Enhancing Particle Swarm Optimization Performance Through CUDA and Tree Reduction Algorithm

Hussein Younis, Mujahed Eleyat
Department of Computer Systems Engineering, Arab American University, Palestine

***Abstract*—In this paper, we present an enhancement for Particle Swarm Optimization performance by utilizing CUDA and a Tree Reduction Algorithm. PSO is a widely used metaheuristic algorithm that has been adapted into a CUDA version known as CPSO. The tree reduction algorithm is employed to efficiently compute the global best position. To evaluate our approach, we compared the speedup achieved by our CUDA version against the standard version of PSO, observing a maximum speedup of 37x. Additionally, we identified a linear relationship between the size of swarm particles and execution time; as the number of particles increases, so does computational load – highlighting the efficiency of parallel implementations in reducing execution time. Our proposed parallel PSOs have demonstrated significant reductions in execution time along with improvements in convergence speed and local optimization performance - particularly beneficial for solving large-scale problems with high computational loads.**

Keywords—Particle swarm optimization; tree reduction algorithm; parallel implementations; CUDA; GPU

I. Introduction

Optimization techniques are crucial in various domains for finding optimal solutions to complex problems. However, Particle Swarm Optimization, a widely used metaheuristic algorithm, has demonstrated limitations in terms of convergence speed and local optimization performance [1] [2]. As a result, researchers have turned to parallel computing techniques like Compute Unified Device Architecture (CUDA) a parallel computing platform and application programming interface (API) developed by NVIDIA, to enhance the performance of PSO by implementing it on a parallel architecture. Significant reductions in computing time compared to traditional implementations using different programming languages have been observed by researchers.

In the field of parallel computing, practitioners often employ various techniques to break down a computational task into smaller subtasks that can be executed simultaneously on multiple processors. These subtasks, commonly known as threads, are vital in this approach and are managed for execution by an operating system. CUDA supports shared memory parallel programming, which enables multiple processors or cores to access a shared memory space efficiently [3].

The integration of CUDA technology plays a pivotal role in enabling the seamless implementation of Particle Swarm Optimization (PSO) within a parallel architecture. This cutting-edge approach harnesses the power of GPUs to efficiently distribute workloads into smaller tasks, allowing for concurrent processing on the Graphics Processing Units. By utilizing CUDA for the parallel execution of PSO on GPUs, computational tasks benefit from enhanced efficiency and performance through the utilization of parallel processing capabilities, ultimately leading to accelerated computations and improved results in various applications such as optimization, machine learning, and scientific simulations [4].

This parallel method empowers each particle to autonomously execute a designated number of iterations before resynchronization occurs. Many researchers have successfully implemented PSO algorithms using CUDA for GPUs, and the outcomes from these endeavors unequivocally indicate that parallelization significantly enhances the performance capabilities of PSO [5].

This paper produces a CUDA version of the PSO algorithm called (CPSO). The tree reduction algorithm and the CUDA shared memory were used in CPSO to reduce the comparison operations to half and reduce the amount of time spent accessing global memory. The contributions of this work are summarized as follows:

1) Propose a CUDA version of the PSO algorithm.

2) Enhance the CUDA implementation of the PSO algorithm using the tree reduction algorithm and the CUDA shared memory.

3) Compare the proposed algorithms in terms of execution time and speedup to demonstrate the effectiveness and efficiency of our proposed algorithm.

The structure of this paper is outlined as follows: Section II presents the background information, Section III discusses related work, Section IV details the implementation of PSO algorithms, Section V outlines the experimental setup, Section VI presents the results and discussions, and Section VII provides the conclusions and suggestions for future work.

II. Background

A. Graphics Processing Unit

The Graphics Processing Unit (GPU) was initially developed in the 1970s as an electronic circuit for displaying vector graphics [6]. Over the years, the GPU has undergone significant development and has evolved into a highly parallel processor capable of performing complex computations, providing significant performance boosts for graphics-intensive applications [7]. One of the main advantages of the GPU is its ability to provide higher instruction throughput and memory bandwidth than the Central Processing Unit (CPU) within the same power envelope. Unlike the CPU which is designed to execute threads as fast as possible and execute only a few threads in parallel depending on the number of cores, the GPU is designed to execute thousands of sequences of operations, called “threads,” in parallel up to 1024 threads per block limit of the GPU [8]. The main architecture of the CPU and GPU is illustrated in Fig. 1, where the CPU (host) and GPU (device) work together and communicate via a PCI-express bus [9].

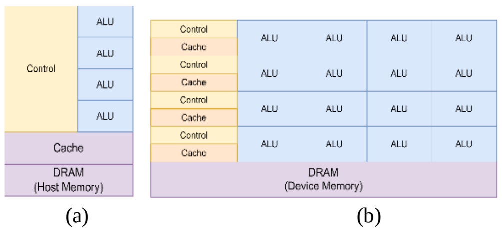


Fig. 1. The architecture of (a) CPU; and (b) GPU.

B. Compute Unified Device Architecture

NVIDIA, a leading player in the field of visual computing and parallel processing, introduced the CUDA in 2007 as a parallel platform programming model [10]. CUDA provides a set of tools that enable the development of high-performance applications to be executed on GPUs. Typically, a CUDA program consists of two parts: the first part is executed on the host CPU, while the second part is executed on the device GPU, with the result being returned to the host CPU [11]. As shown in Fig. 2, the CUDA programming architecture consists of N number of grids which depends on the limitations of the GPU hardware. Each grid is composed of blocks, and each block contains multiple threads that can be executed concurrently. The thread blocks are organized into 1D, 2D, or 3D arrays of threads [12].

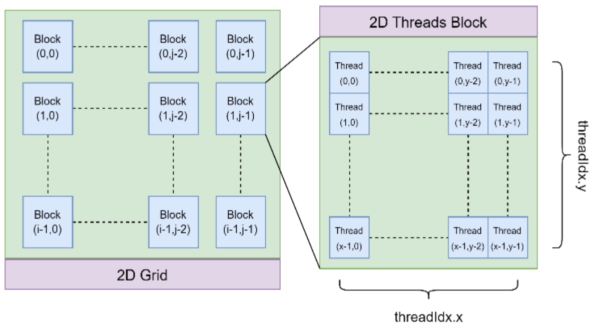


Fig. 2. CUDA 2D grid and thread block indexes presentation.

Each thread within a block has a specific index that is used to identify its location during the execution of the CUDA function, known as the 'kernel' function. The thread index for a 1D dimension is calculated using the following equation [13]:

$$thread\ index = (blockIDx.x \times blockDim.x) + threadIdx.x \quad (1)$$

Here, $blockIDx.x$ represents the x-dimension identifier of the thread block, $blockDim.x$ represents the x-dimension of the thread block, and $threadIdx.x$ represents the x-dimension identifier of the thread.

In GPUs, threads can be executed together in parallel in groups called "warps" which consist of 32 or 64 threads depending on the GPU architecture [14]. Within each warp, the threads execute the same instruction at the same time, a concept known as Single Instruction Multiple Threads (SIMT) which minimizes the amount of branching and divergence between threads which can result in performance penalties [15]. Each CUDA thread possesses its private local memory and can access data from multiple memory spaces. Furthermore, each block has shared memory that can be accessed by its threads or by other thread blocks as illustrated in Fig. 3. Local memory offers the fastest memory access speed for each thread, followed by shared memory. Global, static, and texture memory speeds are relatively slower [15].

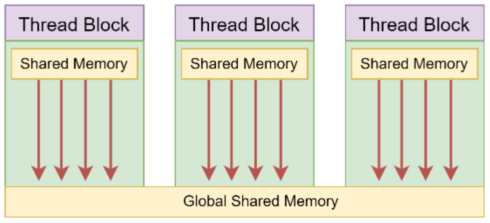


Fig. 3. Memory hierarchy in GPUs.

CUDA provides a function called “cudaMallocManaged” for unified memory management between CPU and GPU without explicit data transfers. It acts like a single memory space that can be accessed by both CPU and GPU. The overhead of explicit data transfer between CPU and GPU is reduced which improves the overall performance by providing a unified memory space [16]. Also, CUDA provides a function called “cudaMemPrefetchAsync” to prefetch data from the host or device memory to the device cache before it is needed. The main advantage of “cudaMemPrefetchAsync” is that data movement operation is performed asynchronously, optimizing memory access patterns and reducing data transfer latency in CUDA applications [17], [18]. Typically, a GPU program consists of one or more kernels which are collections of tasks executed sequentially by GPUs. These kernels are composed of blocks, separate groupings of Arithmetic Logic Units (ALUs). Each block contains multiple threads, representing various levels of computation. Usually, the threads within a block collaborate to calculate a specific value. It is important to note that threads within the same block can share memory, enabling efficient data interchange. In the context of CUDA, the most common computation involves transferring data from the CPU to the GPU [19] The main steps of the CUDA program flow, as depicted in Fig. 4, are: the data is loaded into the host CPU memory and then transferred to the GPU memory using a function called *"cudaMemcpy"*. Subsequently, the kernel is launched on the GPU using the syntax *"kernel<<<numBlocks, threadsPerBlock>>>"* [14]. The "<<<>>>" notation is employed to configure the execution of the kernel by specifying the number of thread blocks and the number of threads per block [20].

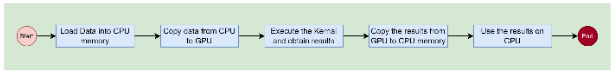

Start
Load Data into CPU memory
Copy data from CPU to GPU
Execute the Kernal and obtain results
Copy the results from GPU to CPU memory
Use the results on CPU
End

Fig. 4. GPU program workflow.

III. Related Work

PSO has been applied and extended in various studies. In this context, several works have focused on optimizing the performance of PSO algorithms, particularly by leveraging parallel computing techniques.

In study [21] the authors provide an overview of the PPSO algorithm which is commonly used in complex optimization problems requiring significant computational power. They discuss different parallelization options for PPSO, including programming languages and communication topologies. Also, they cover various models of parallelization, implementation, and uses of PPSO algorithms, making them a valuable resource for researchers and developers working with PPSO and other parallel optimization algorithms.

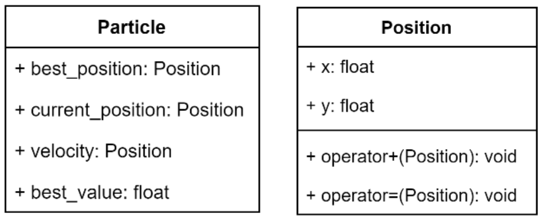
In study [22] the authors propose a novel algorithm called "cuPSO" that reduces the computation time of PSO-based algorithms with massive threads on GPUs. The proposed algorithm addresses excessive memory accesses and thread synchronization overheads faced by traditional reduction-based methods through the use of atomic functions. Experimental results show that cuPSO achieves over 200x speedups compared to the serial version running on the CPU and outperforms the state-of-the-art method by a factor of 2.2 in terms of computation time. Similarly, the authors focus on optimizing particle systems using CUDA-assisted multithreading. They aim to improve the performance of particle systems by enhancing a CUDA particle demo developed by Nvidia using a Python script. The experimental results in their work demonstrate the achievement of desired performance levels by adjusting the number of particles, grid size, and grid orientation. It also presents hypotheses regarding the impact of changing these parameters on processing time and provides experimental results to support these hypotheses [23]. Furthermore, another work introduces a new approach to running standard particle swarm optimization (SPSO) by utilizing GPU's parallel computing capability and NVIDIA's CUDA software platform. Experiments were conducted to optimize benchmark test functions using both GPU-SPSO and CPU-SPSO, results show that GPU-SPSO significantly reduces running time compared to CPU-SPSO more than 11 times faster than CPU-SPSO, especially for large swarm population applications and high dimensional problems [24].

The authors explore and evaluate two different ways of utilizing GPU parallelism in the implementation of particle swarm optimization (PSO) on graphics processing units (GPUs). The execution speed of these two parallel algorithms is compared with a standard sequential implementation of PSO, known as SPSO. The study also includes a comprehensive analysis of the computation efficiency of the parallel algorithms, considering speed-up and scale-up with SPSO. Also, the authors investigate the extent to which PSO can benefit from a parallel implementation using CUDA. The design of the two parallel versions of PSO considered in this study was influenced by the structure of CUDA and compatible GPUs. Additionally, the practical implications of the parallel algorithms resulted in two possible solutions that differentiate the potential use of each version [5].

Finally, another work introduces a parallel implementation of Cooperative Particle Swarm Optimization (CPSO) using CUDA. The work includes a comparison between CPSO implemented in C and C-CUDA, and tests were conducted on standard benchmark optimization functions. The results showed improvements in speed and convergence time, with CUDA's randomizing procedures contributing to better solutions. The paper emphasizes the utility of CUDA for complex and computationally intensive applications [25].

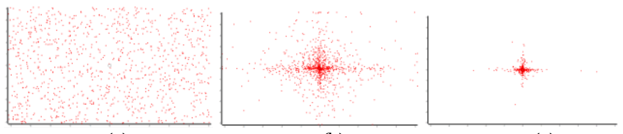
IV. PSO Algorithms Implementation

In general, the choice of data structure in the PSO algorithm is crucial for effectively representing and manipulating particles within the swarm. The main data structure used in proposed PSO algorithms is the Particle structure as illustrated in Fig. 5(a), which encapsulates the necessary information for each particle. This structure typically includes components such as the current position, best position, velocity, and best value. The current position represents the particle's location in the search space, while the best position stores the particle's personal best solution found so far. The velocity determines the particle's movement in the search space, and the best value represents the fitness or objective value associated with the best position. The struct position as illustrated in Fig. 6(b) represents a two-dimensional position in space, with x and y coordinates stored as floating-point values. It also includes two member functions and two overloaded operators. PSO algorithms can efficiently update and track the positions and velocities of particles, facilitating the exploration and exploitation of the search space by utilizing these data structures. The design and implementation of these data structures are critical for the success of PSO in finding optimal solutions to optimization problems.


Particle
+ best_position: Position
+ current_position: Position
+ velocity: Position
+ best_value: float
Position
+ x: float
+ y: float
+ operator+(Position): void
+ operator=(Position): void

(a) (b)

Fig. 5. Data structure for (a) The particle struct; and (b) The position struct.



(a) (b) (c)

Fig. 6. A 2D visualization for PSO with 5000 particles: (a) The initial state of particles; (b) The particle's state after 100 iterations; (c) The particle's state after 200 iterations.

A. *Standard PSO Algorithm*

The Standard Particle Swarm Optimization (SPSO) algorithm is one such technique that draws inspiration from the social behavior of bird flocking or fish schooling. It is categorized as a population-based optimization technique and was first introduced in 1995 by Kennedy and Eberhart. In

SPSO, each solution is referred to as a "particle" that moves through the search space, seeking the optimal position as illustrated in Fig. 6. The search for the optimal position is guided by a "fitness function." Each particle has its position and velocity which are adjusted in each iteration based on its experience and the collaboration with its neighbors in the search space [26], [27].

This collaboration is demonstrated by the following equations [28]:

$$v_i{}^n = \omega v_i{}^n + c_1 r_1 (pbest_i^n - x_i{}^n) + c_2 r_2 (gbest^n - x_i{}^n) \quad (2)$$

$$x_i{}^{n+1} = x_i{}^n + v_i{}^{n+1} \quad (3)$$

where, the $v_i{}^n$ and $x_i{}^n$ present the current velocity and the position of the particle i at the nth iteration respectively, the ω presents the inertia weight, the c_1 and c_2 present the cognitive and social coefficients, respectively, the r_1 and r_2 are random numbers in the range $[0,1]$, the $pbest_i^n$ represents the best position of the particle i in the nth iteration and the $gbest^n$ represents the best position among all particles at the nth iteration.

Algorithm 1 demonstrates the pseudocode of the SPSO algorithm, with the following description of the SPSO parameters [26]:

- Population: It presents the total number of particles in the swarm space.
- T_{max} present the maximum number of iterations.
- x_i and v_i present the current position and the velocity, respectively, for the particle p_i.
- italicsfitness$_i$ and pbest_fitness$_i$: present the fitness value and the best fitness value, respectively, for the particle p_i.
- $pbest_i$ presents the best position for the particle p_i.
- gbest and gbest_fitness present the team best position and best fitness value, respectively, of the entire swarm space.
- Termination condition presents the criteria that determine when the SPSO will stop searching for the optimal solution.

Algorithm 1 Sequential SPSO algorithm

```
For every particle $p_i$ in the swarm space, where $0 \le i <$ population do:
    Initialize the $x_i$ and $v_i$ randomly
    Evaluate the fitness$_i$ by the $x_i$ using the fitness function.
    Initialize the pbest_fitness$_i$ and $pbest_i$.
    Update the $gbest$ and the $gbest\_fitness$.
End
For every iteration $t = 0,1,2,\dots,T_{max}$, do:
    For every particle $p_i$ in the swarm space, where $0 \le i <$ population do:
        Update the position $x_i$ and the velocity $v_i$ for particle $p_i$ by the Eq. (4) and (5).
        Evaluate the new fitness$_i$ by the $x_i$ using the fitness function.
        If the new fitness$_i$ > pbest_fitness$_i$ then update pbest_fitness$_i$ by new fitness$_i$ and $pbest_i$ by $x_i$.
        If the new fitness$_i$ > $gbest\_fitness$ then update $gbest\_fitness$ by new fitness$_i$ and $gbest$ by $pbest_i$.
        If the $gbest\_fitness$ met the termination condition, then exit from main and secondary loops.
    End
End
```

The time complexity of the SPSO algorithm is typically$O(T \times P)$, where the T is the number of iterations and the P is the number of particles in the swarm space.

B. CUDA PSO Algorithm

The SPSO algorithm is one such technique to leverage the power of parallel computing of GPU, to introduce the CUDA Particle Swarm Optimization (CPSO). The CPSO involves parallelizing by assigning each particle to a separate thread on the GPU to update its position and velocity based on its own best position ($pbest$) and the best position ($gbest$) founded by any particle in the swarm space. Also, the unified memory management and shared memory within each block have been utilized since the SPSO is a memory-bound problem. The CPSO consists of three kernels update particle velocity, update particle position, and compute the best position ($gbest$). The pseudocode of the CPSO is demonstrated in Algorithm 2.

Algorithm 2 CUDA PSO Algorithm (CPSO)

```
Set the blockSize equal to 32 and determine the grid size by the Eq. (3).
Allocate memory for particles, $gbest$ and the $gbest\_fitness$ using cudaMallocManaged.
Initialize the particles with random starting positions, velocities, and $pbest$.
compute $gbest$ and the $gbest\_fitness$ using ComputeGlobalBestPosition kernel.
Prefetch the particles array to the GPU.
For every iteration $t = 0,1,2,\dots,T_{max}$, do:
    Update the velocity of each particle using the kernel updateParticleVelocity.
    Update the position of each particle using the kernel updateParticlePosition.
    Update the $gbest$ and the $gbest\_fitness$ using the kernel ComputeGlobalBestPosition.
    If the $gbest\_fitness$ met the termination condition, then terminate.
    End
End
Wait for GPU to finish the computation.
Free allocated memory.
```

The kernel "ComputeGlobalBestPosition" is implemented by applying the "tree" reduction algorithm where each block calculates its bbest within its shared memory where the bbest presents the best position within each block. This means that each block independently determines the bbest among the particles it is responsible for. The resulting minimum bbest is stored in shared memory and then reduced across all blocks to find the overall minimum value to obtain gbest. The following Fig. 7 illustrates a chart that shows how this works for a block of eight threads.

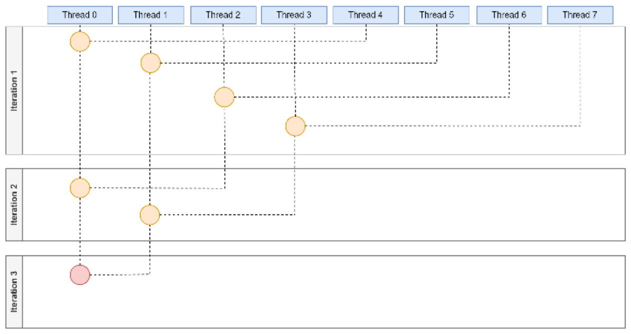


Fig. 7. Tree reduction algorithm workflow for eight threads.

For the first iteration, Thread 0 compares the best fitness value at index 0 with value at index 4, Thread 1 compares value at index 1 with value at index 5, Thread 2 compares value at index 2 with value at index 6, Thread 3 compares value at index 3 with value at index 7 and Threads 4-7 do nothing. For the second iteration, Thread 0 compares value at index 0 with value at index 2, Thread 1 compares value at index 1 with value at index 3 and Threads 2-3 do nothing. For the third iteration, Thread 0 compares value at index 0 with value at index 1 to obtain the final minimum value.

In each iteration of the loop, the number of threads that perform a comparison is halved. This means that the number of iterations required to reduce all values to a single minimum value is $log_2(N)$, where N is the number of threads in the block. After the parallel reduction loop completes, each thread block has found its own minimum value and corresponding index. These values are stored in shared memory. The final step is to reduce across all thread blocks to find the overall minimum value and corresponding index. This is done on the CPU after all threads have completed their computations.

The pseudocode of the "ComputeGlobalBestPosition" is illustrated in Algorithm 3.

Algorithm 3 *ComputeGlobalBestPosition kernel*

```
Declares a shared memory array blockBestValueArray with a
size of 32 that will be used for the parallel reduction within
each block.
Compute the thread ID within the block $t_{id}$ for the current
thread by threadIdx.x and the idx for the current thread by Eq.
(1). where $idx$ is the global index of the particle that this
thread is responsible for.
Initialize $blockBestValueArray[t_{id}]$ with $pbest$ value of
$idx's$ particle.
Synchronize all threads within the block using __syncthreads
() to ensure that all threads have finished updating the shared
memory variables.
loop i= $\frac{blockDim.x}{2}$, $i < 0, i >>= 1$ do the following:
    If the $t_{id} < i$ then
        If $blockBestValueArray[t_{id}] >$
        $blockBestValueArray[t_{id} + i]$, then update
        $blockBestValueArray[t_{id}]$ with
        $blockBestValueArray[t_{id} + i]$.
        End
    End
End
Synchronize all threads within the block using __syncthreads
() to ensure that all threads have finished updating the shared
memory variables.
If the $t_{id} = 0$ then
    Update the $gbest$ value by the first element of
    blockBestValueArray.
End
```

The kernel "updateParticleVelocity" is implemented where each thread is responsible for a particle update their velocity based on Eq. (4). The pseudocode of the "updateParticleVelocity" is illustrated in Algorithm 4.

Algorithm 4 *updateParticleVelocity kernel*

```
Compute the idx for the current thread by Eq. (1) where $idx$ is
the global index of the particle that this thread is responsible
for.
Declares a shared memory variable $gbest$ to access by all
threads within each block.
Synchronize all threads within the block using __syncthreads
() to ensure that all threads have finished loading the shared
memory variable.
If the $idx$ < population then
    Update the velocity for particle $p_{idx}$ by the Eq. (4).
End
```

The kernel "updateParticlePosition" is implemented where each thread is responsible for a particle updating its position based on Eq. (5) and updating its pbest. The pseudocode of the "updateParticlePosition" is illustrated in Algorithm 5.

Algorithm 5 *updateParticlePosition kernel*

```
Compute the idx for the current thread by Eq. (1) where $idx$ is
the global index of the particle that this thread is responsible
for.
If the $idx$ < population then
    Update the position for particle $p_{idx}$ by the Eq. (5).
    Evaluate the new $fitness_{idx}$ by the $x_{idx}$ using the fitness
    function.
    If the new $fitness_{idx}$ > $pbest\_fitness_{idx}$ then update
    $pbest\_fitness_{idx}$ by new $fitness_{idx}$ and $pbest_{idx}$ by
    $x_{idx}$.
End
```

V. Experimental Setup

The software and hardware specifications for the computer used to implement and test the PSO and CPSO algorithms are listed in Table I and Table II respectively. The details specification of the Graphics Card is listed in Table III.

TABLE I. SOFTWARE SPECIFICATION

Name	Version
Microsoft Windows 11	22H2
Visual Studio	2022
Nsight Systems	2023.2.3
CUDA	12020
OpenMP	2.0

TABLE II. HARDWARE SPECIFICATION

Specification	Properties
Processor	AMD Ryzen 9 5900HX, 3301 MHz, 8 Core(s), 16 Logical Processor(s)
Physical Memory (RAM)	32.0 GB
Graphics Card	NVIDIA GeForce RTX 3080 Laptop GPU

TABLE III. THE DETAILS SPECIFICATIONS OF THE GRAPHICS CARD

Specification	Properties
Global memory	16,383 MB
Shared memory	48 kb
Block registers	65,536
Max threads per block	1024
Max dimensions of a block	(1024, 1024, 64)
Max dimensions of a grid	$(2^{31} - 1, 65535, 65535)$
Warp size	32 threads
CUDA core	6,144 cores
Memory bandwidth	760.3 GB/sec
Memory channels	8
memory bus width	256-bit
Memory clock	1750 MHz

An important design parameter of the PSO algorithm is the fitness function. We choose the Euclidean distance function as the fitness function for all the experiments, as shown in Eq. (4). The parameter w used by the fitness function is set as 1 and learning factor $c1$ and $c2$ as 2, which are commonly seen settings [29].

$$f(x,y) = \sqrt{x^2 + y^2} \quad (4)$$

VI. Results and Discussion

The experiments were conducted ten times per particle number to calculate the minimum, maximum, and average execution time to ensure the reliability of results. Also, the median execution time and standard deviation were calculated. Table IV provides execution time data of the SPSO algorithm on CPU for different numbers of particles. The data shows that as the number of particles increases, the median execution time also increases. For example, the median execution time for 32 particles is 328 (ms), while the median execution time for 65,536 particles is 793,717 (ms). This indicates that the SPSO algorithm on the CPU becomes slower as the number of particles increases. In addition, the standard deviation also increases as the number of particles increases. This indicates that there is more variation in the execution times for larger numbers of particles which may be due to increased memory usage and/or contention for system resources.

TABLE IV. SPSO ALGORITHM EXECUTION TIME ANALYSIS

Particles	Iteration	SPSO CPU				
		Execution Time (MS)			Median Execution Time (MS)	Standard Deviation (MS)
		MIN	**MAX**	**AVG**		
32	100,000	309	387	335.90	328	23.31
64	100,000	604	693	634.30	622.50	28.35
128	100,000	1,222	1,401	1,259.30	1,243	48.38
256	100,000	2,676	2,813	2,735.10	2,731.50	40.05
512	100,000	5,378	5,664	5,489.90	5,482	81.48
1,024	100,000	11,189	12,237	11,496.10	11,360.50	309.38
2,048	100,000	24,548	25,640	4,865.50	24,812	296.38
4,096	100,000	45,907	47,328	46,369.80	46,248.50	387.45
8,192	100,000	101,203	103,771	102,598	102,670	703.28
16,384	100,000	199,465	201,425	200,503	200,560	796.56
32,768	100,000	93,643	402,628	398,791	399,447	3,717.68
65,536	100,000	788,879	798,531	793,711	793,717	4,784.19

TABLE V. CPSO ALGORITHM EXECUTION TIME ANALYSIS

Particles	Iteration	CPSO CPU				
		Execution Time (MS)			Median Execution Time (MS)	Standard Deviation (MS)
		MIN	**MAX**	**AVG**		
32	100,000	5,405	5,432	5,422.33	5,430	15.04
64	100,000	5,478	5,617	5,537	5,516	71.84
128	100,000	5,703	5,823	5,752.33	5,731	62.78
256	100,000	5,783	5,902	5,824	5,787	67.58
512	100,000	5,769	5,882	5,829.67	5,838	56.96
1,024	100,000	5,797	5,893	5,841.33	5,834	48.42
2,048	100,000	5,806	5,939	5,864	5,847	68.11
4,096	100,000	5,931	6,027	5,988	6,006	50.47
8,192	100,000	6,179	6,268	6,221	6,216	44.71
16,384	100,000	6,828	6,904	6,869.67	6,877	38.53
32,768	100,000	13,121	13,212	13,180.33	13,208	51.42
65,536	100,000	21,382	21,471	21,431	21,440	45.18

Table V shows the execution time data for the CPSO algorithm on a CUDA-enabled GPU for different numbers of particles. The data indicates that as the number of particles increased, the average and median execution times also increased. The minimum and maximum execution times were also increased, with the highest execution time being 21,471 ms for 65,536 particles and 100,000 iterations. However, the standard deviation shows that there is relatively little variation in the execution times across the different numbers of particles.

As shown in Fig. 8 the execution times for SPSO increase significantly as the number of particles increases. For example, the execution time for 65,536 particles is 793,711 (ms). The execution times for CPSO are much lower than SPSO and show much more consistent execution times across different numbers of particles. For example, the execution time for 65,536 particles is 21,431 (ms), and the percentage of the decrease in execution time is approximately 97.308% by CPSO compared to SPSO.

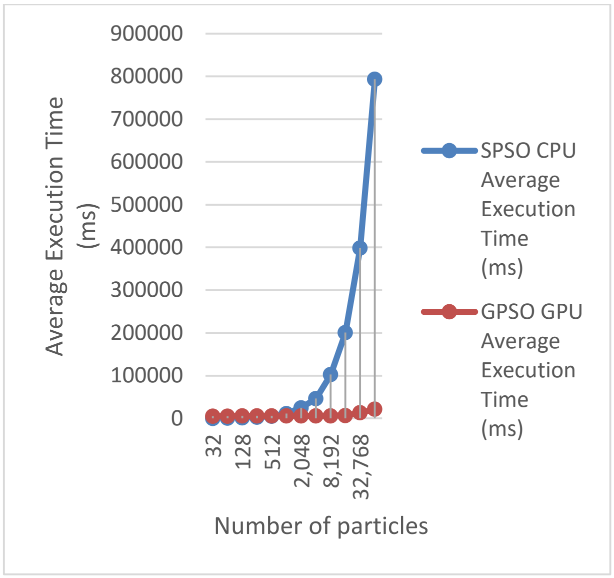


Fig. 8. Comparison of average execution time (ms) between SPSO on CPU and GPSO on GPU with respect to number of particles.

To compare the performance of these implementations, we calculated the speedup for CPSO relative to SPSO using the execution time for SPSO as a baseline. The speedup for CPSO was calculated using the following formula [30]:

$$Speedup = \frac{Execution\ Time(SPSO)}{Execution\ Time(Parallel\ Implementation)} \quad (5)$$

Table VI shows the speedup for CPSO relative to SPSO using the execution time for SPSO as a baseline. Based on the provided data for SPSO and CPSO with different particle counts and iterations, here is a summary of the key findings:

1) *Execution time trends:* The execution time increases as the number of particles and iterations increase. Also, for both Standard PSO (SPSO) and Constricted PSO (CPSO), the execution time generally follows an increasing trend with higher particle counts.

2) *Speedup comparison:* The speedup values for SPSO and CPSO range from 0.1 to 37.0 across different particle counts. These values indicate the parallelization efficiency, with higher values indicating better performance improvement with parallel processing.

3) *Comparison between SPSO and CPSO:* In most cases, CPSO shows lower execution times compared to SPSO for the same particle count and number of iterations. This difference suggests that the constriction factor used in CPSO may contribute to faster convergence and better optimization performance.

4) *Impact of particle count:* Increasing the number of particles has a significant impact on execution time, with higher particle counts leading to longer execution times. The data shows a clear trend of increasing execution time as the number of particles grows exponentially.

5) *Optimal performance considerations:* The choice between SPSO and CPSO should be based on the specific optimization problem and the desired trade-off between execution time and convergence speed. It is essential to consider the balance between speedup, execution time, and convergence efficiency when selecting the appropriate PSO variant.

TABLE VI. THE SPEEDUP FOR CPSO RELATIVE TO SPSO

Particles	**Iteration**	**Execution Time (MS)**		**speedup**
		SPSO	**CPSO**	
32	100,000	335.90	5,422.33	0.1
64	100,000	634.30	5,537	0.1
128	100,000	1,259.30	5,752.33	0.2
256	100,000	2,735.10	5,824	0.5
512	100,000	5,489.90	5,829.67	0.9
1,024	100,000	11,496.10	5,841.33	2.0
2,048	100,000	4,865.50	5,864	4.2
4,096	100,000	46,369.80	5,988	7.7
8,192	100,000	102,598	6,221	16.5
16,384	100,000	200,503	6,869.67	29.2
32,768	100,000	398,791	13,180.33	30.3
65,536	100,000	793,711	21,431	37.0

VII. CONCLUSIONS

In this work, we propose a CUDA version of the standard PSO algorithm to shorten the execution time for solving the PSO problem. We have shown the key ideas of the parallelizing algorithms for CUDA. Many experiments were conducted to improve the execution efficiency of the proposed algorithm by leveraging the power of parallel computing of GPU with the tree reduction algorithm, the CPSO achieving 37x speedup compared with the serial version SPSO and outperforming SPSO in terms of speedup. The result obtained shows that the relationship between swarm particle size and execution time is linear as the number of particles increased, the computational load also increased, making parallel implementations more effective at reducing the execution time. However, CPSO performed faster than SPSO for a high dimensional population, there was no significant improvement for the small number of particles. So, CPSO can especially benefit from optimizing with a large swarm population. For future work, further optimization and fine-tuning of the CPSO algorithm could be explored to enhance its performance with smaller particle numbers. Additionally, investigating the scalability and adaptability of the proposed CUDA-based PSO algorithm to handle even larger swarm populations and more complex optimization problems would be a valuable direction for future research. Integration with advanced parallel computing techniques and exploring hybrid approaches could also be considered to push the boundaries of speed and efficiency in solving PSO problems.

REFERENCES

[1] K. Zheng, X. Yuan, Q. Xu, L. Dong, B. Yan, and K. Chen, “Hybrid particle swarm optimizer with fitness-distance balance and individual self-exploitation strategies for numerical optimization problems,” Inf Sci (N Y), vol. 608, 2022, doi: 10.1016/j.ins.2022.06.059.

[2] R. M. Hou, Y. L. Hou, C. Wang, Q. Gao, and H. Sun, “A Hybrid Wavelet Fuzzy Neural Network and Switching Particle Swarm Optimization Algorithm for AC Servo System,” Math Probl Eng, vol. 2016, 2016, doi: 10.1155/2016/9724917.

[3] C. T. Yang, C. L. Huang, and C. F. Lin, “Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters,” in Computer Physics Communications, 2011. doi: 10.1016/j.cpc.2010.06.035.

[4] T. Kovac, T. Haber, F. Van Reeth, and N. Hens, “Heterogeneous computing for epidemiological model fitting and simulation,” BMC Bioinformatics, vol. 19, no. 1, 2018, doi: 10.1186/s12859-018-2108-3.

[5] L. Mussi, F. Daolio, and S. Cagnoni, “Evaluation of parallel particle swarm optimization algorithms within the CUDATM architecture,” Inf Sci (N Y), vol. 181, no. 20, 2011, doi: 10.1016/j.ins.2010.08.045.

[6] J. Peddie, The History of the GPU - New Developments. 2023. doi: 10.1007/978-3-031-14047-1.

[7] P. K. Das and G. C. Deka, “History and Evolution of GPU Architecture,” 2015. doi: 10.4018/978-1-4666-8853-7.ch006.

[8] T. M. Aamodt, W. W. L. Fung, and T. G. Rogers, “General-Purpose Graphics Processor Architectures,” Synthesis Lectures on Computer Architecture, vol. 13, no. 2, 2018, doi: 10.2200/S00848ED1V01Y201804CAC044.

[9] M. Adnan, P. A. Longley, A. D. Singleton, and I. Turton, “Parallel Computing in Geography,” in GeoComputation, Second Edition, 2014. doi: 10.1201/b17091-10.

[10] R. Ansorge, “A Brief History of CUDA,” in Programming in Parallel with CUDA, 2022. doi: 10.1017/9781108855273.013.

[11] M. Harris and I. Gelado, “More on CUDA and graphics processing unit computing,” in Programming Massively Parallel Processors: A Hands-on Approach: Third Edition, 2017. doi: 10.1016/B978-0-12-811986-0.00020-0.

[12] I. Gelado and M. Harris, “Advanced practices and future evolution,” in Programming Massively Parallel Processors: a Hands-on Approach, Fourth Edition, 2022. doi: 10.1016/B978-0-323-91231-0.00013-6.

[13] W. mei W. Hwu, D. B. Kirk, and I. El Hajj, “Multidimensional grids and data,” in Programming Massively Parallel Processors: a Hands-on Approach, Fourth Edition, 2022. doi: 10.1016/B978-0-323-91231-0.00004-5.

[14] “CUDA C++ Programming Guide.” Accessed: Aug. 08, 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[15] Nvidia, “Nvidia Cuda Getting Started Guide,” NVIDIA Corporation, no. July, 2013.

[16] M. Knap and P. Czarnul, “Performance evaluation of Unified Memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs,” Journal of Supercomputing, vol. 75, no. 11, 2019, doi: 10.1007/s11227-019-02966-8.

[17] L. Gebraad and A. Fichtner, “Seamless GPU Acceleration for C++-Based Physics with the Metal Shading Language on Apple’s M Series Unified Chips,” Seismological Research Letters, vol. 94, no. 3, 2023, doi: 10.1785/0220220241.

[18] S. Lee and J. S. Vetter, “OpenARC: Extensible OpenACC compiler framework for directive-based accelerator programming study,” in Proceedings of WACCPD 2014: 1st Workshop on Accelerator Programming Using Directives - Held in Conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Storage and Analysis, 2014. doi: 10.1109/WACCPD.2014.7.

[19] P. Xu, M. Y. Sun, Y. J. Gao, T. J. Du, J. M. Hu, and J. J. Zhang, “Influence of data amount, data type and implementation packages in GPU coding,” Array, vol. 16, 2022, doi: 10.1016/j.array.2022.100261.

[20] S. Cho, J. Choi, J. Hong, and H. Han, “Multithreaded double queuing for balanced CPU-GPU memory copying,” in Proceedings of the ACM Symposium on Applied Computing, 2019. doi: 10.1145/3297280.3297426.

[21] W. H. Mahdi and N. Taspiner, “Overview for Parallel Particle Swarm Optimization Algorithms (PPSO),” in 2022 14th International Conference on Electronics, Computers and Artificial Intelligence, ECAI 2022, 2022. doi: 10.1109/ECAI54874.2022.9847459.

[22] C. C. Wang, C. Y. Ho, C. H. Tu, and S. H. Hung, “cuPSO: GPU Parallelization for Particle Swarm Optimization Algorithms,” in Proceedings of the ACM Symposium on Applied Computing, 2022. doi: 10.1145/3477314.3507142.

[23] F. N. Sibai, A. Potvin, and S. Ngo, “Optimizing particle systems through CUDA-assisted multithreading,” WSEAS Transactions on Systems and Control, vol. 15, 2020, doi: 10.37394/23203.2020.15.69.

[24] Y. Zhou and Y. Tan, “GPU-based parallel particle swarm optimization,” in 2009 IEEE Congress on Evolutionary Computation, CEC 2009, 2009. doi: 10.1109/CEC.2009.4983119.

[25] J. Kumar, L. Singh, and S. Paul, “GPU based parallel cooperative particle swarm optimization using C-CUDA: A case study,” in IEEE International Conference on Fuzzy Systems, 2013. doi: 10.1109/FUZZ-IEEE.2013.6622514.

[26] S. Sengupta, S. Basak, and R. A. Peters, “Particle Swarm Optimization: A Survey of Historical and Recent Developments with Hybridization Perspectives,” Machine Learning and Knowledge Extraction, vol. 1, no. 1. 2019. doi: 10.3390/make1010010.

[27] D. Sedighizadeh and E. Masehian, “Particle Swarm Optimization Methods, Taxonomy and Applications,” International Journal of Computer Theory and Engineering, 2009, doi: 10.7763/ijcte.2009.v1.80.

[28] A. Khare and S. Rangnekar, “A review of particle swarm optimization and its applications in Solar Photovoltaic system,” Applied Soft Computing Journal, vol. 13, no. 5. 2013. doi: 10.1016/j.asoc.2012.11.033.

[29] X. Li, “A multimodal particle swarm optimizer based on fitness Euclidean-distance ratio,” in Proceedings of GECCO 2007: Genetic and Evolutionary Computation Conference, 2007. doi: 10.1145/1276958.1276970.

[30] P. Zhang, Y. Li, Y. Li, G. Chen, W. Hua, and Z. Jiao, “Research on calculation of surface irradiance for infrared extended sources based on CUDA parallel speedup,” Opt Express, vol. 30, no. 19, 2022, doi: 10.1364/oe.470137.