Collaborative Multi-Swarm PSO for Task Matching using Graphics Processing Units

Steven Solomon
University of Manitoba
umsolom9@cs.umanitoba.ca

Parimala Thulasiraman
University of Manitoba
thulasir@cs.umanitoba.ca

Ruppa K. Thulasiram
University of Manitoba
tulsi@cs.umanitoba.ca

ABSTRACT

We investigate the performance of a highly parallel Particle Swarm Optimization (PSO) algorithm implemented on the 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 GPUs 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.

Track: Parallel Evolutionary Systems

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms

Performance

Keywords

Particle Swarm Optimization, GPU, CUDA, Task Matching

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 increase. In response 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, min-max and min-min [1], and suffrage [2]. More recently, bio-inspired heuristic algorithms such as Particle Swarm Optimization [3] (PSO) have been used and studied for this problem [4, 5]. 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 [3], 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. [6]'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 paper is organized as follows. The next section discusses the CUDA programming model and GPU architecture, followed by a brief discussion on both single and multi-swarm PSO. Section 4 discusses the related work in PSO for task matching, parallel PSO on GPUs, and multi-swarm PSO. Section 5 provides a description of our PSO implementation on the GPU, and Section 6 shows both performance and solution quality results. Finally, we provide our conclusions and a discussion on future work in Section 7.

2. CUDA PROGRAMMING MODEL

In this section we provide a brief overview of CUDA programming, the CUDA threading model, and the GPU architecture. Nvidia [7] provides further information in their

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'11, July 12–16, 2011, Dublin, Ireland.
Copyright 2011 ACM 978-1-4503-0557-0/11/07 ...$10.00.

CUDA C Programming Guide. As we currently work with the GTX 260 GPU (based on the GT200 architecture), all information in this section pertains to that model of GPU.

From a high level perspective, the GPU is composed of two separate units: the core and the off-chip memory. The core of the GPU is composed of an array of Streaming Multiprocessors, or SMs. Each SM contains eight CUDA cores. The CUDA cores are the computational cores of the GPU, and handle the execution of threads. The execution model, which Nvidia [7] refers to as SIMT, or Single Instruction Multiple Threads, follows a strategy similar to that of Single Instruction Multiple Data (SIMD).

Each of the SMs on the GPU contains a small amount of high performance *shared* memory, 16KB in total for the GTX 260 GPU. Nvidia [7] claims that accesses to shared memory are as fast as accessing a register if no bank conflicts exist. As shared memory is split in to 16 32-bit wide banks (16 threads at a time perform a memory transaction in parallel), bank conflicts occur when multiple threads request data from the same bank. These conflicting requests are serviced in serial, rather than in parallel. However, in the case where all threads access the same location, the value is broadcast to all threads at once.

In contrast to shared memory, the global memory is the largest memory space available on the GPU and is read/write accessible to all threads. Unfortunately global memory access carries a significant latency penalty on the order of 400 to 800 cycles [7]. As an added downside, global memory accesses are not cached. There are, however, structured access patterns which allow threads to collaborate on their accesses, resulting in global memory coalescing [7]. This coalescing effect reduces the total number of memory transactions required to service groups of threads.

Stemming from global memory we have the texture and constant memories. These are read-only memories that exist within global memory, and thus still have the high latency costs. The difference with these memories, however, is that the hardware provides small caches at the SM level for data accesses from these memories.

With CUDA, threads are grouped into a few organizations. At the lowest level, sets of 32 threads are organized into a warp. Each thread within a warp is given the same instruction to execute and they all execute on the same SM. The previously mentioned memory coalescing technique becomes important when discussing threads at the warp level. In the best case, this technique reduces the 32 individual memory requests (one from each thread in the warp) to only 2 memory requests (recall that memory requests are handled 16 threads at a time). In the GT200 architecture, coalescing occurs when at least two threads in the half-warp are accessing data from the same segment in global memory. Hence, every thread in the half-warp that accesses data from the same segment in global memory can (and will) have their requests combined (or coalesced) into a single request. Clearly, the most beneficial effect occurs if one can structure the data accesses such that all 16 threads access from the same segment. Proper exploitation of coalescing leads to tremendous improvements in the performance of global memory and, correspondingly, large improvements in the overall execution time of a GPU algorithm.

Sets of warps that are executing on the same SM (but not necessarily *all* warps executing on an SM) are organized into thread blocks. A thread block is given its own exclusive access to a piece of the shared memory for the SM it is executing on. Finally, thread blocks are then further organized into the thread grid. The thread grid encapsulates all threads involved in the execution of a GPU kernel/application.

3. PARTICLE SWARM OPTIMIZATION

The Particle Swarm Optimization (PSO) algorithm, first described by Kennedy and Eberhart [3] 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 paper, we follow the modified PSO algorithm as established by Shi and Eberhart [8]. 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 rnd() generates a uniformly distributed random number between 0 and 1. w, the inertial weight factor along with c_1 and c_2 provide 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 → numIterations do
    for all particles in swarm do
        Compute fitness of current location
        Update X_Pbest if necessary
        Update X_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. [9], 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. [9] 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. [9], 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.

4. RELATED WORK

4.1 Multi-Swarm PSO

There exists a wide variety of multi-swarm PSO implementations in the literature. One such work by van den Bergh and Engelbrecht [10] considers tasking each swarm with the optimization of one of the problem's dimensions. The authors showed that their solution provides better solutions as the number of dimensions increases. Another example by Liang and Suganthan [11] modify the dynamic multi-swarm algorithm which works by initializing a small number of particles in each swarm and randomly moving particles between each swarm after a given number of iterations have passed. The authors augment this algorithm by including a local refining step that occurs every given number of iterations on the top solutions found thus far.

As was previously mentioned, we follow the work described by Vanneschi et al. [9] for our implementation on the GPU. Their "MPSO" algorithm solves an optimization problem via multiple swarms that communicate by moving particles amongst the swarms. They describe a modification of 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 performing even better than MPSO.

4.2 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 [12] and Zhou and Tan [13] both describe a simple implementation of PSO on the GPU. Their implementations use a single swarm and splits the major portions of PSO into separate kernels, with one thread managing each particle. In order to generate random numbers Veronese and Krohling [12] use a GPU implementation of the Mersenne Twister pseudo-random number generator, whereas Zhou and Tan [13] use the CPU to generate pseudo-random numbers and transfer these to the GPU. When applied to benchmarking problems, the authors of both works show that their GPU algorithms outperforms sequential CPU implementations.

Mussi et al. [6] provide another, more recent GPU implementation of PSO. In this case, the authors assign a single thread to a single dimension for each particle. Mussi et al. [6] test their algorithm against benchmarking problems with up to 120 dimensions, and show that the parallel GPU algorithm vastly outperforms a sequential application. Finally, the authors mention in passing the ability to run multiple swarms, but do not elaborate or test such situations.

In all of these cases, the theme has been parallelizing single swarm PSO (or multiple swarms with no described strategy for collaboration or even testing). In the most recent case, Mussi et al. [6] have provided an extremely fine-grained implementation of PSO that takes advantage of the threading capabilities of the GPU. The authors only test with cases up to 120 dimensions and 32 particles, however. In our work, we wished to test not only with many more dimensions, but also with many more particles. 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.

4.3 PSO for Task Matching

Applying PSO to the task matching/mapping problem has been studied in the past by various groups. In essence, there are two styles of PSO applied to this problem: 1. the original, continuous algorithm, and, 2. discrete PSO which locks the particles to integer values along the solution space. In all of the works discussed, the authors define the solution space as an n dimensional space, one dimension per task. The location within a dimension defines the machine that a given task is matched to. Zhang et al. [5], for example, apply the continuous PSO algorithm to the task mapping problem. The authors further make use of the Smallest Position Value (SPV) technique (described by Tasgetiren et al. [14]) to generate a position permutation from the location of the particles. They compare their PSO algorithm to a genetic algorithm and show that PSO provides improved performance. A recent work by Sadasivam and Rajendran [15] also consider the continuous PSO algorithm coupled with the SPV technique. The authors focus their efforts on providing load balancing between grid resources. Unfortunately, they compare their PSO algorithm only to that of a random algorithm (with promising results, however).

Opposite to continuous PSO, Kang et al. [16] propose an implementation of discrete PSO for task mapping on the grid. They compare 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. [4] described a similar discrete PSO solution with favourable results compared to the max-min algorithm. Both of these works test only small problem sizes (< 100 tasks).

5. COLLABORATIVE MULTI-SWARM PSO ON THE GPU

To lead into our description of the GPU implementation, we will first discuss the main concepts of multi-swarm PSO for task matching without consideration of the GPU architecture. From this groundwork we can then move on to discuss the specifics of the GPU version itself.

To start, 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, and, 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 these same two components: 1. The total number of tasks, $|T|$, and, 2. The total number of machines, $|M|$. A solution for the task matching problem consists of a vector, $V = (t_1, t_2, ..., t_{|T|})$ 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 Machine Available Time (MAT) of the solution. The MAT of a machine is nothing more than the total amount of time required by the machine to complete all tasks assigned to it. 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.

Similar to the work described in Section 4.3, 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 $|M| - 1$.

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 4.32427, but, rather, must be mapped to machine 4 or 5. Unlike Kang et al. [16] or Yan-Ping et al. [4], 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 Figure 1. However, unlike Zhang et al. [5] or Sadasivam and Rajendran [15] 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.

5.1 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. Finally, we store a set of pre-generated random numbers in global memory as well. Each of these distinct sets of data are stored in their own one-dimensional array in global memory.

For position, velocity, and particle/swarm-best positions, we store the dimensional values for the particles of a given swarm in a special ordering. Rather than group each dimension up for one particle and then moving on to the next, we group the values up by dimension. Figure 2 provides an example of how this data is stored (swarm-best positions are stored per swarm, rather than per 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 Section 5.2, however it is suffice to say that this ensures all kernels data accesses to these memory locations will be coalesced. 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 3 shows this organization.

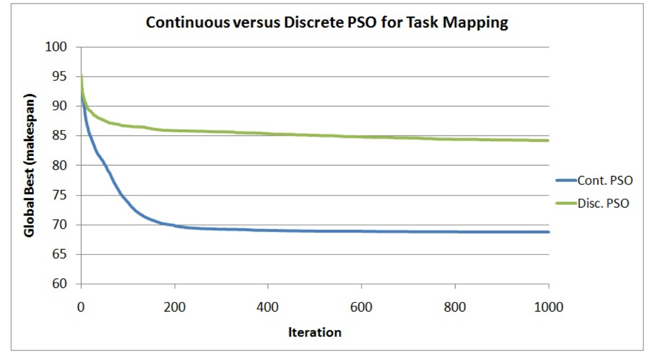

Figure 1: Global best results for continuous and discrete PSO by iteration.

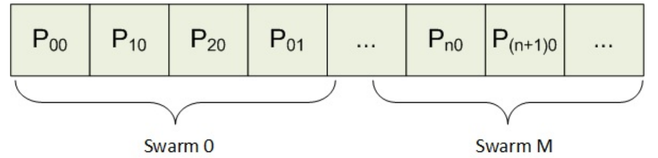

Figure 2: Global memory layout of position, velocity, and particle best positions (P_{xy} refers to particle x's value along dimension y).

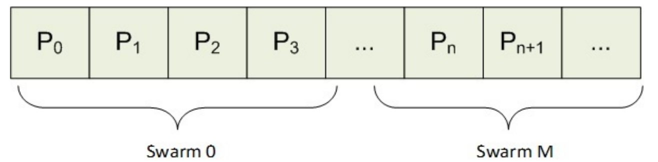

Figure 3: Global memory layout of fitness and particle best values.

The aforementioned ETC matrix represents a data structure that all threads require access to during the calculation of a particle's fitness (makespan). 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. As a result, we place the ETC matrix into texture memory. As discussed in Section 2, texture

memory provides a hardware cache at the SM level. As it is extremely 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.

5.2 GPU Algorithm

We lead into our description of the GPU implementation by first discussing the issue of random number generation. As we know from equation 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 [7] to generate high quality, pseudo-random numbers on the GPU. We generate a large amount of random numbers at a time (250MB worth) and then generate more numbers in chunks of 250MB or less when these have been used up.

Our implementation of multi-swarm PSO is split up into a series of kernels that map to the various phases of the algorithm. These phases and kernels are as follows:

5.2.1 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.

5.2.2 Update Position and Velocity

This phase updates the velocity of all particles using equation 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 equations 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.

5.2.3 Update Fitness

In this phase, we update the fitness values for all particles after they have moved to their new positions. This phase is more complex than the previous two phases and, unfortunately, does not exploit parallelism to the same degree. For this phase, we map a single thread to each particle, rather than to each dimension of each particle. The reason for this choice of thread-particle mapping is due to how makespan is computed. This computation involves first determining the MAT for each machine, and then taking the maximum value as the makespan.

One option for parallelization involves having a thread compute the makespan for a single machine and then performing a parallel reduction to find the makespan for each particle. The issue with this approach, however, is that a particle's position vector stores the machine each task is matched to. As a result, various tasks are assigned to various machines, we don't know which ahead of time. If we parallelized this phase at the machine MAT computation level then all threads would have to iterate through all of the dimensions of a particle's position anyways. 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. The first uses shared memory as a scratch space for computing the MAT for each machine. Each thread therefore requires $|M|$ floating-point elements of shared memory. There may be cases, however, where the number of threads per block combined with $|M|$ requires an amount of shared memory exceeding the capabilities of the GPU. In these cases we use the second kernel, where the scratch space is stored in global memory. Given only one thread block executing per SM, the first kernel can support 128 threads (particles) with a machine count of approximately 30, whereas the second kernel is not bounded by the very small amount of shared memory available.

This kernel showcases our reasoning for choosing the ordering of position elements in global memory. While computing the makespan, each thread reads the position from the current dimension in order to discover the task matching corresponding to this dimension. With our ordering, coalescing is guaranteed as each thread within a block reads an element in global memory next to those read by other threads. This coalescing results in an approximate *200%* performance improvement over an uncoalesced version.

5.2.4 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. Furthermore, we assign all threads composing each 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, and if so, the replacement occurs.

In the second step, the threads in a block 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 rely on only a single thread to accomplish this task.

5.2.5 Swap Particles

Finally, the swap particles phase replaces the n worst particles in a swarm with the n best particles of its neighboring swarm (using a ring topology to determine neighbors). This phase is composed of two separate kernels. 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 run n parallel reductions, with each reduction finding the nth best/worst particle. 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 located particles into another shared memory buffer. We then restart the reduction for finding the $n+1th$ 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.

5.2.6 *CPU Control Loop*

In our algorithm, the CPU only manages the main loop of PSO and the invocation of the various GPU kernels.

6. RESULTS

To test the performance of our GPU algorithm we compare it against a sequential collaborative multi-swarm PSO algorithm. This sequential algorithm has not been optimized for a specific CPU architecture, but it has been tuned specifically 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 3.0Ghz. 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 First-Come-First-Serve (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).

6.1 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. We further run these tests on data consisting of 80 tasks and 8 machines. As a result, the shared memory version of the fitness kernel is used throughout. Figure 4 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.

We further measured the total time taken for each of the GPU kernels as the swarm count increases. The results are shown in Figure 5. The update position and velocity kernel forms the greatest contributor to the increase in the GPU's execution time as the swarm count increases. We explain this by revisiting the overall responsibilities of this kernel. That is, the update position and velocity kernel requires a large number of global memory reads and writes (to read in the many-dimensioned position, velocity, and current bests position data), and is relatively computationally intensive when determining the new velocity. When combined with

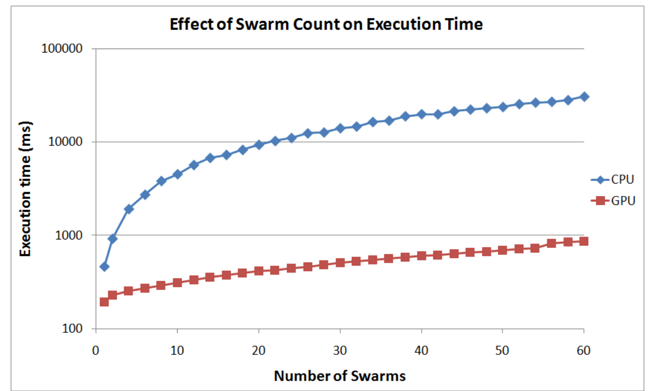

Figure 4: Comparison between sequential CPU and GPU algorithm as swarm count increases.

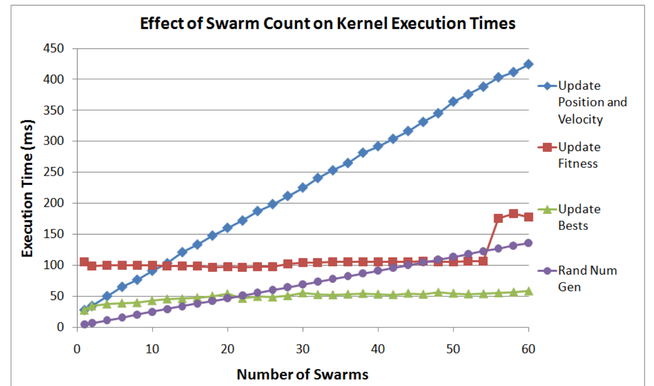

Figure 5: Total execution time for the various GPU kernels as the swarm count increases.

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 2 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.

In order to test our use of texture memory, 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.

Next, we examine the algorithm's performance and scaling for increasing machine counts as well as increasing task counts. For the machine count scaling we keep the task count and the number of swarms static at 80 and 10 respectively. Similarly, for the task count scaling we keep the machine count and number of swarms static at 8 and 10 respectively. Figure 6 shows the results for task count scaling.

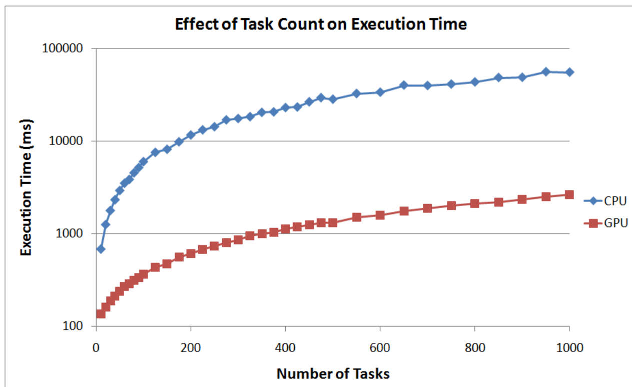

Figure 6: Comparison between sequential CPU and GPU algorithm as task count increases.

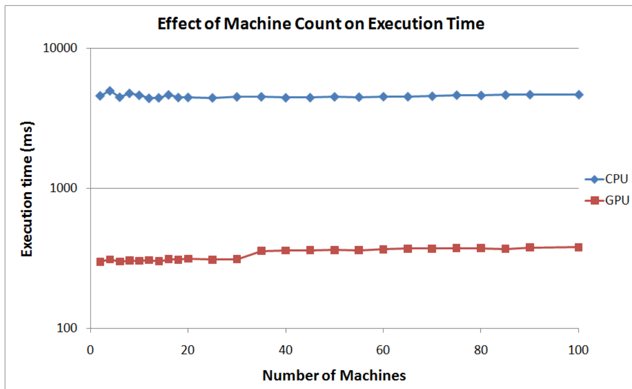

Figure 7: Comparison between sequential CPU and GPU algorithm as machine count increases.

The results are very similar to those of the swarm count tests in that the GPU algorithm significantly outperforms the sequential CPU algorithm. Overall, however, the GPU cannot provide the same level of speedup while the swarm count remains low. We expect this, as increasing the number of swarms increases the exploitable parallelism at a faster rate than the task count. While the individual kernel execution times are not shown, the update position and velocity kernel dominates the run time again as the shared memory fitness kernel is used throughout.

With the machine count scaling test we finally observe the effect that switching to the global memory fitness kernel has on the runtime. Figure 7 shows the results with machine counts from 2 to 100. Unlike the previous performance tests, we see that the execution time does not change dramatically as the machine count increases. However, the GPU execution time increases by 14% when the machine count increases from 30 to 35. 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.

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

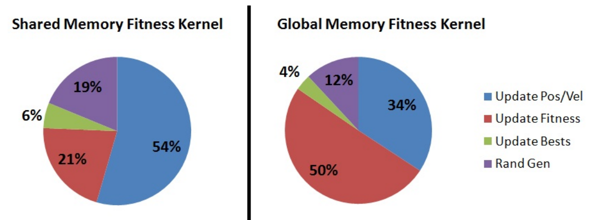

Figure 8: Percentage of execution time taken by most significant kernels.

execution time each kernel uses. Figure 8 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.

6.2 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 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, we set this value to 0.995, and run $1,000$ 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: 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
1000	100	3.106	3.109

Table 1 provides the averaged results of the solution quality experiments, normalized to the FCFS solution. We first tested small data sets of sizes similar to those from Sadasivam and Rajendran [15] as well as Yan-Ping et al. [4]. We can see from these that, unfortunately, PSO outperforms MSPSO in many of the tests. We do not conclusively know what the cause of this is, however, it is clear that, on occasion, a large number of particles within a single swarm can more readily search for an optimal solution. Furthermore, as the problem size increases, both variants of PSO fail to generate improved solutions when compared to FCFS.

7. CONCLUSIONS AND FUTURE WORK

At the start of this work 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, and, 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, we believe the majority of the contributions made in this paper 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. We leave these performance tests and modifications as future work.

8. ACKNOWLEDGEMENTS

The first author acknowledges Natural Sciences and Engineering Research Council (NSERC) Canada for the prestigious CGS Award. The last two authors acknowledge NSERC for partial financial support for this research through Discovery Grants.

9. REFERENCES

[1] Richard F. Freund, Michael Gherrity, Stephen Ambrosius, Mark Campbell, Mike Halderman, Debra Hensgen, Elaine Keith, Taylor Kidd, Matt Kussow, John D. Lima, Francesca Mirabile, Lantz Moore, Brad Rust, and H. J. Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with smartnet. In *The Seventh IEEE Heterogeneous Computing Workshop*, pages 184–199, Orlando, Florida, USA, Mar. 30 1998.

[2] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *The Eighth IEEE Heterogeneous Computing Workshop*, pages 30–44, San Jaun, Puerto Rico, Apr. 12 1999.

[3] Kennedy J. and Eberhart R. Particle swarm optimization. In *IEEE Intl. Conf. on Neural Networks*, pages 1942–1948 (vol.4), Perth, Australia, Nov. 27–Dec. 1 1995. IEEE.

[4] Bu Yan-Ping, Zhou Wei, and Yu Jin-Shou. An improved PSO algorithm and its application to grid scheduling problem. In *Intl. Symp. on Computer Science and Computational Technology*, pages 352–355, Shanghai, China, 20–22 Dec. 2008. IEEE.

[5] Lei Zhang, Yuehui Chen, Runyuan Sun, Shan Jing, and Bo Yang. A task scheduling algorithm based on PSO for grid computing. *Intl. J. of Computational Intelligence Research*, 4(1):37–43, 2008.

[6] Luca Mussi, Fabio Daolio, and Stefano Cagnoni. Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture. *Information Sciences*, In Press, Sept. 3 2010.

[7] NVIDIA. *NVIDIA CUDA C Programming Guide Version 3.2*, October 2010.

[8] Y. Shi and R. Eberhart. A modified particle swarm optimizer. In *IEEE World Congress on Computational Intelligence*, pages 69–73, Anchorage, Alaska, USA, May 4–9 1998. IEEE.

[9] Leonardo Vanneschi, Daniele Codecasa, and Giancarlo Mauri. An empirical comparison of parallel and distributed particle swarm optimization methods. In *The Genetic and Evolutionary Computation Conference*, pages 15–22, Portland, Oregon, USA, July 7–11 2010.

[10] Frans van den Bergh and Andries P. Engelbrecht. A cooperative approach to particle swarm optimization. *IEEE Trans. on Evolutionary Computation*, 8(3):225–239, June 2004.

[11] J. J. Liang and P. N. Suganthan. Dynamic multi-swarm particle swarm optimizer with local search. In *IEEE Congress on Evolutionary Computation*, pages 522–528, Edinburgh, UK, Sept. 2–4 2005.

[12] Lucas de P. Veronese and Renato A. Krohling. Swarm's flight: Accelerating the particles using C-CUDA. In *IEEE Congress on Evolutionary Computation*, pages 3264–3270, Trondheim, Norway, May 18–21 2009.

[13] You Zhou and Ying Tan. GPU-based parallel particle swarm optimization. In *IEEE Congress on Evolutionary Computation*, pages 1493–1500, Trondheim, Norway, May 18–21 2009.

[14] M. Fatih Tasgetiren, Yun-Chia Liang, Mehmet Sevkli, and Gunes Gencyilmaz. Particle swarm optimization and differential evolution for the single machine total weighted tardiness problem. *International Journal of Production Research*, 44(22):4737–4754, Nov. 2006.

[15] G. Sudha Sadasivam and Viji Rajendran. An efficient approach to task scheduling in computational grids. *Intl. J. of Computer Science and Applications*, 6(1):53–69, 2009.

[16] Qinma Kang, Hong He, Hongrun Wang, and Changjun Jiang. A novel discrete particle swarm optimization algorithm for job scheduling in grids. In *Fourth Intl. Conf. on Natural Computation*, pages 401–405, Jinan, China, Aug. 25–27 2008. IEEE.