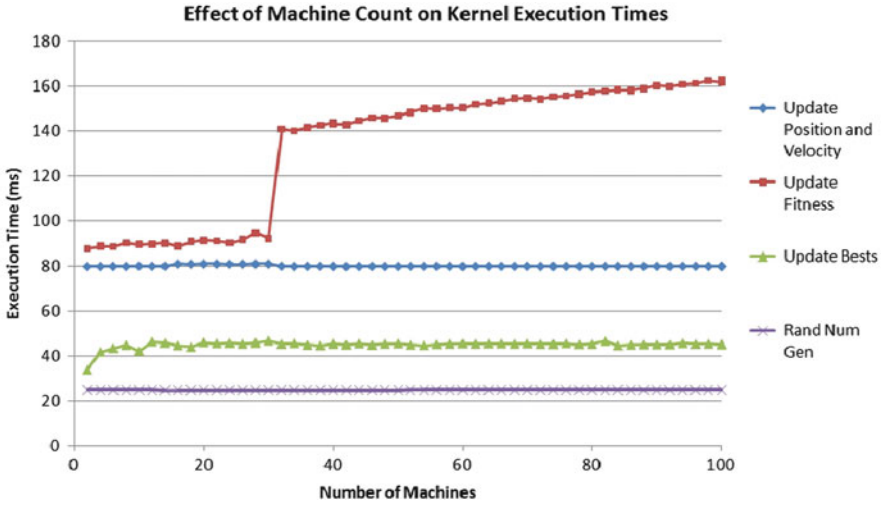


Fig. 13 Total execution time for the various GPU kernels as the machine count increases

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

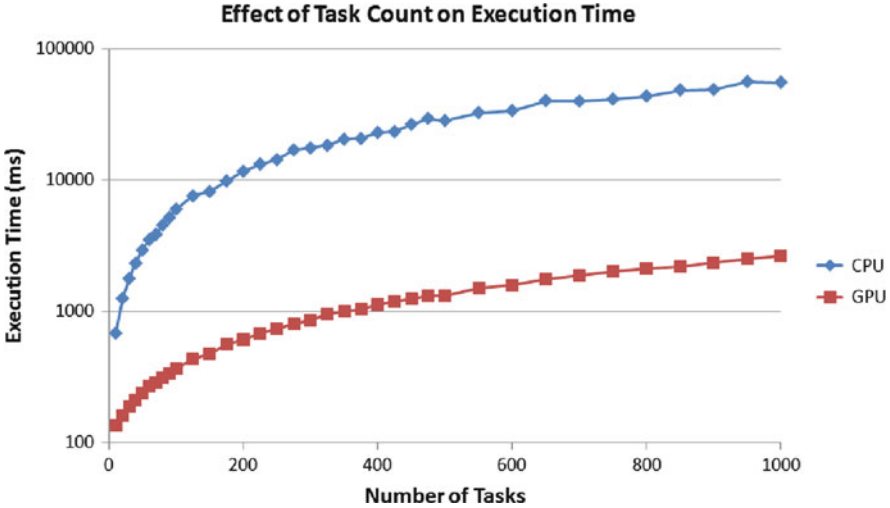


Fig. 14 Comparison between sequential CPU and GPU algorithm as task count increases

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

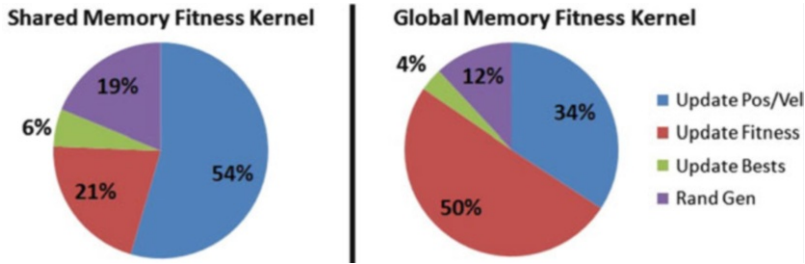


Fig. 15 Percentage of execution time taken by most significant kernels

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)