DOI: 10.6977/IJoSI.202510_9(5).0003
W. Zhu, S-K. Huang, etc./Int. J. Systematic Innovation, 9(5), 23-42 (2025)

A graphics processing unit-based parallel simplified swarm optimization algorithm for enhanced performance and precision

Wenbo Zhu[1], Shang-Ke Huang[2], Wei-Chang Yeh[2,3]*, Zhenyao Liu[4], Chia-Ling Huang[5]

[1]School of Mechatronical Engineering and Automation, Foshan University, Foshan, Guangdong, China

[2]Integration and Collaboration Laboratory, Department of Industrial Engineering and Engineering Management, College of Engineering, National Tsing Hua University, Hsinchu, Taiwan

[3]Department of Industrial and Systems Engineering, College of Electrical Engineering and Computer Science, Chung Yuan Christian University, Taoyuan, Taiwan

[4]School of Economics and Management, Taizhou University, Taizhou, Jiangsu, China

[5]Department of International Logistics and Transportation Management, School of Transportation and Tourism, Kainan University, Taoyuan, Taiwan

*Corresponding author E-mail: wcyeh@ie.nthu.edu.tw

(Received 13 January 2025; Final version received 02 August 2025; Accepted 04 August 2025)

Abstract

Graphics processing units (GPUs) have emerged as powerful platforms for parallel computing, enabling personal computers to solve complex optimization tasks effectively. Although swarm intelligence algorithms naturally lend themselves to parallelization, a GPU-based implementation of the simplified swarm optimization (SSO) algorithm has not been reported in the literature. This paper introduces a compute CUDA-SSO algorithm on the CUDA platform, with a time complexity analysis of O (Ngen × Nsol × Nvar), where Ngen is the number of iterations, Nsol is the population size (i.e., number of fitness function evaluations), and Nvar represents the required pairwise comparisons. By eliminating resource preemption of personal best and global best updates, CUDA-SSO significantly reduces the overall complexity and prevents concurrency conflicts. Numerical experiments demonstrate that the proposed approach achieves an order-of-magnitude improvement in run time with superior solution precision relative to central processing unit-based SSO, making it a compelling methodology for large-scale, data-parallel optimization tasks.

Keywords: Compute Unified Device Architecture, Graphics Processing Unit, Parallelism, Simplified Swarm Optimization, Swarm Intelligence Algorithms

1. Introduction

In recent years, graphics processing units (GPUs) have significantly impacted high-performance computing, particularly for data- and compute-intensive applications. Originally designed to accelerate real-time three-dimensional graphics, GPUs now offer a parallel architecture that can handle massive throughput in general-purpose scientific computing. Thanks to the availability of thousands of arithmetic logic units (ALUs) and large memory bandwidth, personal computers equipped with modern GPUs have become highly effective platforms for performing large-scale computations (AlZubi et al., 2020; Hachaj & Piekarczyk, 2023). This evolution has fueled a surge of interest in GPU-accelerated algorithms across diverse fields, including medical image processing (Corral et al., 2024; Mittal & Vetter, 2014), energy optimization (Mortezazadeh et al., 2022), and geospatial modeling (Hager et al., 2008).

One notable class of algorithms that can benefit significantly from the massive parallelism of GPUs is swarm intelligence (SI). Swarm intelligence algorithms (SIAs), such as particle swarm optimization (PSO), genetic algorithms (GA), and fireworks algorithms,

draw inspiration from natural phenomena (e.g., bird flocking, fish schooling, and evolutionary processes). By orchestrating collective behaviors, these methods iteratively refine candidate solutions within a high-dimensional search space (Abbasi et al., 2020; Navarro et al., 2014; NVIDIA, n.d.). SIAs naturally lend themselves to parallel implementations, since core operations such as fitness evaluation and local solution updating occur at the per-particle or per-agent level, often with minimal dependency among individuals. Prior studies have documented considerable speedups when porting SIAs to GPU architectures (Tan & Ding, 2015; Yeh, 2017; Yeh & Wei, 2012; Yildirim et al., 2015), highlighting the strong synergy between swarm parallelism and GPU hardware concurrency.

Despite the demonstrated success of GPU-based SIAs, one variant, simplified swarm optimization (SSO), has received limited attention on modern parallel platforms. Since its inception in 2009 (Lee et al., 2012), SSO has proven to be an effective population-based search method, praised for its conceptual simplicity and robust performance on real-world optimization tasks (Corley et al., 2006; Luo et al., 2019; Yeh, 2015). However, existing research on SSO has primarily examined serial (central processing unit [CPU]-based) implementations, leaving a conspicuous gap regarding its parallel potential. By focusing on SSO, researchers can harness its inherently straightforward swarm-update rules to realize high degrees of concurrency. Moreover, the method's minimal parameter requirements and flexible encoding scheme make it a compelling candidate for GPU-based large-scale optimization.

To address this gap, we propose a compute unified device architecture (CUDA) SSO (CUDA-SSO) framework under the NVIDIA CUDA environment. Departing from sequential SSO procedures, CUDA-SSO capitalizes on concurrent kernel launches to distribute the computational workload across thousands of GPU threads. This design not only accelerates fitness evaluations, typically the most time-consuming step in swarm algorithms, but also introduces a parallel update mechanism to circumvent resource-preemption issues associated with personal best (pBest) and global best (gBest) states in swarm-based searches. By carefully encapsulating data in global memory and minimizing CPU–GPU data transfers, we demonstrate both improved solution quality and a drastic reduction in overall execution time.

The main contributions of this paper are:

(i) A novel GPU-based SSO framework (CUDA-SSO) that adopts data-parallel kernels and reduces the theoretical time complexity of swarm search steps.

(ii) A discussion of resource conflict avoidance by re-structuring personal and gBest updates in a parallel context.

(iii) A comprehensive evaluation of standard benchmark functions, showcasing an order-of-magnitude speedup in run time, accompanied by higher solution accuracy than CPU-based SSO implementations.

The remainder of this paper is organized as follows. Section 2 presents an overview of the classical SSO algorithm, the fundamentals of general-purpose GPU computing, and related GPU-based SIAs. Section 3 details the proposed CUDA-SSO algorithm, including its kernel-based design, memory model, and theoretical time complexity analysis. Section 4 provides experimental results with various benchmark functions, comparing performance and precision against the baseline CPU-based SSO. Finally, Section 5 summarizes the findings, discusses potential improvements, and outlines directions for future work.

2. Background

Recent advances in high-performance computing and optimization have witnessed the integration of diverse approaches such as SI, evolutionary strategies, and gradient-based search methods. In particular, SIAs offer decentralized collective search capabilities, while gradient descent (GD) relies on local derivative information to iteratively refine candidate solutions. Understanding how these paradigms intersect—or diverge—can shed light on algorithmic design principles that balance global exploration with local exploitation. This section introduces SSO, a data-parallel swarm algorithm noted for its streamlined update rules. We then highlight key distinctions between GD and swarm-based approaches, discuss the essentials of general-purpose GPU (GPGPU) computing, and conclude with an overview of relevant GPU-based SIAs to contextualize the motivations behind our work on CUDA-SSO.

2.1. SSO

SSO was initially proposed by Yeh (2009) as a lightweight yet robust variant of SI, offering a balance between algorithmic simplicity and practical performance. Unlike more elaborate SIAs (e.g., PSO with velocity–position updates or GA with crossover–mutation operators), SSO employs a small set of parameters (C_w, C_p, and C_g) that guide the sampling of new solutions from each particle's current state (x_{ij}^t), pBest (p_{ij}^t), and gBest (g_j). This approach obviates the need for velocity vectors or mutation rates, reducing the parameter-tuning overhead that can complicate other SIAs.

Fundamentally, each iteration of SSO can be broken into:

(i) Solution update: For each solution i and variable j, the new solution $x_{ij}^{(t+1)}$ is drawn from one of three sources—current solution, pBest, or gBest based on probabilities (C_w, C_p, and C_g).

(ii) Fitness evaluation: Each updated particle is assigned a fitness score $x_{ij}^{(t+1)}$.

(iii) Best-value updates: If f(Xi) is better than a particle's pBest, it is replaced. If f(Xi) outperforms the current gBest, it is updated accordingly.

2.1.1. Fundamental concepts and update strategy

SSO operates over a population $\left\{X_i^t \mid i = 1, 2, \ldots, N_{sol}\right\}$, where $X_i^t = (x_{i,1}^t, x_{i,2}^t, ..., x_{i,m}^t)$ is a vector representing the i^{th} candidate solution at generation t and $x_{i,j}^t$ is the j^{th} variable in X_i^t for $t = 1, 2, \ldots, N_{gen}$ and $i = 1, 2, \ldots, N_{sol}$. Two supporting data structures track the algorithm's progress:

(i) pBests: $P_i = (p_{i,1}, p_{i,2}, p_{i,m})$: The historically best position of each particle, reflecting individually optimal solutions found over previous iterations.

(ii) gBest: $P_{gBest} = (g_1, g_2, g_m)$: The optimal solution observed across the entire population.

Within each iteration, SSO applies a simple step function to update the value of each variable $x_{i,j}^t$ in the solution X_i^t. As shown in Eq. (1), a random number ρ is a random value drawn from a continuous distribution ranging from 0 to 1, which drives the selection among four possibilities: retaining the current value $x_{i,j}^t$, adopting $p_{i,j}$, adopting g_j, or performing no update.

$$x_{i,j}^{t+1} = \begin{cases} x_{i,j}^t \text{ if } \rho \in \left[0, C_w = c_w\right) \\ p_{i,j} \text{ if } \rho \in \left[C_w, C_p = C_w + c_p\right) \\ g_j \text{ if } \rho \in \left[C_p, C_g = C_p + c_g\right) \\ x \text{ if } \rho \in \left[C_g, 1\right) \end{cases} \quad (1)$$

Here, $p_{i,j}$ denotes the j^{th} coordinate of the pBest of the i^{th} solution, and g_j represents the corresponding coordinate in gBest. The relative magnitudes of (C_w, C_p, and C_g) balance exploration (i.e., adopting global or pBests) against exploitation (i.e., retaining current values). This compact parameterization facilitates a more controlled search dynamic than in many other SIAs.

2.1.2. Advantages of SSO over genetic algorithms

Genetic algorithms have historically been a cornerstone of evolutionary computation, relying on crossover and mutation operations to evolve solution populations. However, SSO can frequently perform better in certain problem classes due to its simpler update mechanism and more focused parameter space. Key comparative advantages of SSO include:

(i) Reduced parameter tuning: Traditional GAs demand meticulous adjustment of crossover rates, mutation probabilities, and selection schemes. By contrast, SSO relies on three probabilities (Cw, Cp, and Cg) to guide each variable's update. This hyperparameter reduction often translates into faster and more reproducible experimentation, minimizing the risk of suboptimal tuning.

(ii) Potentially faster convergence: In SSO, particles can directly adopt globally optimal positions, whereas GAs depend on randomized genetic operators to spread promising traits. Consequently, SSO may converge more rapidly on certain continuous or weakly multimodal functions, mainly when the objective landscape permits direct exploitation of high-fitness regions.

(iii) Implementation simplicity: GA-based crossover and mutation operators can become complicated when dealing with high-dimensional or heterogeneous solution representations. SSO's step-function update—requiring only a few lines of code—facilitates implementation clarity, reducing the likelihood of design or coding errors.

(iv) GPU suitability: Although GAs can be parallelized, SSO's probabilistic mechanism, wherein each variable is updated according to a small set of global or pBests, typically presents fewer data dependencies across particles. This structure lends itself well to massive parallelization on GPUs, making SSO an attractive option for large-scale optimization tasks in high-performance computing environments.

Hence, SSO offers a comparatively straightforward and potentially more consistent pathway to large-scale optimization, particularly when research or industrial constraints limit tuning resources or demand high solution fidelity within compressed timeframes.

2.1.3. SSO flowchart

SSO's simplicity has proven advantageous in several applications. For instance, Chung & Wahid (2012) and Yeh (2012; 2013) demonstrate its effectiveness in tackling complex real-world tasks such as reliability design and feature selection. Further refinements, such as orthogonal SSO (Yeh, 2014), reinforce the adaptability of SSO's framework. However, although prior literature confirms SSO's suitability for large-scale research, most studies have employed CPUs, where time complexity grows rapidly

with the population size and dimensionality. This motivates the pursuit of a GPU-based parallelization strategy that can leverage SSO's inherent data-parallel characteristics.

Algorithm 1 outlines the typical CPU-based SSO flow. Each iteration updates particles by sampling the step function, evaluates the fitness value for each particle, and updates pBests and gBest if any improvement is found. Although CPU-SSO can yield excellent results for moderate-scale problems, it becomes slow when the population and number of variables are large.

Algorithm 1. The typical CPU-based SSO

```
Initialize:
    Nsol = 50, Nvar = 30, Ngen = 100
    Var_max = 5.12, Var_min = -5.12
    sol = Nsol × Nvar
    pBests = Nsol × Nvar
    gBest = 0
    Cw = 0.2, Cp = 0.5, Cg = 0.8
    explorationTime = 0
while explorationTime ≤ cpuTimeLimit do
    for iter in 1 to Ngen do
    stepFunc(sol, pBests, gBest, randNum(Var_max,
    Var_min))
     evaluate(solF, pF, gF)
      if solF < pF then pBests(i) = sol(i)
         if solF < gF then gBest = sol(i)
         end if
       end if
     end for
end while
```

2.2. General-Purpose GPU Computing

Modern GPUs were originally engineered to accelerate real-time three-dimensional graphics tasks such as rasterization and shading. Over time, these architectures evolved into GPGPU (Hussain et al., 2016), wherein highly parallel GPU hardware is repurposed to handle a variety of data-intensive computations. By distributing large workloads among thousands of arithmetic cores, developers offload parallel tasks to the GPU while reserving more complex, serial procedures for the CPU.

2.2.1. Execution model (CUDA framework)

NVIDIA's CUDA (NVIDIA, n.d.) extends C/C++ to enable heterogeneous computing. In CUDA, the following function types determine where (CPU vs. GPU) and how (serial vs. parallel) code is executed:

(i) Host functions: Host code is defined in C/C++ and runs on the CPU. It is responsible for high-level logic, memory allocation, and kernel launch.

(ii) Kernel functions: GPU kernels are invoked by the CPU but executed on the GPU, and are subdivided into thread blocks and further organized into warps of 32 threads, following the single instruction, multiple threads paradigm. They are ideal for data-parallel workloads such as fitness evaluations or array/vector operations.

(iii) Device functions: Device functions are defined and executed only on the GPU and are typically called from within kernel functions to factor out repeated computations.

In this model, thousands of concurrent threads can be spawned to run the same kernel, allowing GPUs to efficiently process large, independent datasets.

2.2.2. Compute unified device architecture memory hierarchy

Compute Unified Device Architecture's memory model separates storage into multiple tiers, each balancing capacity and speed.

(i) Registers: Per-thread registers provide high-speed storage and are best suited for frequently accessed variables that do not exceed the register file capacity.

(ii) Shared memory: On-chip shared memory allocated per block enables fast data exchange among threads in the same block and is particularly useful for shared computations, partial sums, and other cooperative tasks where multiple threads access and modify the same data.

(iii) Global memory: Off-chip global memory provides large-capacity storage accessible by all threads but has relatively high latency compared to on-chip resources, making efficient access patterns (e.g., memory coalescing) essential to achieve high throughput.

(iv) Constant and texture memory: Read-only caches accelerate common look-ups and are helpful when all threads repeatedly use the same constant or when two-dimensional array access patterns can be optimized via texture hardware.

High-performance GPU applications often involve coalescing memory accesses, judiciously using shared memory, and minimizing branch divergence (warp divergence). These considerations ensure that multiple threads fetch contiguous elements simultaneously and execute consistent instruction paths whenever possible.

2.2.3. Data transfers and central processing unit-GPU coordination

Since the CPU and GPU have separate memory spaces, data must typically be transferred via the

Peripheral Component Interconnect Express (PCIe) bus. Although essential for many GPGPU workflows, these transfers introduce non-negligible latency. Strategies to reduce transfer overhead include:

(i) Batching data: Copying large chunks of data at a time rather than frequent small transfers.
(ii) Asynchronous transfers: Overlapping data transfers with kernel execution improves device utilization.
(iii) Unified Memory: Leveraging CUDA's managed memory features to let the runtime handle page migrations between CPU and GPU, albeit with some overhead for page-fault handling.

2.2.4. Implications for SIAs

SIAs—including PSO, GA, Firefly Algorithm, and SSO—naturally benefit from GPGPU acceleration due to their population-based structure. Each individual (particle, agent, or chromosome) can be evaluated in parallel, and gBest values can be updated in a relatively small overhead step.

(i) Fitness evaluations: Commonly, the most significant computational bottlenecks can be massively parallelized by assigning a subset of particles (or subdimensions) to separate threads or warps.
(ii) Update mechanisms: Since SIA updates often involve reading global parameters (e.g., best solutions) and then writing back updated values for each particle, careful design of coalesced memory accesses and thread synchronization (e.g., to avoid race conditions when writing to a gBest value) is critical.
(iii) Data dependencies: Many SIAs only require limited information exchange—such as neighbor-based or globally best-based communication—so the parallel workload is generally well-defined. Nonetheless, if a swarm's communication topology is complex (e.g., hierarchical or multiswarm structures), the kernel must incorporate additional synchronization steps or multiple kernel launches to handle inter-group interactions without causing warp divergence or data hazards.

When population sizes or problem dimensions become large, GPU-enabled SIAs can harness thousands of parallel threads across multiple streaming multiprocessors (SMs), substantially reducing run time relative to CPU-only approaches. Consequently, adopting CUDA or similar frameworks for SIAs—while paying close attention to memory usage, thread management, and synchronization—can yield significant speedups in large-scale optimization scenarios. Synchronization in CUDA refers to coordinating the execution of threads to wait for each other at specific points—usually to ensure that data dependencies are respected (i.e., one thread does not read a value before another finishes writing it).

2.3. GPU-Based SIAs Implementation

Parallelization of SIAs on GPUs leverages the natural data-parallel structure of these methods. Within each iteration, every swarm particle (or agent) usually updates its position, evaluates its objective function, and exchanges information with other particles according to the algorithm's communication model.

2.3.1. An Overview of notable GPU-based SIA

Table 1 provides an overview of notable GPU-based SIAs, detailing which functions were ported to GPU kernels in representative studies. The summarized methods include standard and Euclidean PSO (Tsutsui & Fujimoto, 2009; W. Zhu, 2011), multichannel PSO (Krömer et al., 2011), multi-objective Gas (Wong, 2009; H. Zhu et al., 2011), and GA/differential-evolution hybrids (Mussi et al., 2011; Ruder, 2016), among others.

As these steps can be performed independently or partially synchronized, the GPU is well-suited to handle the large number of concurrent threads required to process high-dimensional populations.

2.3.2. Four key kernel functions

SIAs naturally align with parallel architectures due to their population-based structure (Yeh, 2017; Yeh & Wei, 2012). In a GPU context, typical SIA workflows can be divided into four key kernel functions:

(i) Initialize (I): Kernel Function (I) initializes the population with random numbers and stores them in global memory. Benefiting from the intuitive implementation and data access in global memory, most SIAs generated the population on the CPU (NVIDIA Corporation, 2012). It might have got a vast improvement for computing efficiency if (I) the population on GPU instead of CPU, although the way to arrange the global memory may not be that intuitive (Mussi et al., 2011; Ruder, 2016).
(ii) Evaluate fitness (E): Krömer et al. (2011) have demonstrated that the most expensive step in SIAs was to evaluate candidate solutions. The most straightforward to deploy kernel function (E) is the master–slave paradigm, where the centralized controller dispatches particles in a single population for parallelism. This approach introduced no differences from an algorithmic perspective but reduced the time-consuming from a computational perspective.

Table 1. Summary of studies of taxonomy analysis for swarm intelligence algorithms

References	Swarm intelligence algorithm	Methodology	Speedup
Tsutsui & Fujimoto (2009)	Stand particle swarm optimization (PSO)	(I), (C), (U) on CPU. (E) on a GPU without shared memory	×6–8
W. Zhu (2011)	Euclidean PSO	(I), (C), (U) on CPU. (E) on a GPU without shared memory	×1–5
Krömer et al. (2011)	Multichannel PSO	(U) on CPU, (I), (E), (C) on a GPU without shared memory	×30
Wong (2009)	Multi-objective genetic algorithm (GA)	(I) on CPU, (E), (C), (U) on a GPU without shared memory	10–2
H. Zhu et al. (2011)	Coarse-grain parallelization of GA	(I), (C), (U) on CPU, (E) on a GPU only without shared memory	×60
Li & Zhang (2011)	Asynchronous and synchronous PSO	(I), (E), (C), (U) on a GPU with shared memory	-
Mussi et al. (2011)	GA	(I), (E), (C), (U) on a GPU with shared memory	×2–12
Ruder (2016)	GA and differential evolution (DE)	(I), (E), (C), (U) on a GPU with shared memory and synchronization	×3–28 for GA, ×19–34 for DE

Abbreviations: C: Communication; E: Evaluate fitness; I: Initialize; U: Update swarm

As shown in Table 1, Li & Zhang (2011) proposed a CUDA-based multichannel particle swarm algorithm. Wong (2009) implemented a parallel multi-objective GA. Tsutsui and Fujimoto (2009) ran a sequential SIA, dispatching a parallel GA for the particles.

According to NVIDIA (n.d.) and Mussi et al. (2011), using shared memory in GPU code can guarantee speedup for data transferring. However, most did not perform (E) using shared memory.

(i) Communication (C): Unlike the directly distributing function (E), the function (C) proposes a more complicated model. It is distinguished by being loosely connected to the population and irregularly exchanging particles. Communicate mechanisms were enabled between swarms according to the law of data access, which means that communication between distributed groups of particles is acceptable.

(ii) Update Swarm (U): Adjust the positions or velocities (if applicable) of each particle based on shared information. Function (C) and function (U) do not have a single pattern to fit all SIAs. We must only attend to the warp divergence and bank conflict in these two functions.

Across these works, the (E) kernel typically offers the largest room for speedup, since fitness calculation often dominates the total run time. Many authors have thus focused on accelerating (E) by distributing the population's fitness evaluations to GPU threads.

2.3.3. Implementation challenges

Despite the potential computational gains, several implementation challenges arise when porting SIAs to GPUs:

(i) Memory-access patterns and coalescing: Efficient GPU kernels rely heavily on coalesced global-memory transactions, whereby consecutive threads access consecutive memory addresses. Achieving such patterns can involve reorganizing particle data structures, interleaving population elements, or carefully aligning data to minimize misaligned accesses. Failure to do so can negate much of the theoretical speedup from parallelization.

(ii) Shared memory constraints: While shared memory is a low-latency on-chip resource that can accelerate repeated data accesses, the amount available per block (commonly 48 KB or less) may be insufficient for storing large populations or high-dimensional problems. Consequently, many GPU-based SIAs place most of their data in global memory and resort to shared memory only for small suboperations, such as partial sums or local best-value comparisons.

(iii) Warp divergence and synchronization: GPU threads operate in warps of 32 concurrent threads. If branches in the kernel cause differing execution paths within the same warp, performance can degrade significantly due to warp divergence. SIA kernels that incorporate random sampling, conditionals for updating best solutions, or communication topologies must minimize thread divergence and carefully place synchronization barriers (syncthreads or kernel launches) to avoid race conditions when reading/writing global or shared data structures (e.g., gBest positions).

(iv) Communication topologies: In many SIAs, information sharing is crucial for guiding the swarm. This communication can be ring-based,

star-based, hierarchical, or fully connected. Implementing these topologies on a GPU requires balancing frequent data exchanges with the cost of global or shared-memory transactions, especially as the population grows. Some researchers tackle this by employing loosely coupled subswarms, reducing the number of cross-group communications and associated overhead.

(v) Scalability and precision: GPU-based SIAs often demonstrate significant speedups over CPU counterparts when the population size is large enough to saturate GPU resources. However, if the swarm or dimensionality is too small, kernel-launch overhead and data-transfer latencies may outweigh parallelization benefits. Furthermore, some applications demand higher-precision arithmetic (e.g., double precision) that can reduce throughput on specific GPU architectures. Algorithm designers must thus tune swarm sizes, memory layouts, and data precision settings for optimal results.

These considerations indicate that GPU-based SIAs benefit most when carefully tailored to exploit hardware concurrency while mitigating memory and synchronization bottlenecks. Ongoing advances in GPU architectures—expanded on-chip memory, more sophisticated warp schedulers, and built-in library support—continue to ease the adaptation of SIAs for large-scale, real-world optimization problems.

Building on these insights, the present work aims to extend SSO into the GPU domain, integrating the conceptual simplicity of SSO's update mechanism with the massive parallelism of CUDA. Our proposed CUDA-SSO applies kernel-based parallelization to SSO's most time-consuming and data-parallel steps, achieving significant speed gains and avoiding concurrency conflicts when updating personal and gBest states. In the following section, we elaborate on the algorithmic framework of CUDA-SSO, including memory organization, random number generation, and a theoretical complexity analysis.

3. Compute Unified Device Architecture-SSO

Compute Unified Device Architecture-SSO adapts the conventional SSO to leverage CUDA's parallelism. As illustrated in Fig. 1, each kernel function runs concurrently across threads, reducing both evaluation time and memory transaction overhead.

3.1. Random Number Generation

Random number generation (RNG) is essential in SIAs because almost every aspect of the search—particle initialization, stochastic exploration, and crossover/mutation (in other SIAs)—depends on drawing pseudo-random values. In CUDA-SSO, these numbers govern how each variable in a particle decides whether to retain its current value, adopt its pBest, or adopt the gBest. As a result, generating robust random values at high speed is critical to ensure both algorithmic performance and solution diversity.

A naive approach to RNG would compute random numbers on the CPU and then transfer them to the GPU each iteration. However, such data movement across the PCIe bus can introduce significant latency. Instead, CUDA-SSO uses NVIDIA's cuRAND (random number generation library (NVIDIA, n.d.) to generate random numbers directly on the GPU, thereby reducing CPU–GPU switching overhead. The following points highlight key considerations for efficient RNG in CUDA-SSO.

(i) cuRAND generators: NVIDIA's cuRAND library provides multiple generator types (e.g., Philox, Mersenne Twister, and XORWOW) suited to various performance and quality requirements. Philox typically offers a good balance for most GPU-based Monte Carlo or optimization tasks due to its combination of speed and sufficiently robust randomness.

(ii) State management: A dedicated initialization kernel uses cuRAND application programming interfaces to set up independent RNG states for each thread on the GPU. Each state is assigned a seed, sequence number, and offset. This allows threads to maintain independent RNG states, avoiding global memory contention during the main kernel execution.

(iii) Scalability: Due to CUDA-SSO allocating one or more threads per particle/variable, the number of random values can become quite large, reaching Nsol × Nvar × Ngen. However, cuRAND's batched generation methods allow bulk requests of random values, leveraging GPU concurrency to rapidly produce millions of samples.

(iv) Memory footprint and access: RNG states are typically stored in global memory for all threads to access during kernel execution, with each thread updating its local state after retrieving random samples via curand (& state). To minimize overhead, threads often load their RNG state into registers, generate all required samples, and write the state back to global memory only once per iteration, reducing global memory transactions.

(v) Kernel integration: Each thread within the main CUDA-SSO search kernel can invoke cuRAND library calls to draw random floats (e.g., uniform or normal distributions) and apply them to the SSO step function. While careful synchronization may be necessary if multiple threads share RNG states, this is typically avoided by assigning

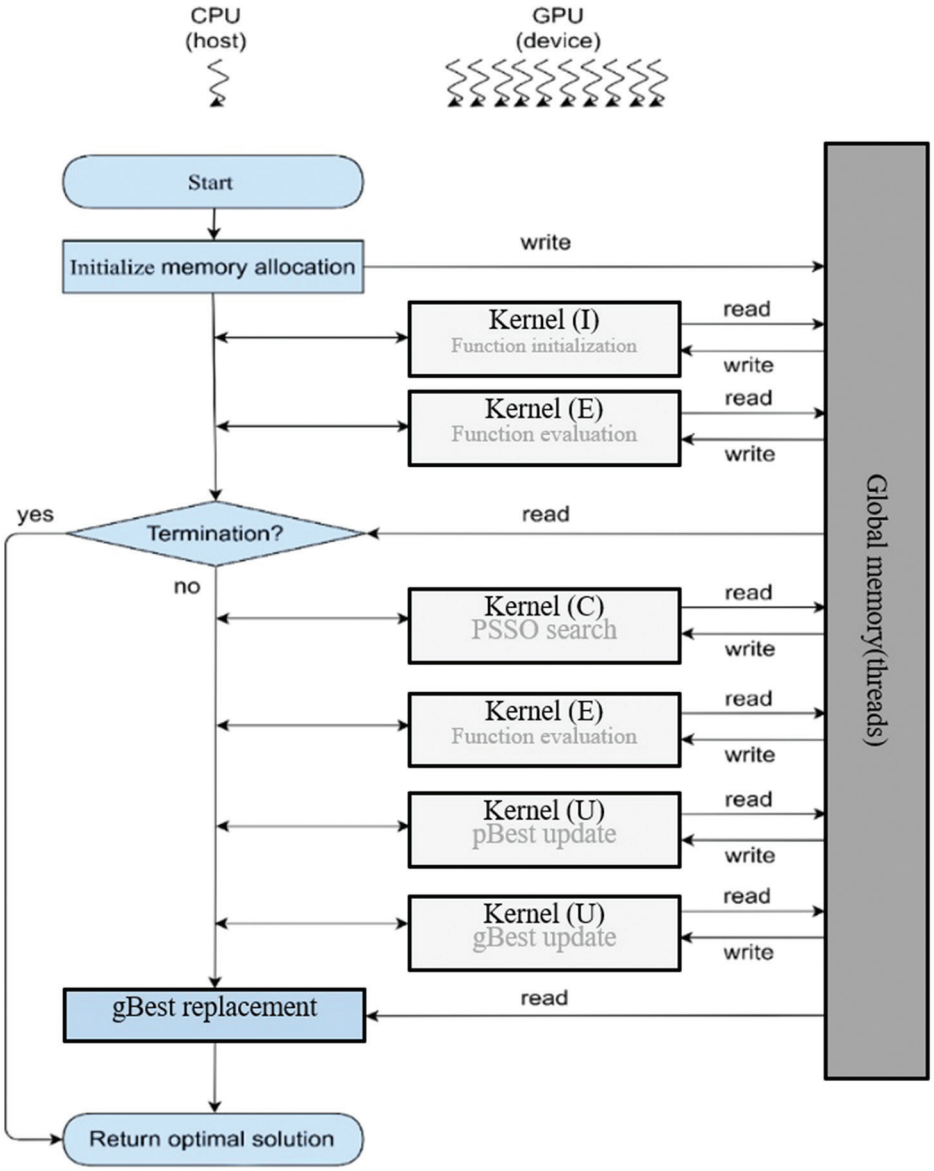

Fig. 1. Proposed compute unified device architecture-simplified swarm optimization
Abbreviations: C: Communication; CPU: Central processing unit; E: Evaluate fitness; gBest: Global best; GPU: Graphics processing unit; I: Initialize; pBests: Personal bests; PSSO: Particle-based simplified swarm optimization; U: Update swarm

unique states to each thread.

(vi) Quality versus speed: While XORWOW offers faster performance, it may exhibit lower randomness quality for specific statistical tests. Although Philox or Mersenne Twister variants may run slightly slower, they often deliver more reliable distributions. While most swarm optimizations work well with any reasonably distributed, uncorrelated RNG, mission-critical or precision-sensitive applications may require more robust generators.

By generating all random numbers on the GPU, CUDA-SSO avoids frequent PCIe transfers and ensures that random samples are available on demand with minimal latency. This strategy significantly improves the algorithm's scalability, allowing $N_{sol} \times N_{var} \times N_{gen}$ random draws to be produced efficiently as the swarm evolves. Consequently, RNG bottlenecks, which often plague GPU-accelerated optimization, are effectively mitigated, paving the way for faster and more diverse exploration in the high-dimensional search space.

3.2. Thread Organization

Efficient thread organization is a cornerstone of high-performance GPU applications, and CUDA-SSO

takes advantage of CUDA's execution hierarchy to maximize throughput and minimize uncoalesced memory accesses. This section details how thread blocks, warps, and memory layouts are arranged to accommodate large particle populations and high-dimensional search problems.

3.2.1. Warp-level particle management

In CUDA-SSO, each warp—consisting of 32 threads—typically maps to one particle, such that the warp's threads can collaboratively handle that particle's variables (position vector, random updates, and fitness computation). This design has several advantages.

(i) Straightforward synchronization: Since a warp executes in a lockstep single-instruction multiple-threads fashion, synchronization within the warp is simpler. For many operations, native warp intrinsics (e.g., __syncwarp()) allow partial sums or shared computations to be done without incurring the overhead of a block-wide synchronization (__syncthreads()).

(ii) Fine-grained parallelism: If a particle has N_{var} variables, they can be distributed across multiple threads, allowing partial work (e.g., updating each variable or computing partial fitness) to proceed in parallel within the same warp.

(iii) Reduced warp divergence: Since all threads in a warp handle logically contiguous parts of the same particle, branching is minimized. Divergence primarily arises if the particle's data triggers conditionals (e.g., random updates to different variables). However, these are usually minor compared to divergences caused by dissimilar data accesses across multiple particles.

Compute unified device architecture's thread blocks group warps together, and a grid of blocks covers the entire population.

Block sizes are chosen in multiples of 32 (e.g., 128, 256, and 512 threads/block) to ensure warp alignment. In CUDA-SSO, a block typically manages several particles—each warp in the block handles a separate particle's data.

Grid sizes are determined by how many blocks are needed to encompass all particles. For instance, if the swarm has N_{sol} = 10,000 particles and each block manages eight warps, we need at least 10,000/8 = 1,250 blocks to cover the swarm. This approach scales well on modern GPUs with multiple SMs capable of running dozens of blocks concurrently.

To fully utilize GPU bandwidth, CUDA-SSO arranges each particle's data (e.g., position vector, best values) contiguously in global memory. When warp threads access consecutive addresses, coalesced reads reduce the required memory transactions. Key design elements include:

(i) Particle-centric layout: The position vector, pBest, and related metadata for each particle are stored back-to-back in memory. Threads within a warp access sequential indices, aligning memory requests with hardware transaction boundaries.

(ii) Avoiding strided access: If data for a single particle were scattered or interleaved with multiple particles, warp threads would fetch non-consecutive addresses, leading to uncoalesced accesses and lowered throughput. By contiguously grouping a particle's variables, CUDA-SSO preserves coalescing even when the swarm is large.

(iii) Shared memory trade-off: Although shared memory can accelerate repeated data accesses (e.g., partial sums), large swarm sizes (hundreds or thousands of particles, each with tens to hundreds of variables) rapidly exceed the typical 48–96 kb shared memory per block. Consequently, global memory becomes the main data store. Nevertheless, kernel designers may still use shared memory for sub-operations (e.g., block-level reductions) if it is feasible within the memory budget.

3.2.2. Synchronization and concurrency

Swarm intelligence demands occasional synchronization to ensure that updated particle states or gBest values are consistently available. In CUDA-SSO, two main synchronization patterns arise:

(i) Warp-level: For tasks that only require threads within the same warp to coordinate—such as partial computation of a single particle's fitness—warp intrinsics (__syncwarp()) suffice. This is faster than a full __syncthreads(), affecting all block threads.

(ii) Block- or grid-level: Specific global or pBest updates may require broader synchronization:

- __Syncthreads() ensures all threads in the block finalize local data before proceeding.
- Multiple kernel launches act as implicit grid-wide barriers, guaranteeing that all blocks complete one stage (e.g., updating pBests) before starting the next (e.g., computing the gBest).

Ensuring all local updates are complete before any best-value comparisons helps avoid race conditions, which might otherwise lead to inconsistent reads or partial updates of shared variables.

For huge swarms or high-dimensional search spaces, a single kernel launch might strain available GPU memory or underutilize certain multiprocessors. CUDA-SSO addresses these scenarios by subdividing the population:

(i) Population splitting: Instead of handling all NsolN_{\mathrm{sol}} particles in one kernel, the

swarm can be partitioned into subsets processed by multiple sequential kernel launches or multiple streams. Each subset undergoes search and fitness evaluation before merging partial bests.

(ii) Multi-kernel scheduling: Modern GPUs support concurrent kernels, enabling partial overlaps in execution. If each subset's memory footprint is smaller, more streams can run concurrently on different SMs, improving load balancing and overall throughput.

(iii) Trade-off: Although subdividing can improve concurrency, it introduces additional steps for merging partial gBest values across subsets. Careful scheduling is needed so that merging overhead does not offset gains from improved load distribution.

By adhering to warp-based particle updates, coalesced memory access patterns, and appropriate synchronization, CUDA-SSO efficiently distributes workload across a GPU's many SMs. In turn, this enables (i) high utilization, where a large swarm or high-dimensional setting can saturate GPU computational resources, (ii) scalability, where as problem sizes grow, additional blocks and warps smoothly extend parallel coverage, and (iii) maintainability, where warp-level design keeps each particle's logic self-contained, simplifying debugging and code maintenance.

Developers must still tune parameters such as block size, register usage, and shared-memory allocations for specific GPU architectures (e.g., differences between NVIDIA Turing, Ampere, or Hopper architectures). Nonetheless, the fundamental strategy—one warp per particle, coalesced global memory, and synchronization barriers for best-value consistency—forms a robust template for realizing scalable, high-performance SI on GPUs (Gordon & Whitley, 1993; Hadley, 1964; Wolpert & Macready, 1995).

3.3. Compute Unified Device Architecture-SSO Implementation

Leveraging GPU-based parallelism requires a careful design of kernel functions, memory layouts, and synchronization strategies. In CUDA-SSO, each iteration (or generation) processes a large population of particles on the GPU, avoiding frequent transfers across the PCIe bus. By dividing search, fitness evaluation, and best-value updates into separate kernels, the algorithm can efficiently harness the GPU's concurrent execution model.

3.3.1. Kernel-launch structure

Algorithm 2 illustrates the main flow of CUDA-SSO. Each generation begins with random number generation on the GPU, followed by parallel kernels for the search process (step function) and fitness evaluations. Afterward, pBests and the gBest are updated in parallel, with each block or warp managing a subset of particles.

Algorithm 2. Flowchart for CUDA-simplified swarm optimization

```
sol = Nsol × Nvar
pBests = Nsol × Nvar
gBest = 0
set block size
syncThreads()

Initialize population
Initialize block size
Transfer data from CPU to GPU

//Kernel functions executed in parallel
for gen = 0 to Ngen do
    Search process for all particles      //stepFuncin
                                          parallel
    syncThreads()
    Update pBest for each solution        //Kernel (U)
    Update gBest for each solution        //Kernel (U)
    syncThreads()
end for

Send data back to the CPU
```

The above design leverages the GPU's parallel capabilities to handle large numbers of particles in each generation and ensures that intermediate results are kept consistent across all threads before the next update commences. Here is how it works:

(i) Parallel kernel launches: The design separates operations into distinct parallel kernels for the search process (step function) and for updating pBests and gBest values. This approach enables the concurrent execution of computation (E) and communication (C) operations before synchronizing for updates (U).

(ii) Synchronization: The system uses syncThreads() or similar synchronization barriers to ensure all threads complete their current operations, whether searching or updating optimal values, before moving forward. This synchronization is vital for preventing race conditions and maintaining consistent pBests and the gBest.

(iii) GPU–CPU transfers: To minimize PCIe bus overhead, data transfers between CPU and GPU occur only twice: once at initialization and once at completion. During iterations, all population data remains in GPU memory, following the memory management guidelines outlined in Section 3.2.

3.3.2. Parallel updates of pBests and gBests

Algorithms 3 and 4 illustrate how pBests and the gBest are updated in a parallel environment. By distributing the workload across GPU threads, CUDA-SSO prevents any single update from dominating run time and fully exploits GPU concurrency.

Algorithm 3. Parallel updates of personal bests.
```
syncThreads()
for each particle i in parallel do
    Load current sol[i] and pBests[i]
    if f(sol[i]) < f(pBests[i]) then
        pBests[i] = sol[i]
    end if
end for
syncThreads()
```

Algorithm 4. Parallel updates of the global best.
```
syncThreads()
for each particle i in parallel do
    Load current pBests[i] and gBest
    if f(pBests[i]) < f(gBest) then
        gBest = pBests[i]
    end if
end for
syncThreads()
```

Implementation details of Algorithms 3 and 4 are discussed in the following:

(i) Warp/block-level work: Each particle is processed in parallel. While it is not explicitly stated that one warp must correspond to a single particle, this configuration can be achieved by selecting suitable block and grid sizes, thereby reducing warp divergence and simplifying synchronization.

(ii) Coalesced memory access: In these snippets, each thread (or warp) reads data stored contiguously in global memory for the assigned particle i. If both sol *i* and pBest *i* reside in adjacent memory locations, warp-level access requests naturally coalesce into fewer transactions.

(iii) Synchronization points: The syncThreads() calls at the start and end of each code block ensure that all local read/write operations to pBests or gBest finish before another kernel or step begins. That is, the communication for global search does not rely on synchronization mechanisms, as these typically incur substantial overhead. Such barriers prevent partial updates or inconsistent reads across parallel threads.

3.4. Time Complexity Analysis

Compared to CPU-SSO's sequential structure, CUDA-SSO distributes the update and evaluation workload over many GPU threads, effectively reducing the time complexity within each iteration. Fig. 2 contrasts CPU-SSO's single-thread approach versus CUDA-SSO's multi-thread parallelism. While CPU-SSO tends to scale with $O(n^3)$ under large

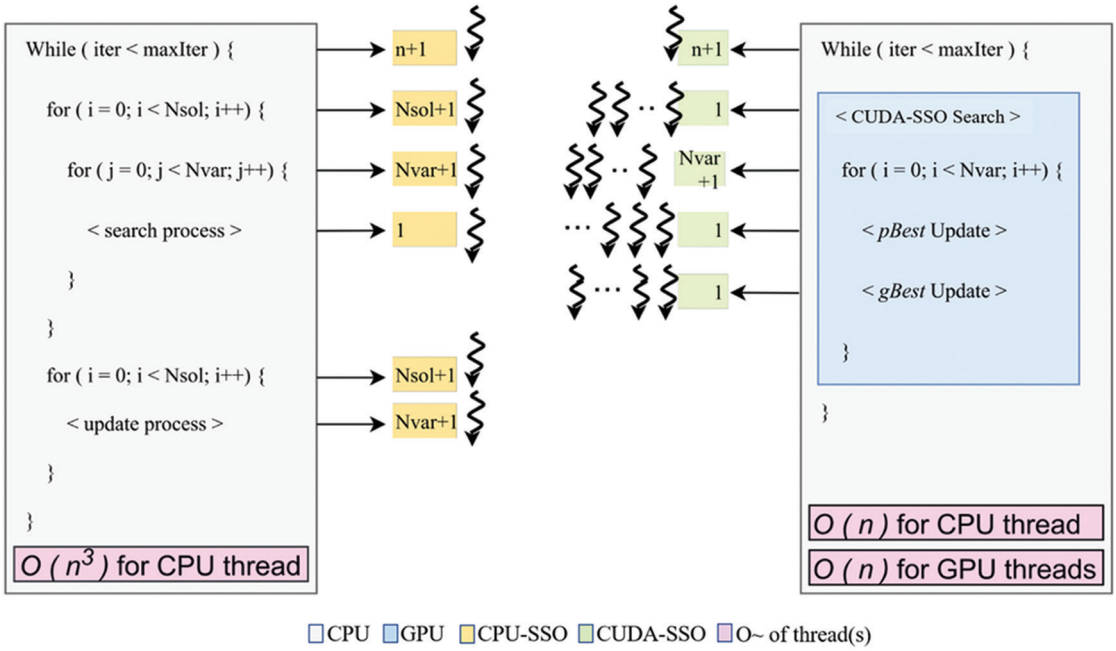

Fig. 2. The time complexity analysis

Abbreviations: C: Communication; CPU: Central processing unit; CUDA: Compute unified device architecture; gBest: Global best; GPU: Graphics processing unit; pBests: Personal bests; SSO: Simplified swarm optimization

W. Zhu, S-K. Huang, etc./Int. J. Systematic Innovation, 9(5), 23-42 (2025)
DOI: 10.6977/IJoSI.202510_9(5).0003

Table 2. Benchmark functions for compute unified device architecture-simplified swarm optimization

ID	Function	Definition	Feasible bounds
f_1	Sphere	$f_1 = \sum_{i=1}^{P} x_i^2$	$[-5.12,5.12]^P$
f_2	Hyper-epllipsoid	$f_2 = \sum_{i=1}^{P} i \cdot x_i^2$	$[-5.12,5.12]^P$
f_3	Schwefel's	$f_3 = \sum_{i=1}^{P} \left(\sum_{j=1}^{i} x_j^2 \right)$	$[-65.536,65.536]^P$
f_4	Rosenbrock	$f_4 = \sum_{i=1}^{P-1} \left[100 \cdot \left(x_{i+1} - x_i^2 \right)^2 + \left(1 - x_i \right)^2 \right]$	$[-2.048,2.048]^P$
f_5	Rastrigin	$f_5 = 10 \cdot P + \sum_{i=1}^{P-1} \left[x_i^2 - 10\left(2\pi x_i \right) \right]$	$[-5.12,5.12]^P$
f_6	Ackley	$f_6 = -a \cdot \exp\left(-b \cdot \sqrt{\frac{1}{P} \sum_{i=1}^{P} x_i} \right) - \exp\left(\frac{1}{P} \sum_{i=1}^{P} \cos\left(c x_i \right) \right)$	$[-32.768,32.768]^P$
f_7	Griewank	$f_7 = \frac{1}{4000} \sum_{i=1}^{P} x_i^2 - \prod_{i=1}^{P} \cos\left(\frac{x_i}{\sqrt{i}} \right) + 1$	$[-600,600]^P$
f_8	Powell	$f_8 = \sum_{i=1}^{P/4} \left[\left(x_{4i-3} + 10 x_{4i-2} \right)^2 \right] + \left[5\left(x_{4i-1} - x_{4i} \right)^2 + \left(x_{4i-2} - 2x_{4i-1} \right)^4 \right] + \left[10\left(x_{4i-3} - x_{4i} \right)^4 \right]$	$[-4,5]^P$
f_9	Schwefel	$f_9 = 418.9829 \cdot P - \sum_{i=1}^{P} x_i \cdot \sin\left(\sqrt{\lvert x_i \rvert} \right)$	$[-5.12,5.12]^P$

population sizes, CUDA-SSO exhibits near $O(n)$ scaling in the dominating computational kernel.

Table 3. Experimental parameters of compute unified device architecture-simplified swarm optimization

No.	Graphics processing unit model	Compute unified device architecture-simplified swarm optimization
1	Block size	C_w, C_p, C_g
2	-	Population size: N_{sol}
3	-	Number of variables: N_{var}
4	-	Number of generations: N_{gen}

Table 4. Factor for the parameters of compute unified device architecture-simplified swarm optimization search

No.	*Cw*, *Cp*, and *Cg*
1	0.1, 0.3, 0.7
2	0.1, 0.4, 0.8
3	0.2, 0.4, 0.6
4	0.2, 0.5, 0.9
5	0.3, 0.4, 0.5
6	0.3, 0.6, 0.8

4. Experiments and Analysis

4.1. Benchmark Functions and Design of Experiments

We tested nine standard benchmark functions, shown in Table 2. These functions include both separable and inseparable properties, with multimodal and unimodal complexities. Each function has a dimension of $N_{var} = 50$. By controlling parameters such as N_{gen} (the maximum iteration count), N_{sol} (population size), and N_{var} (number of variables), we gauge both the convergence (precision) and run time (speedup) of CPU-SSO versus CUDA-SSO.

From Table 3, we know we need to do a seven-factor experimental design, 128 experiments. It is impossible to do such a job with contracted computational resources. Thus, the parameters: block size, N_{sol}, N_{var}, and N_{gen} were arranged as follows: 1,024, 100, 50, and 1000, referring to other papers (Li & Zhang, 2011; NVIDIA Corporation, 2012).

The remaining parameters to be tested are the CUDA-SSO search parameters: C_w, C_p, and C_g. Six parameter levels were evaluated in the experiments, as shown in Table 4. The experimental design of the parameter combinations presented in Table 4 was analyzed using scipy.stats library (Pllana & Xhafa,

Table 5. The parameter combinations analyzed using the Kruskal–Wallis H-test

Parameters	Values					
Cw	0.1	0.1	0.2	0.2	0.3	0.3
Cp	0.3	0.4	0.4	0.5	0.4	0.6
Cg	0.7	0.8	0.6	0.9	0.5	0.8
Method						
Ranking	3,843.173	1,968.923	4,840.817	2,037.200	6,270.421	1,919.306
Statistic	19,1.0773		*p*-value		2.2989086e-39	

Table 6. Precision comparison for central processing unit-simplified swarm optimization and compute unified device architecture-simplified swarm optimization

Function	Central processing unit-simplified swarm optimization			Compute unified device architecture-simplified swarm optimization		
	Average	Standard	Minimum	Average	Standard	Minimum
f_1	54.9497	7.4781	39.0219	41.0156	5.3095	28.5125
f_2	1,152.7869	110.1388	986.4035	820.1844	91.6444	635.6414
f_3	192,950.2539	18,823.6598	162,102.9062	127,504.9484	17,093.0233	103,114.1562
f_4	1,573.8801	179.6216	1,190.2180	1,103.9103	134.5448	730.0332
f_5	269.3232	14.4775	248.3413	220.6183	16.2710	189.2935
f_6	16.7117	0.2739	16.0508	15.2896	0.3655	14.7103
f_7	199.0340	20.2784	156.4854	145.3612	19.4239	95.3518
f_8	1,989.3588	396.4583	1,438.9280	1,181.2840	270.4324	727.8101
f_9	20,719.6228	4.5922	20,706.0234	20,708.0471	3.6021	20,702.3574

2017) by the Kruskal–Wallis H-test. According to the Kruskal–Wallis H-test results in Table 5, the p=2.2989086e-39 is <0.05 in the 95% confidence level, indicating significant differences among the six parameter combinations. Based on the ranking values, the sixth parameter combination demonstrated the best performance. Therefore, the best performance was achieved when the parameters (C_w, C_p, and C_g) were set to (0.3, 0.6, and 0.8), which were adopted as the final parameter settings.

To set the same difficulty in all problems, first, we must choose a dimension particle size (P) search space for all benchmark functions. Second, we use the P obtained from the first step to test the performance of CUDA-SSO. In this subsection, the experiments are executed by the benchmark function f_1.

We implemented CPU-SSO according to Section 2.1 and proposed CUDA-SSO, as described in Section 3. In mimics, we ran f_1–f_9 20 times independently, with 1000 iterations for each run. For CPU-SSO, we performed the same number of function evaluations as CUDA-SSO. The two algorithms have been tested on the same criterion for a fair comparison. The experimental parameters were set as follows: P=50, Cw=0.3, Cp=0.6, Cg=0.8. In our experimental environment, the comparison speedup was tested by N_{sol} = 100, 200, 300, and 350.

Table 7. Friedman test for the precision of the solutions in compute unified device architecture-simplified swarm optimization

Function	Statistic	*p*-value
f_1	19.9200	0.0002
f_2	24.6000	0.0000
f_3	24.6000	0.0000
f_4	21.9600	0.0001
f_5	24.6000	0.0000
f_6	24.9600	0.0000
f_7	21.7200	0.0001
f_8	23.1600	0.0000
f_9	19.5600	0.0002

4.2. Precision and Speedup

This subsection shows the trial for CPU-SSO and CUDA-SSO in 20 independent runs by testing the benchmark functions (Table 2). The average result and corresponding standard deviation are illustrated in Table 6. We utilized the Friedman test (Friedman, 1994) to verify differences. As described in Table 7, most cases have statistical differences for the precision of the solutions in CUDA-SSO.

In addition, the algorithmic flow and data structure of CUDA-SSO (Section 3.3) significantly improved the value of gBest. Table A1 shows the output data of the precision of the solutions for CUDA-SSO.

In general, as far as the average and the minimum of the performances were concerned, CUDA-SSO's performances on multimodal function and unimodal function f1 to f9 worked better than CPU-SSO.

Besides the precision of the solutions, efficiency is a critical factor that must be considered. Speedup and efficiency are among the most common measurement methods to compare the test results. They were illustrated in Eq. (2) and Eq. (3). Nevertheless, either speedup or efficiency cannot reflect the exploitation of computational power. Thus, our research adopted performance criteria: rectified efficiency (Eq. [4]).

$$Speedup = \frac{Time_{CPU}}{Time_{GPU}} \tag{2}$$

$$Ratio = \frac{Power_{GPU}}{Power_{CPU}} \tag{3}$$

$$RE = \frac{Speedup}{Ratio} \tag{4}$$

The output data of the speedup test for CUDA-SSO is listed in Table A2. Speedup experiments are depicted in Table 8. A series of experiments was carried out to check the speedup of CPU-SSO and CUDA-SSO. Among these experiments, the Nsol was set to 100, 200, 300, and 350, respectively. The result showed that CUDA-SSO accelerates up to ×164.2206 compared with CPU-SSO when Nsol = 100. The speedup's performance was becoming more prominent as the size of Nsol increased. The maximum speedup was ×1,604.3382 in the case of Nsol = 350.

Table 8. Running time and speedup for the benchmark function Rosenbrock

Nsol	Central processing unit-simplified swarm optimization	Compute unified device architecture-simplified swarm optimization	Rectified efficiency
100	48.8263	0.13875	164.2206
200	193.10285	0.154	585.1602
300	434.8518	0.1638	1,238.8940
350	582.71855	0.1695	1,604.3382

5. Conclusion

This paper introduced a GPU-based CUDA-SSO, leveraging the well-known SSO's simplicity and integrating it into the CUDA framework. By adopting a parallel processing strategy and minimizing data transfers between CPU and GPU, CUDA-SSO excels in computational speed and solution precision. Our experiments demonstrated:

(i) Time complexity reduction: CUDA-SSO mitigated CPU-SSO's O(n3) scalability issues by distributing the workload across thousands of GPU threads.
(ii) Significant speedups: For benchmark functions, CUDA-SSO outperformed CPU-SSO with speedups up to ×1,604.34\times 1,604.34 at larger population sizes.
(iii) Improved solution accuracy: Statistical analysis (Friedman and Kruskal–Wallis tests) showed that CUDA-SSO yielded notably higher-quality solutions than CPU-SSO across multiple benchmark functions.

To improve the overall efficiency of the proposed approach, future research may explore alternative memory allocation strategies, as memory management plays a crucial role in the performance of parallel and distributed systems—particularly where access speed and bandwidth are critical. Adaptive memory techniques can help reduce latency, lower contention, and optimize resource usage. In addition, parameter tuning and choosing algorithmic parameters that significantly impact model effectiveness and computational cost should be emphasized. Future studies can achieve more scalable, efficient, and reliable performance by integrating efficient memory management with robust parameter tuning. Although rectified efficiency is introduced, future research could provide rigorous justification or comparisons with traditional parallel efficiency metrics.

Acknowledgments

None.

Funding

This research was supported in part by the Natural Science Foundation of China (Grant No. 62106048) and the Ministry of Science and Technology (MOST), China, under grants MOST 107-2221-E-007-072-MY3, MOST 110-2221-E-007-107-MY3, and NSTC 113-2221-E-007-117-MY3.

Conflict of Interest

The authors declare that there are no conflicts of interest.

Author Contributions

Conceptualization: Wenbo Zhu, Shang-Ke Huang, Wei-Chang Yeh

Formal analysis: All authors

Methodology: Wenbo Zhu, Shang-Ke Huang, Wei-Chang Yeh

Writing–original draft: All authors

Writing–review & editing: Wenbo Zhu, Shang-Ke Huang, Wei-Chang Yeh, Zhenyao Liu

Availability of Data

Descriptions of the research data are provided in Section 4.1.

Further Disclosure

A preliminary version of this work was briefly posted on arXiv for reference (https://doi.org/10.48550/arXiv.2110.01470).

References

Abbasi, M., Rafiee, M., Khosravi, M.R., Jolfaei, A., Menon, V.G., & Koushyar, J. M. (2020). An efficient parallel genetic algorithm solution for vehicle routing problem in cloud implementation of the intelligent transportation systems. *Journal of Cloud Computing*, 9(6), 6. https://doi.org/10.1186/s13677-020-0157-4

AlZubi, S., Shehab, M., Al-Ayyoub, M., Jararweh, Y., & Gupta, B. (2020). Parallel implementation for 3D medical volume fuzzy segmentation. *Pattern Recognition Letters*, 130, 312–318. https://doi.org/10.1016/j.patrec.2018.07.026

Chung, Y.Y., & Wahid, N. (2012). A hybrid network intrusion detection system using simplified swarm optimization (SSO). *Applied Soft Computing*, 12(9), 3014–3022. https://doi.org/10.1016/j.asoc.2012.04.020

Corley, H.W., Rosenberger, J., Yeh, W.C., & Sung, T.K. (2006). The cosine simplex algorithm. *The International Journal of Advanced Manufacturing Technology*, 27, 1047–1050. https://doi.org/10.1007/s00170-004-2278-1

Corral, J.M.R., Civit-Masot, J., Luna-Perejón, F., Díaz-Cano, I., Morgado-Estévez, A., & Domínguez-Morales, M. (2024). Energy efficiency in edge TPU vs. Embedded GPU for computer-aided medical imaging segmentation and classification. *Engineering Applications of Artificial Intelligence*, 127, 107298. https://doi.org/10.1016/j.engappai.2023.107298

Friedman, J.H. (1994). An overview of predictive learning and function approximation. In: *From Statistics to Neural Networks*. Vol. 136. Springer,

Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-79119-2_1

Gordon, V.S., & Whitley, D. (1993). Serial and parallel genetic algorithms as function optimizers. In: *Proceedings of the 5th International Conference on Genetic Algorithms (ICGA)*. Urbana-Champaign, IL, USA.

Hachaj, T., & Piekarczyk, M. (2023). High-level hessian-based image processing with the frangi neuron. *Electronics*, 12(19), 4159. https://doi.org/10.3390/electronics12194159

Hadley, G. (1964). *A Nonlinear and Dynamics Programming*. Addison-Wesley Professional, Reading, MA, USA.

Hager, G., Zeiser, T., & Wellein, G. (2008). Data Access Optimizations for Highly Threaded Multi-Core Cpus with Multiple Memory Controllers. In: *Proceedings of 2008 IEEE International Symposium on Parallel and Distributed Processing*, p1–7. https://doi.org/10.1109/IPDPS.2008.4536341

Hussain, M.M., Hattori, H., & Fujimoto, N. (2016). A CUDA implementation of the standard particle swarm optimization. In: *Proceedings of 2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC) IEEE*, Timisoara, Romania. p24–27. https://doi.org/10.1109/SYNASC.2016.043

Krömer, P., Platoš, J., Snášel, V., & Abraham, A. (2011). A comparison of many-threaded differential evolution and genetic algorithms on CUDA. In: *Proceedings of 2011 Third World Congress on Nature and Biologically Inspired Computing: IEEE*, Salamanca, Spain, p19–21. https://doi.org/10.1109/NaBIC.2011.6089641

Lee, W.C., Chuang, M.C., & Yeh, W.C. (2012). Uniform parallel-machine scheduling to minimize makespan with position-based learning curves. *Computers and Industrial Engineering*, 63(4), 813–818. https://doi.org/10.1016/j.cie.2012.05.003

Li, W., & Zhang, Z. (2011). A Cuda-Based Multi-Channel Particle Swarm Algorithm. In: *Proceedings of 2011 International Conference on Control, Automation and Systems Engineering (CASE): IEEE*, Singapore. https://doi.org/10.1109/ICCASE.2011.5997786

Luo, C., Sun, B., Yang, K., Lu, T., & Yeh, W.C. (2019). Thermal infrared and visible sequences fusion tracking based on a hybrid tracking framework with adaptive weighting scheme. *Infrared Physics and Technology*, 99, 265–276. https://doi.org/10.1016/j.infrared.2019.04.017

Mittal, S., & Vetter, J.S. (2014). A survey of methods for analyzing and improving GPU energy efficiency. *ACM Computing Surveys (CSUR)*, 47(2), 1–23. https://doi.org/10.1145/263634

Mortezazadeh, M., Wang, L.L., Albettar, M., & Yang, S. (2022). CityFED - city fast fluid dynamics for Urban microclimate simulations on graphics processing units. *Urban Climate*, 41, 101063. https://doi.org/10.1016/j.uclim.2021.101063

Mussi, L., Daolio, F., & Cagnoni, S. (2011). Evaluation of parallel particle swarm optimization algorithms within the CUDA™ architecture. *Information Sciences*, 181(20), 4642–4657. https://doi.org/10.1016/j.ins.2010.08.045

Navarro, C.A., Hitschfeld-Kahler, N., & Mateu, L. (2014). A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Communications in Computational Physics*, 15(2), 285–329. https://doi.org/10.4208/cicp.110113.010813a

NVIDIA Corporation. (2012). *CUDA C Best Practices Guide*. Ver. 4.1. [Technical Report]. United States: NVIDIA Corporation.

NVIDIA. (n.d.). *CUDA C Programming Guide*. NVIDIA Documentation. Available from: https://docs.nvidia.com/cuda/cuda-c-programming-guide [Last accessed on 2024 Nov 01].

Pllana, S., & Xhafa, F. (2017). *Programming Multicore and Many-Core Computing Systems*. John Wiley and Sons, United States. https://doi.org/10.1002/9781119332015

Ruder, S. (2016). *An Overview of Gradient Descent Optimization Algorithms*. [arXiv Preprint].

Tan, Y., & Ding, K. (2015). A survey on GPU-based implementation of swarm intelligence algorithms. *IEEE Transactions on Cybernetics*, 46(9), 2028–2041. https://doi.org/10.1109/TCYB.2015.2460261

Tsutsui, S., & Fujimoto, N. (2009). Solving quadratic assignment problems by genetic algorithms with GPU computation: A case study. In: *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*. Montreal, Québec, Canada, 2009. p8–12. https://doi.org/10.1145/1570256.157035

Wolpert, D.H., & Macready, W.G. (1995). *No Free Lunch Theorems for Search. Technical Report SFI-TR-95-02-010*. Santa Fe Institute, United States.

Wong, M.L. (2009). Parallel multi-objective evolutionary algorithms on graphics processing units. In: *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*. Montreal, QC, Canada. https://doi.org/10.1145/1570256.1570354

Yeh, W.C. (2009). A two-stage discrete particle swarm

optimization for the problem of multiple multi-level redundancy allocation in series systems. *Expert Systems with Applications*, 36(5), 9192–9200. https://doi.org/10.1016/j.eswa.2008.12.024

Yeh, W.C. (2012). Simplified swarm optimization in disassembly sequencing problems with learning effects. *Computers and Operations Research*, 39(9), 2168–2177. https://doi.org/10.1016/j.cor.2011.10.027

Yeh, W.C. (2013). New parameter-free simplified swarm optimization for artificial neural network training and its application in the prediction of time series. *IEEE Transactions on Neural Networks and Learning Systems*, 24(4), 661–665. https://doi.org/10.1109/TNNLS.2012.2232678

Yeh, W.C. (2014). Orthogonal simplified swarm optimization for the series-parallel redundancy allocation problem with a mix of components. *Knowledge Based Systems*, 64, 1–12. https://doi.org/10.1016/j.knosys.2014.03.011

Yeh, W.C. (2015). An improved simplified swarm optimization. *Knowledge Based Systems*, 82, 60–69. https://doi.org/10.1016/j.knosys.2015.02.022

Yeh, W.C. (2017). A new exact solution algorithm for a novel generalized redundancy allocation problem. *Information Sciences*, 408, 182–197. https://doi.org/10.1016/j.ins.2017.04.019

Yeh, W.C., & Wei, S.C. (2012). Economic-based resource allocation for reliable grid-computing service based on Grid Bank. *Future Generation Computer Systems*, 28(7), 989–1002. https://doi.org/10.1016/j.future.2012.03.005

Yildirim, E., Arslan, E., Kim, J., & Kosar, T. (2015). Application-level optimization of big data transfers through pipelining, parallelism and concurrency. *IEEE Transactions on Cloud Computing*, 4(1), 63–75. https://doi.org/10.1109/TCC.2015.2415804

Zhu, H., Guo, Y., Wu, J., Gu, J., & Eguchi, K. (2011). Paralleling euclidean particle swarm optimization in CUDA. In: Proceedings of *2011 4th International Conference on Intelligent Networks and Intelligent Systems: IEEE*, Kuming, China. https://doi.org/10.1109/ICINIS.2011.35

Zhu, W. (2011). Massively parallel differential evolution-pattern search optimization with graphics hardware acceleration: An investigation on bound constrained optimization problems. *Journal of Global Optimization*, 50(3), 417–437. https://doi.org/10.1007/s10898-010-9590-0

AUTHOR BIOGRAPHIES

Wenbo Zhu is currently affiliated with the School of Mechatronical Engineering and Automation, Foshan University, Foshan, Guangdong, China. He has published original research articles in journals such as *Signal Processing*, *Journal of Medical Imaging and Health Informatics*, and *Computers in Biology and Medicine*. His current research interests include artificial intelligence algorithms, particularly pattern recognition, machine learning, and evolutionary computation.

Shang-Ke Huang is currently affiliated with the Integration and Collaboration Laboratory, Department of Industrial Engineering and Engineering Management, College of Engineering, National Tsing Hua University, Hsinchu, Taiwan.

Wei-Chang Yeh is a Chair Professor of the Department of Industrial Engineering and Engineering Management at National Tsing Hua University in Taiwan. He received his M.S. and Ph.D. degrees in Industrial Engineering from the University of Texas at Arlington. His current and future research focus primarily on algorithm development, including exact solution methods and soft computing approaches applied to various network reliability and optimization problems (e.g., wireless sensor network, cloud computing, IoT, big data, and energy systems). He has published more than 300 research papers in high-ranking journals and conferences and has received numerous awards, including the two Outstanding Research Awards, a Distinguished Scholars Research Project Award, and two Overseas Research Fellowships from the Ministry of Science and Technology in Taiwan.

Zhenyao Liu is an Assistant Professor in the School of Economics and Management, Taizhou University, Jiangsu Province, China. He received his Ph.D. degree in Industrial Engineering and Engineering Management from National Tsing Hua University, Taiwan. His research areas include soft computing and machine learning.

Chia-Ling Huang is a Professor in the Department of International Logistics and Transportation Management at Kainan University, Taiwan. She received her Ph.D. in Industrial Engineering and Management from National Chiao Tung University, Hsinchu, Taiwan. Her research interests include reliability, network analysis, and statistical applications.

Appendix

Table A1. Output data of the precision of the solutions for compute unified device architecture-simplified swarm optimization

Type	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9
CPU	57.82	1,069.30	185,060.89	1,344.71	259.23	16.95	196.72	1,786.07	20,718.92
CPU	39.02	1,207.01	179,721.33	1,579.31	260.32	16.84	234.03	1,438.93	20,712.57
CPU	60.39	1,010.91	231,277.19	1,305.95	251.80	16.52	181.27	2,504.35	20,706.02
CPU	49.92	1,024.91	217,473.16	1,595.23	253.40	16.88	175.41	2,661.07	20,722.49
CPU	53.80	993.56	234,086.36	1,466.08	291.53	16.65	200.09	1,911.11	20,721.33
CPU	56.81	1,213.34	195,778.31	1,601.14	284.36	16.58	202.84	1,842.27	20,721.46
CPU	53.31	1,058.56	194,398.47	1,479.26	263.41	16.76	203.98	2,825.84	20,725.72
CPU	47.54	1,361.57	190,197.06	1,705.98	249.95	17.20	184.71	1,768.86	20,719.32
CPU	69.00	986.40	162,102.91	1,647.99	269.47	16.62	213.29	1,462.96	20,721.66
CPU	61.88	1,281.48	173,873.59	1,536.36	286.90	16.72	189.20	1,773.87	20,719.98
CPU	62.03	1,256.16	184,865.73	1,619.13	279.64	16.62	201.49	2,083.50	20,713.47
CPU	60.32	1,204.71	192,596.94	1,699.70	265.15	16.40	218.22	2,384.77	20,722.15
CPU	49.26	1,147.99	200,337.53	1,679.18	284.74	17.02	197.28	1,662.26	20,717.72
CPU	63.23	1,041.88	212,481.92	1,731.55	257.07	16.46	235.21	1,544.44	20,721.78
CPU	61.97	1,206.52	164,635.55	1,641.58	278.79	16.41	158.47	1,481.78	20,717.70
CPU	60.68	1,233.61	177,676.94	1,190.22	285.63	16.05	200.81	2,092.47	20,719.91
CPU	51.53	1,261.56	200,216.28	1,470.10	280.31	17.14	156.49	1,922.02	20,719.85
CPU	44.84	1,242.82	194,000.47	1,972.64	248.34	16.89	205.72	2,448.71	20,727.23
CPU	50.65	1,210.58	182,236.14	1,385.07	251.20	16.96	215.06	2,038.71	20,719.54
CPU	44.97	1,042.85	185,988.31	1,826.44	285.23	16.57	210.41	2,153.19	20,723.66
GPU	43.39	855.83	118,060.55	1,063.94	189.29	15.43	156.66	1,188.65	20,704.46
GPU	45.56	725.88	141,413.03	1,092.53	231.59	14.71	159.39	1,249.47	20,707.90
GPU	45.01	967.91	131,710.67	730.03	205.58	15.26	95.35	727.81	20,714.68
GPU	36.17	845.70	134,990.25	1,338.93	205.13	15.05	143.00	1,039.97	20,709.13
GPU	,43.27	939.87	129,875.73	972.40	192.81	15.21	154.22	962.32	20,702.36
GPU	34.54	821.64	111,364.34	1,299.02	242.38	15.21	167.35	904.35	20,708.38
GPU	42.41	782.17	133,603.25	1,074.08	211.55	15.45	135.07	1,211.44	20,710.81
GPU	42.46	739.65	108,214.88	1,248.61	222.93	15.59	133.56	1,332.69	20,716.58
GPU	54.16	912.20	103,114.16	1,077.81	227.66	15.91	170.80	1,028.79	20,706.01
GPU	37.96	871.04	114,409.24	1,021.06	237.07	15.30	124.51	1,232.94	20,705.02
GPU	28.51	860.33	130,606.30	1,206.30	207.38	15.42	126.97	742.96	20,710.50
GPU	37.52	916.54	137,729.39	1,190.32	236.03	15.73	128.43	1,276.49	20,707.72
GPU	33.99	936.44	145,870.27	1,209.92	220.32	15.56	156.92	925.97	20,706.60
GPU	38.63	804.80	121,314.86	1,177.88	225.26	14.82	147.35	1,715.38	20,704.36
GPU	39.50	645.15	127,713.55	1,133.51	230.44	15.76	136.74	1,054.60	20,707.92
GPU	45.26	727.07	104,419.79	1,066.93	254.98	14.72	159.17	1,357.28	20,710.06
GPU	43.49	844.17	155,562.56	914.05	228.61	14.74	180.26	1,749.65	20,703.45
GPU	41.12	809.00	117,463.82	1,139.54	210.75	15.54	162.61	1,450.71	20,711.72
GPU	44.30	635.64	111,732.80	1,024.59	207.29	15.58	139.15	1,406.60	20,708.95
GPU	43.08	762.64	170,929.52	1,096.76	225.29	14.80	129.73	1,067.60	20,704.33

Abbreviations: CPU: Central processing unit; GPU: Graphics processing unit.

Table A2. Output data of the speedup test for compute unified device architecture-simplified swarm optimization

Type	Particle size	100	200	300	350
CPU	1	49.063	191.183	437.161	564.453
CPU	2	49.073	189.712	439.999	562.614
CPU	3	48.418	190.58	440.908	565.67
CPU	4	47.88	192.824	437.476	563.651
CPU	5	47.758	192.861	428.533	563.799
CPU	6	48.389	191.056	434.753	565.119
CPU	7	49.176	188.301	434.557	571.929
CPU	8	48.248	190.205	431.854	575.904
CPU	9	48.212	189.323	435.387	568.348
CPU	10	50.346	189.678	432.782	582.892
CPU	11	49.061	192.337	432.366	583.594
CPU	12	49.662	194.05	427.215	607.547
CPU	13	49.306	195.631	429.057	601.964
CPU	14	49.547	192.663	433.167	598.993
CPU	15	48.484	197.172	435.056	599.659
CPU	16	48.968	196.5	432.65	598.553
CPU	17	48.827	195.68	439.168	604.617
CPU	18	48.903	197.722	436.881	591.776
CPU	19	47.779	196.185	439.258	594.146
CPU	20	49.426	198.394	438.808	589.143
Average		48.8263	193.10285	434.8518	582.71855
GPU	1	0.15	0.166	0.18	0.19
GPU	2	0.139	0.152	0.174	0.167
GPU	3	0.139	0.144	0.162	0.166
GPU	4	0.138	0.144	0.169	0.174
GPU	5	0.138	0.142	0.156	0.161
GPU	6	0.139	0.143	0.167	0.172
GPU	7	0.141	0.148	0.157	0.16
GPU	8	0.136	0.154	0.16	0.169
GPU	9	0.137	0.159	0.161	0.17
GPU	10	0.136	0.146	0.166	0.165
GPU	11	0.137	0.173	0.166	0.165
GPU	12	0.136	0.168	0.162	0.172
GPU	13	0.137	0.153	0.163	0.169
GPU	14	0.143	0.152	0.162	0.177
GPU	15	0.144	0.151	0.166	0.161
GPU	16	0.135	0.16	0.165	0.177
GPU	17	0.14	0.168	0.165	0.169
GPU	18	0.135	0.158	0.158	0.166
GPU	19	0.134	0.15	0.158	0.17
GPU	20	0.141	0.149	0.159	0.17
Average		0.13875	0.154	0.1638	0.1695

Abbreviations: CPU: Central processing unit; GPU: Graphics processing unit.