Parallelization of Particle Swarm Optimization

E4750 Fall 2019 Report
Austin Ebel, abe2122
Columbia University

Abstract— **Particle Swarm Optimization (PSO) is a technique most often associated with cost function minimization, however it has seen limited use in practical applications as its performance suffers severely on CPUs in higher dimensional search spaces. This is because resources and runtimes scale quadratically with increasing complexity. This project aims to reduce the runtime complexity of this algorithm by making use of parallel computing techniques on GPUs. Optimal use of shared memory, block size, and data transfer techniques can result in significant performance improvements, but are often difficult to find because they are extremely application specific. This project uses an iterative approach to finding optimal combinations, in hopes of increasing the viability of Particle Swarm Optimization in modern applications.**

I. Overview

A. Problem in a Nutshell

More formally, PSO is a population-based global search algorithm wherein one initializes many agents (particles) that move around a given search space with some position and velocity in an attempt to find the extrema of functions. The magnitude and direction of velocity depends on the particle's own constraints, but also global information about the group of particles. This typically results in a swarming-like effect toward a function's minimum or maximum over many iterations. It is closely associated with the more popular technique of gradient descent [1], however differs primarily in that is much more computationally intensive and computes optimal directionality without using gradients.

While PSO, much like gradient descent, is inherently a sequential process - converging to extrema over many iterations - each particle is largely independent from one another within each iteration (and where most of the computational complexity occurs). Because of this, PSO is a prime candidate for parallelization.

Optimal use of shared memory, correct block size, and memory coalescing play a pivotal role in increasing application performance on GPUs. [2] and [3] show the extent of such optimizations. Due to the iterative nature of PSO, significant amounts of data are transmitted between the CPU and GPU during runtime. In general, memory coalescing and data reuse in shared memory play large roles in these types of iterative applications to prevent additional DRAM bursts and reduce global memory accesses. As the interplay between shared memory usage and block size is largely application specific, a brute force method was applied to find optimal parameters.

B. Prior Work

The formalization of PSO is attributed James Kennedy and Russell Eberhart in 1995 [4] as an approach to simulate the flocking of birds and schooling of fish. It was simplified after the fact as its application to optimization was realized. Following this formalization, an attempt to parallelize PSO came in 2004 with Schutte, J. et. al [5], who used a multi-core CPU to show significant performance improvement of the algorithm given load-balanced cost function evaluation. In 2009, Zhou, Y. et al [6] presented novel work implementing PSO on NVIDIA GPUs, showing up to an 11x increase in performance over traditional CPUs for a given cost function. Due to the continual increase of computational power in GPUs, PSO has seen use in the fields of image processing and medicine in [7] and [8] respectively.

II. Description

Section 2.1 summarizes the objectives and technical challenges faced during this project. Section 2B gives a detailed formulation of PSO with associated block diagrams and flow charts for clarity, Section 2C details a pseudocode implementation of parallelization, and a step by step discussion of software design.

A. Objectives and Technical Challenges

The overarching goal of achieving a performance increase on GPUs can be broken down into 3 primary components: finding an optimal block size(s), finding optimal tile size(s) for shared memory allocation, and minimizing the movement of data between the GPU and CPU. While reducing data transfers will almost universally increase performance, finding optimal block and tile sizes is considerably less straightforward. In applications that have little to no data reuse inside a kernel, using shared memory is a wasted resource given that no new reads from global memory are required. Furthermore, the time required to copy data from global memory to shared memory can significantly harm performance. Thus, an ideal combination must be found in order to ensure optimal performance.

B. Problem Formulation and Design

At the heart of Particle Swarm Optimization is an iterative update of the particle's velocity and position within each timestep. These are given in Eq. 1 and Eq 2.

$$v_{k+1}^i = \omega_k v_k^i + c_1 r_1 (p_k^i - x_k^i) + c_2 r_2 (p_k^g - x_k^i) \quad (1)$$

$$x_{k+1}^i = x_k^i + v_{k+1}^i \tag{2}$$

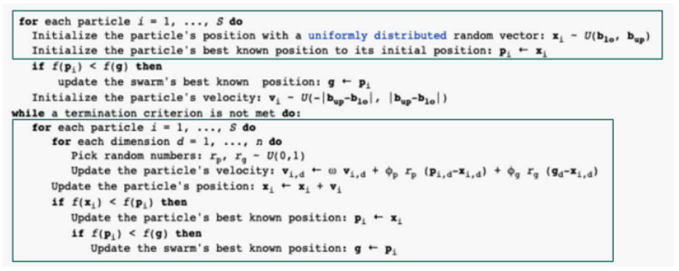
where:

- c_1, c_2, and ω are hyper-parameters that define the relative strength of each component of the velocity calculation
- r_1, r_2 are uniform random variables $U\{0, 1\}$
- x_k^i represents particle i's position at the kth timestep
- p_k^i represents particle i's best *visited* position at the kth timestep
- v_k^i represents particle i's velocity at the kth timestep
- p_k^g represents the global best position among all particles $i = 1, ..., n$ on the kth timestep

For a given cost function in $\mathbb{R}^n$, the velocity calculation above is done using a vector in $\mathbb{R}^{n-1}$.

It can be seen that Eq. 1 has three components in the velocity calculation. Respectively, these have been named the *inertia*, the *personal* factor, and the *societal* factor. The personal component tends to pull the particle's velocity toward it's *personal* best location visited. Similarly, the *societal* factor pulls every particle toward the best known position of the *group*.

A full pseudo-code implementation of the PSO can be seen in Figure 1, and a similar flow chart can be seen in Figure 2.



```
for each particle $i$ = 1, ..., $S$ do
  Initialize the particle's position with a uniformly distributed random vector: $\mathbf{x}_i \sim U(\mathbf{b}_{lo}, \mathbf{b}_{up})$
  Initialize the particle's best known position to its initial position: $\mathbf{p}_i \leftarrow \mathbf{x}_i$
  if $f(\mathbf{p}_i) < f(\mathbf{g})$ then
      update the swarm's best known  position: $\mathbf{g} \leftarrow \mathbf{p}_i$
  Initialize the particle's velocity: $\mathbf{v}_i \sim U(-|\mathbf{b}_{up}-\mathbf{b}_{lo}|, |\mathbf{b}_{up}-\mathbf{b}_{lo}|)$
while a termination criterion is not met do:
  for each particle $i$ = 1, ..., $S$ do
    for each dimension $d$ = 1, ..., $n$ do
      Pick random numbers: $r_p, r_g \sim U(0,1)$
      Update the particle's velocity: $\mathbf{v}_{i,d} \leftarrow \omega\, \mathbf{v}_{i,d} + \phi_p\, r_p\, (\mathbf{p}_{i,d}-\mathbf{x}_{i,d}) + \phi_g\, r_g\, (\mathbf{g}_d-\mathbf{x}_{i,d})$
    Update the particle's position: $\mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{v}_i$
    if $f(\mathbf{x}_i) < f(\mathbf{p}_i)$ then
      Update the particle's best known position: $\mathbf{p}_i \leftarrow \mathbf{x}_i$
      if $f(\mathbf{p}_i) < f(\mathbf{g})$ then
        Update the swarm's best known position: $\mathbf{g} \leftarrow \mathbf{p}_i$
```

Fig. 1. Pseudocode Implementation of PSO

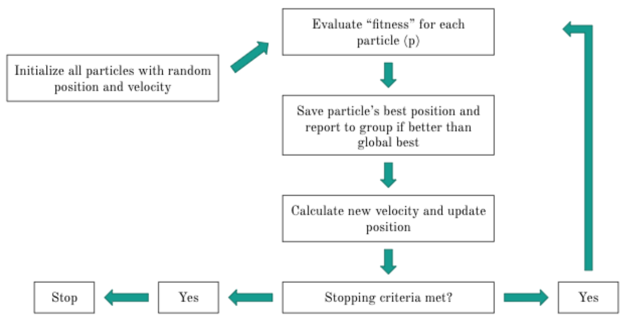

Initialize all particles with random position and velocity
Evaluate "fitness" for each particle (p)
Save particle's best position and report to group if better than global best
Calculate new velocity and update position
Stopping criteria met?
Yes
Stop
Yes

Fig. 2. PSO Implementation Flowchart

Additionally, the ability to parallelize PSO can be seen in Figure 1. It begins by initializing the position of each particle randomly, according to $U\{a, b\}$, where a and b can be chosen by the user as upper and lower bounds to the search space, however it is also common to start every particle at the same position. Serial algorithms require that each particle's initial position be calculated sequentially, however GPUs offer the ability to perform *all* initializations in parallel.

The most impactful parallelization comes from within each iteration (as seen by the second outline in Figure 1), and can be broken down into three sub-parallelization problems: the velocity calculation, the updating of the particle's best known position, and the update of the global best position. These three problems are discussed at length in the following section.

C. Software Design

In the broadest sense, a parallel PSO implementation can be seen in Figure 3, where $f(x)$ represents the three core parallelizable elements within each iteration. The following three sections will present a more in depth look at each of these elements.

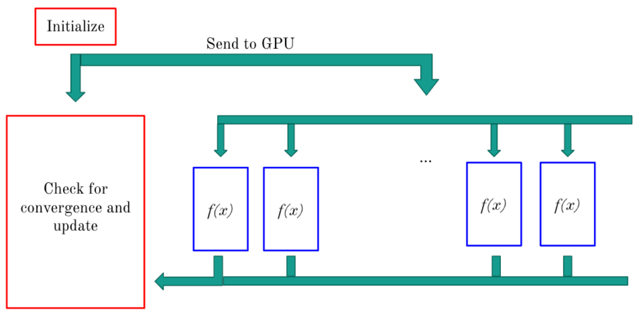


Fig. 3. PSO Block Diagram

1) Velocity Calculation:

The updated position and velocity calculations involve moving several arrays to the GPU. Following the notation of Eq. 1, these arrays have values x_k, p_k, and v_k. Data is stored in 2D arrays of size $(\#\ of\ particles, \#\ of\ dimensions - 1)$. For example, solving a 10-dimensional x^2 minimization with 30 particles will result in arrays of size $(30, 9)$. The parallelization of this velocity calculation follows the block diagram in Figure 4, and a more detailed pseudocode implementation can be seen in Figure 5.

Within this calculation, shared memory techniques were investigated in an attempt to improve efficiency by reducing the number of reads from global memory. PSO suffers from very little data reuse (only utilizing the current position twice per iteration as seen in Eq. 1), thus, as seen in the Results section, it benefits from little to no shared memory usage.

If the allocated thread falls within these input arrays, each particle *and* vector component of each particle computes updated positions and velocities in parallel.

2) Individual Evaluation:

Following Figures 1 and 2, after computing the updated positions and velocities, one must then evaluate every particle's new position, modifying p_k^i if the evaluation leads to less error than in the previous iteration. Figure 6 details

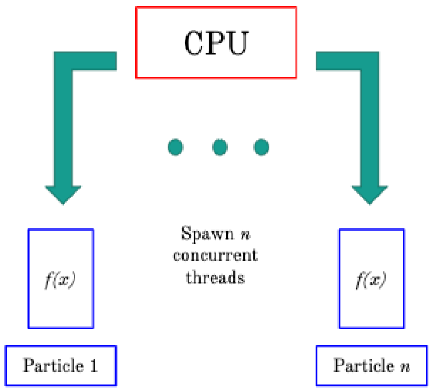
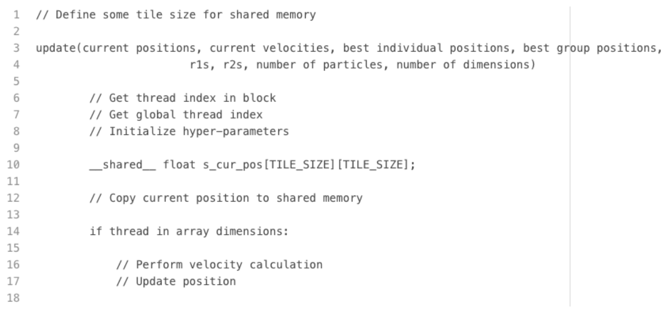


Fig. 4. Velocity Calculation Block Diagram

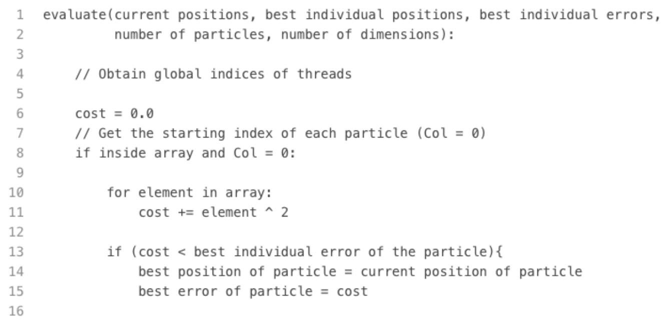


```
// Define some tile size for shared memory

update(current positions, current velocities, best individual positions, best group positions,
                       r1s, r2s, number of particles, number of dimensions)

        // Get thread index in block
        // Get global thread index
        // Initialize hyper-parameters

        __shared__ float s_cur_pos[TILE_SIZE][TILE_SIZE];

        // Copy current position to shared memory

        if thread in array dimensions:

            // Perform velocity calculation
            // Update position

```

Fig. 5. Velocity Calculation Pseudocode

pseudocode for this parallelizable element of PSO, and Figure 7 shows a block diagram of this element.

Of note, this aspect cannot be fully parallelized due to inherent race conditions that arise if multiple threads attempt to update a common value. This pseudocode implementation bypasses this with a for loop, however CUDA's $atomicAdd$ operation can be used in practice to remove this race condition.



```
evaluate(current positions, best individual positions, best individual errors,
         number of particles, number of dimensions):

    // Obtain global indices of threads

    cost = 0.0
    // Get the starting index of each particle (Col = 0)
    if inside array and Col = 0:

        for element in array:
            cost += element ^ 2

        if (cost < best individual error of the particle){
            best position of particle = current position of particle
            best error of particle = cost

```

Fig. 6. Evaluation Calculation Pseudocode

3) *Group Evaluation:*

The final parallelizable element comes when updating the group's best position, p_k^g, at the kth iteration. If, within the previous individual evaluation, a cost is observed that is less than the group's best known cost, it must be broadcasted to the entire group of particles and updated. Again, race

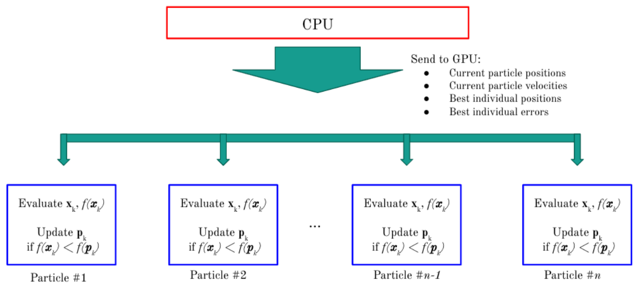
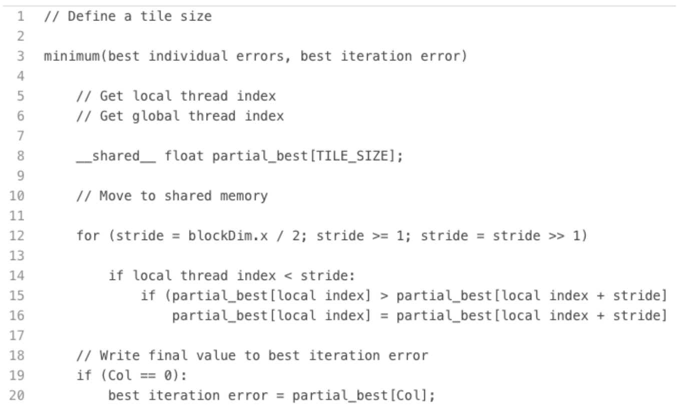

CPU
Send to GPU:
• Current particle positions
• Current particle velocities
• Best individual positions
• Best individual errors
Evaluate $\mathbf{x}_k$, $f(\boldsymbol{x}_k)$
Update $\mathbf{p}_k$ if $f(\boldsymbol{x}_k) < f(\boldsymbol{p}_k)$
Particle #1
Particle #2
...
Particle #n-1
Particle #n

Fig. 7. Evaluation Calculation Block Diagram

conditions may apply if multiple particles see smaller costs than the previous group's best cost. A reduction algorithm to find the minumim value in an array can be used to circumvent this race condition. Pseudocode for this reduction algorithm can be seen in Figure 8.



```
// Define a tile size

minimum(best individual errors, best iteration error)

    // Get local thread index
    // Get global thread index

    __shared__ float partial_best[TILE_SIZE];

    // Move to shared memory

    for (stride = blockDim.x / 2; stride >= 1; stride = stride >> 1)

        if local thread index < stride:
            if (partial_best[local index] > partial_best[local index + stride]
                partial_best[local index] = partial_best[local index + stride]

    // Write final value to best iteration error
    if (Col == 0):
        best iteration error = partial_best[Col];
```

Fig. 8. Group Comparison Pseudocode

III. Results

All code was run on the device in Table I using the PyCUDA [9] framework. This section will begin by presenting an overall view of PSO error minimization regardless of implementation decision. It will then show a baseline test of serial versus parallel performance. Following, tweaks to block size and shared memory are explored.

TABLE I
Device Specifications

Device Specifications	
Name	GeForce GTX TITAN X
Compute Capability	5.2
Multiprocessors	24
CUDA Cores	4608
Concurrent threads	49152
GPU Clock	1076 MHz
Memory Clock	3505 MHz
Total Memory	4016 MiB
Free Memory	3991 MiB

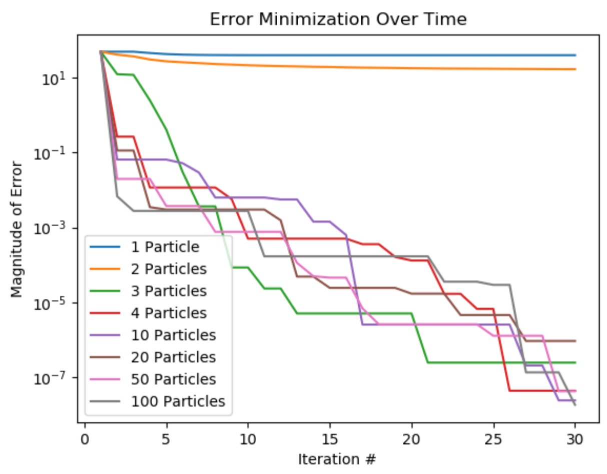


Fig. 9. PSO Accuracy

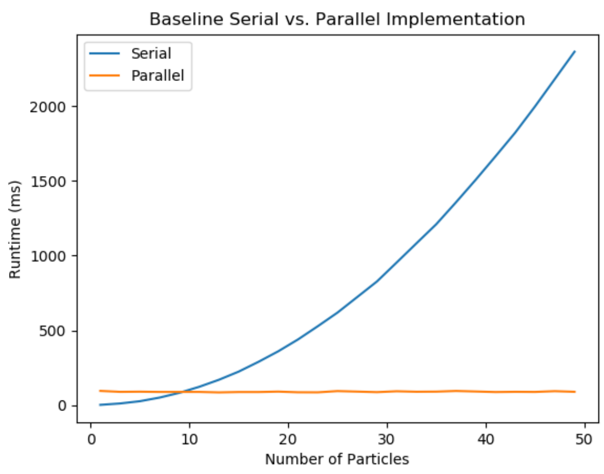


Fig. 10. Serial vs. Parallel Performance

A. *Error Minimization*

Figure 9 shows the accuracy of PSO while varying the number of particles and iterations in a minimization of $f(x) = x^2$ in a 9-dimensional search space. As can be seen, PSO performs poorly both for low iteration counts *and* low particle numbers. A single particle acts similarly to Brownian motion, and can easily get trapped in local extrema as shown in Figure 9. As particle numbers increase, accuracy increases significantly, however levels quickly after three particles. This can be attributed to the relatively simple cost function modeled in this simulation. In practice, cost functions will be much more complex, likely resulting in a wider disparity in performance as particle counts increase.

B. *Baseline Serial versus Parallel Implementations*

Figure 10 shows a comparison of serial versus parallel runtimes of PSO as a function of the complexity of the problem within a *single* iteration. As search spaces become more complex, the number of particles required to obtain worthwhile solutions increase, the number of dimensions increase, and the overall number of iterations required increases. This results in the quadratic scaling of computational resources as complexity increases. Traditional serial algorithms must sequentially compute updated velocities, best individual positions, and best group positions with respective runtimes of $O(n^2), O(n^2)$, and $O(n)$. The parallel algorithm, conversely, takes $O(1), O(logn)$, and $\sim O(n)$ respectively.

Thus the parallel implementation avoids this quadratic scaling of runtime within each iteration. Runtimes in Figure 10 include delays of copying data between the GPU and CPU, thus for particle numbers less than ten, the serial algorithm is more efficient. However, for practical applications, it's safe to assume the number of particles will be significantly past the 10-particle inflection point shown in Figure 10.

IV. Performance versus Block Size

Figure 11 details the effect of block size on performance for an increasing number of particles and dimensions within the code in Figure 6. Interestingly enough, no patterns emerge outside of an initial decrease in runtime as the number of particles and dimensions increase above one. This is likely due to the under-utilization of resources in the kernel at low particle counts. Thus, at least up to 100 particles and dimensions in a relatively simple cost function, block size seems to be independent of performance. Optimal block sizes may emerge for more complicated functions in higher dimensions, however the runtime limitations of the GPU in use prevented this exploration.

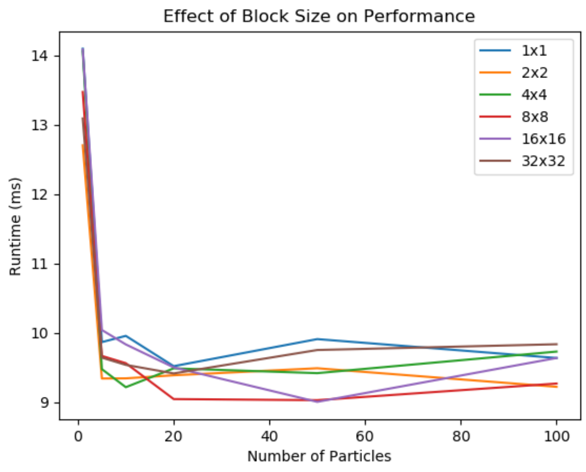


Fig. 11. Effect of Block Size on Performance

A. *Performance versus Tile Size*

Figure 12 details the effect of changing tile size in shared memory on performance for an increasing number of particles within the code detailed in Figure 5. It is apparent that for a higher number of particles, runtime performance suffers when larger tile sizes are used. This can be attributed to the fact that PSO has very little data reuse within each velocity calculation. Eq. 1 shows this fact, as x_k^i will be the only element used multiple times in the calculation. Thus, it is not worthwhile to spend the time copying data from global to shared memory in this case. As tile size increases, so do shared memory sizes within each block. For large tile sizes, it becomes more costly for the thread to access each element in shared memory than for smaller tile sizes.

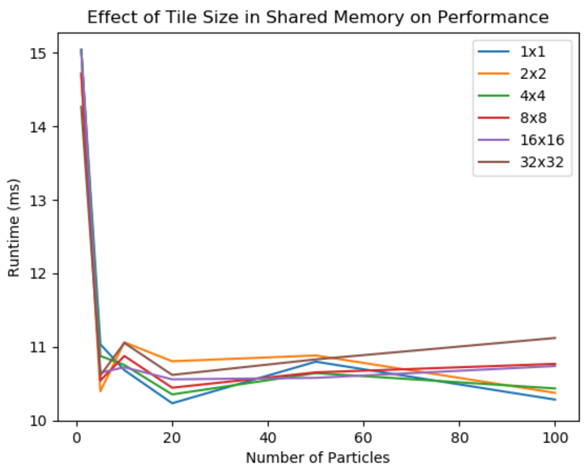


Fig. 12. Effect of Tile Size in Shared Memory on Performance

V. Discussion and Further Work

As shown in the Results section, the parallelization of Particle Swarm Optimization does yield a significant performance increase over the serial version. This performance increase seems on par with previous literature on the subject.

Performance measures do not incorporate *every* data copy from host to device, thus, in practice, one might need a larger number of particles before the parallel algorithm becomes more efficient. In the future, performing as many computations on the GPU is crucial in order to remove the significant performance cost of moving data between devices. Random numbers were generated on the CPU and then copied to the GPU. Moving such generation to the GPU would significantly reduce runtimes. Additionally, CUDA supports an atomicMin() operation, however it presently only works with integers. If at some point NVIDIA incorporates floating point support, this could be a much simpler approach to finding minimum array values as efficiently as possible.

To further decrease runtimes, a pure CUDA implementation could be used instead of PyCUDA. Moreover, several aspects of the problem were left in serial, including finding the index of the optimal array value in the calculation of the group's best position. Finding a more optimal approach to this problem will again significantly reduce runtime in practice.

VI. Conclusion

This project aimed to parallelize aspects of Particle Swarm Optimization (PSO), in an attempt to increase the viability of this algorithm in modern applications. By using a GPU instead of traditional CPUs, four aspects of PSO have been improved, reducing the runtime complexity from quadratic to linear with increasing particle numbers and dimensions. This project set out to understand how varying block size, utitlizing shared memory, and memory coalescing impact performance. Figures 11 and 12 detail this investigation, and show the negative impact of shared memory when data reuse is minimal. Further improvement can be made both by reducing the number of data transfers between the CPU and GPU, and by modifying the reduction kernel to obtain both the minimum value *and* index associated with that value.

VII. Acknowledgements

I would like to thank Professor Zoran Kostic for his time and effort throughout the class this semester. In addition, I'd like to thank Abhyuday Puri, the course TA, for his insights into several key aspects of this problem.

References

[1] An Introduction to Neural Networks. 1995. Gradient Descent Algorithms. doi:10.7551/mitpress/3905.003.0011.

[2] Putt Sakdhnagool and Amit Sabne and Rudolf Eigenmann (2019). RegDem: Increasing GPU Performance via Shared Memory Register Spilling CoRR, abs/1907.02894.

[3] Sze, V., Chen, Y.H., Yang, T.J., & Emer, J. (2017). Efficient processing of deep neural networks: A tutorial and survey Proceedings of the IEEE.

[4] Eberhart, Russell, and James Kennedy. "Particle swarm optimization." Proceedings of the IEEE international conference on neural networks. Vol. 4. 1995.

[5] Schutte, J. F., Reinbolt, J. A., Fregly, B. J., Haftka, R. T., & George, A. D. (2004). Parallel global optimization with the particle swarm algorithm. International journal for numerical methods in engineering, 61(13), 2296-2315.

[6] Zhou, Y., & Tan, Y. (2009, May). GPU-based parallel particle swarm optimization. In 2009 IEEE Congress on Evolutionary Computation (pp. 1493-1500). IEEE.

[7] Gao, Jianwei et al. Multi-GPU Based Parallel Design of the Ant Colony Optimization Algorithm for Endmember Extraction from Hyperspectral Images. Sensors (Basel, Switzerland) vol. 19,3 598. 31 Jan. 2019, doi:10.3390/s19030598

[8] Congsheng Li, Chang Liu, Lei Yang, Luyang He, and Tongning Wu, Particle Swarm Optimization for Positioning the Coil of Transcranial Magnetic Stimulation, BioMed Research International, vol. 2019, Article ID 9461018, 12 pages, 2019. https://doi.org/10.1155/2019/9461018.

[9] Klckner, Andreas, et al. "PyCUDA: GPU run-time code generation for high-performance computing." Arxiv preprint arXiv 911 (2009).

VIII. Individual Student Contributions

This project was completed individually, thus all contributions came from Austin Ebel, with the supplementary help of Professor Zoran Kostic and Abhyuday Puri.