FastPSO: Towards Efficient Swarm Intelligence Algorithm on GPUs

Hanfeng Liu[†§], Zeyi Wen[‡1], Wei Cai[†§1]
kurt.liuhf@gmail.com,zeyi.wen@uwa.edu.au,caiwei@cuhk.edu.cn
[†]School of Science and Engineering, The Chinese University of Hong Kong, Shenzhen, China
[§]Shenzhen Institute of Artificial Intelligence and Robotics for Society, China
[‡]Department of Computer Science and Software Engineering, The University of Western Australia

ABSTRACT

Particle Swarm Optimization (PSO) has been widely used in various optimization tasks (e.g., neural architecture search and autonomous vehicle navigation), because it can solve non-convex optimization problems with simplicity and efficacy. However, the PSO algorithm is often time-consuming to use, especially for high-dimensional problems, which hinders its applicability in time-critical applications. In this paper, we propose novel techniques to accelerate the PSO algorithm with GPUs. To mitigate the efficiency bottleneck, we formally model the PSO optimization as a process of element-wise operations on matrices. Based on the modeling, we develop an efficient GPU algorithm to perform the element-wise operations in massively parallel using the tensor cores and shared memory. Moreover, we propose a series of novel techniques to improve our proposed algorithm, including (i) GPU resource-aware thread creation to prevent creating too many threads when the number of particles/dimensions is large; (ii) designing parallel techniques to initialize swarm particles with fast random number generation; (iii) exploiting GPU memory caching to manage swarm information instead of allocating new memory and (iv) developing a schema to support customized swarm evaluation functions. We conduct extensive experiments on four optimization applications to study the efficiency of our algorithm called "FastPSO". Experimental results show that FastPSO consistently outperforms the existing CPU-based PSO libraries by two orders of magnitude, and transcends the existing GPU-based implementation by 5 to 7 times, while achieving better or competitive optimization results.

CCS CONCEPTS

• **Computing methodologies → Parallel algorithms**.

KEYWORDS

Particle Swarm Optimization, High Performance Computing, GPU

ACM Reference Format:
Hanfeng Liu[†§], Zeyi Wen[‡1], Wei Cai[†§1]. 2021. FastPSO: Towards Efficient Swarm Intelligence Algorithm on GPUs. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472456.3472474

1 INTRODUCTION

Swarm Intelligence mimics the behaviors of social animals (e.g., ants and bees), and exploits information exchanges among individuals in the group to achieve intelligence. Recently, many researchers believe that the advancement of Swarm Intelligence may lead to the next evolution of Artificial Intelligence [4, 22]. Well-known swarm intelligence algorithms include Ant Colony algorithm [5] and Artificial Bee Colony algorithm [13]. Compared with other types of optimization methods like Stochastic Gradient Descent (SGD), Swarm Intelligence based optimizations have the advantages in flexibility, robustness and simplicity of implementation [28].

Particle Swarm Optimization (PSO) is one of the most popular Swarm Intelligence algorithms [15]. PSO has been widely used in many applications, such as neural network architecture search [11, 12] and location management [8]. Figure 1 shows the large number of scientific publications in recent years based on data from Google Scholar. However, the optimization process of PSO is often time-consuming, especially when dealing with a large number of particles or with high-dimensional data. The reason is that when the number of particles or dimensions increases, the number of values within each particle of PSO increases and hence the update cost on the values increases. There have been attempts to reduce the time consumption of PSO [10]. The most common way to accelerate PSO is to exploit hardware such as multi-core CPUs and GPUs. In those algorithms with multi-core CPUs or GPUs, a thread is dedicated to each particle, so that the swarm update process can be solely performed within the thread. Such implementations on GPUs are inefficient [10], because the number of particles may be significantly smaller than the number of GPU cores.

To address the efficiency issues of PSO on GPUs, we first identify that the bottleneck of PSO lies in the update of position and velocity of each particle. Hence, we model the swarm update process of the PSO algorithm as element-wise operations on matrices, which is a finer granularity of parallelism compared to particle level parallelism. Thus, we design a fine grain parallel algorithm on GPUs, where a thread is dedicated to one or more element-wise operations depending on the total number of element-wise operations and the available GPU resources. Our proposed PSO can use GPU resources

[1]Corresponding author

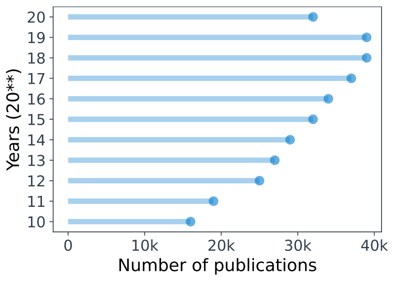


Figure 1: Number of publications of PSO in recent years

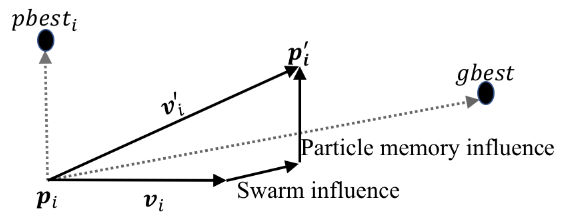


Figure 2: Updating position and velocity of i-th particle

more efficiently, since the number of element-wise operations is significantly larger than the number of particles, which provides more flexibility for parallelism in our algorithm. As a result, our proposed PSO algorithm is more efficient, especially when dealing with high-dimension optimizing problems. Moreover, we propose a series of novel techniques to improve our proposed algorithm, including the use of shared memory and tensor cores, the design of parallel swarm initialization techniques with fast random number generation, caching GPU memory to efficiently manage swarm information and supporting user-defined swarm evaluation functions by a GPU kernel schema. To summarize, we make the following major contributions in this paper.

- We accelerate the PSO algorithm on GPUs by modeling the update process of PSO as element-wise multiplication operations on matrices, which brings higher degree of parallelism. Our algorithm can take advantage of the tensor cores and shared memory.
- We develop a series of novel techniques to further improve our proposed PSO algorithm. First, we propose GPU resource-aware thread creation to prevent the explosion of the number of threads when handling problems with a large number of particles or dimensions. Second, we design parallel techniques to initialize swarm particles before PSO starts with fast random number generation on GPUs. Third, we exploit GPU memory caching to better manage swarm information instead of allocating new memory in the optimization process. Lastly, we provide a schema to support customized swarm evaluation functions which are automatically parallelized on GPUs.
- We conduct extensive experiments to study the efficiency of our proposed PSO algorithm called "FastPSO" on four common optimization problems. The experimental results show that FastPSO can outperform the existing GPU-based PSO implementation by 5 to 7 times. FastPSO is superior especially when processing the high-dimension problems. When compared with the existing CPU-based PSO libraries, FastPSO is two orders of magnitude faster. To further investigate FastPSO, we implement the sequential version of FastPSO and the parallel version of FastPSO with OpenMP. Our study shows that FastPSO on the GPU is an order of magnitude faster than the CPU-based versions, which indicates our efficient use of the GPU resources. Additionally, we perform a case study on searching for the best configuration of thread/block dimensions a GPU-based machine learning library (i.e., ThunderGBM [32]). The accelerated PSO can further reduce the training time of ThunderGBM by around 10% with better thread/block dimension configuration.

The rest of this paper is organized as follows. Section 2 introduces the background of PSO. Section 3 elaborates the details of our proposed PSO algorithm. Section 4 provides experimental results on the efficiency, error and breakdown analysis of the accelerated PSO. A case study of tuning block and thread dimensions on ThunderGBM is also shown in Section 4. Section 5 gives the related works of PSO acceleration. Section 6 concludes the paper and presents potential future research directions.

2 PARTICLE SWARM OPTIMIZATION

Particle Swarm Optimization (PSO) [15] is a global optimization technique inspired by swarm behaviors found in nature (e.g., the bee colony). The goal of PSO is to find the global optimum by a group of particles (i.e., the swarm). Each particle has a velocity and a position. We define $\boldsymbol{v_i} \in \mathbb{R}^d$ and $\boldsymbol{p_i} \in \mathbb{R}^d$ to represent the velocity and position of i-th particle, where d is the dimension of the optimization problem. Each particle in the swarm has accesses to two pieces of information: (i) the direction and distance to the potential global optimum that the particle has seen locally (particle memory information) and (ii) the direction and distance to the potential global optimum which has been seen by the swarm globally (swarm information).

We define $pbest_i$ and $gbest$ (i.e., the local particle best error and global best error) to represent these two pieces of information of the i-th particle. Then, each particle incrementally updates its velocity by randomly weighting these two pieces of information and combining them (i.e., combining $pbest_i$ and $gbest$ with random weighting). Similarly, the position of each particle is essentially updated by combining $pbest_i$ and $gbest$ with random weighting. The swarm maintains a potential global optimum and the goal is for this potential global optimum to converge to the true global optimum. Figure 2 shows that update step of a particle. As shown in the figure, the update of velocity is influenced by $pbest_i$ and $gbest$ in the current iteration. Once the new velocity is updated, the particle moves to a new position with the new velocity.

Formally, PSO makes use of two key formulas: one for computing/updating the velocity of each particle and the other for computing/updating the position of each particle. Let $\boldsymbol{p_i}$ denote the current position of the i-th particle in the swarm. The update formula for the velocity denoted by $\boldsymbol{v_i}$ is given by Equation (1)

Algorithm 1: Particle Swarm Optimization

```
Input  : $n$: the number of particles;
         $\omega, c_1, c_2$: parameters of PSO;
         $max\_iter$: the number of iterations
Output : The optimal positions of the particles
// initialization of each particle
1  for $i \leftarrow 1$ to $n$ do
2      initialize the $\boldsymbol{p}_i$ and $\boldsymbol{v}_i$;
3  end
4  for $t \leftarrow 1$ to $max\_iter$ do
       // evaluation and error update
5      for $i \leftarrow 1$ to $n$ do
6          calculate current error $perror_i$;
7          if $perror_i < pbest_i$ then
8              $pbest_i \leftarrow perror_i$;
9          end
10         if $perror_i < gbest$ then
11             $gbest \leftarrow perror_i$;
12         end
13     end
       // particle update
14     for $i \leftarrow 1$ to $n$ do
15         calculate $\boldsymbol{v}_i^{'}$ with equation (1);
16         calculate $\boldsymbol{p}_i^{'}$ with equation (2);
17     end
18 end
```

below.

$$\begin{aligned}\boldsymbol{v}_i^{'} &= \omega\boldsymbol{v}_i + c_1\boldsymbol{l}_i \odot (pbest_i \cdot \boldsymbol{e} - \boldsymbol{p}_i) \\ &+ c_2\boldsymbol{g}_i \odot (gbest \cdot \boldsymbol{e} - \boldsymbol{p}_i)\end{aligned} \tag{1}$$

where ω represents the particle momentum; c_1 and c_2 represent preference to explore locally and globally, respectively; $\boldsymbol{l}_i$ and $\boldsymbol{g}_i$ are random weight vectors whose components are sampled from $U(0,1)$; $\boldsymbol{e} = [1,1,\ldots,1] \in \mathbb{R}^d$ stands for the vector with all elements of 1; $\odot$ denotes to the Hadamard product which performs the element-wise multiplication of two vectors [9].

Once the velocity of a particle is computed, the position can be updated by Equation (2) shown below.

$$\boldsymbol{p}_i^{'} = \boldsymbol{p}_i + \boldsymbol{v}_i^{'} \tag{2}$$

The progress of PSO is summarized in Algorithm 1. As can be seen in the algorithm, the process of PSO are mainly divided into three phases. The first phase is data initialization (Lines 1-3), which is responsible for initializing the position and velocity of each particle for the swarm in a random manner. The second phase is used for fitness computation and error update (Lines 5-13). Followed by that, the last step is particle update (Lines 14-17), aiming to finish the update of velocities and positions. As we can see from Algorithm 1, there are many 'for loops' which is the main reason why PSO is time-consuming. One of the characteristics of PSO is that it searches globally for the optimum, although it does not have a guarantee of convergence. In particular, initializing particles in a subspace far from the global optimum may reduce the likelihood of convergence to the global optimum, so the initialization step in PSO is crucial (Lines 1-3).

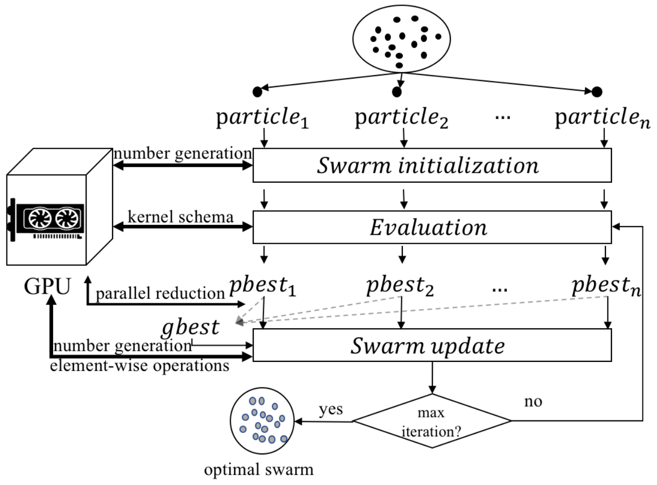


Figure 3: The overview of FastPSO

3 OUR PROPOSED FASTPSO ALGORITHM

This section elaborates the technical details of our proposed FastPSO. For ease of presentation, we divide the PSO optimization process into four steps: (i) swarm initialization, (ii) swarm evaluation, (iii) *pbest* and *gbest* update and (iv) swarm update. Figure 3 shows that overview of FastPSO. The existing methods for accelerating PSO work as follows. In each iteration of PSO, a thread for each particle is created and the thread is responsible for the whole life-cycle of the particle: the calculation of the target value of the particle, the target value update, and updating the position and velocity of the particle. However, the velocity and position vector of each particle often contain a large number of elements while the number of particles is much smaller than the number of GPU cores, which makes the existing methods inefficient, especially when handling high-dimensional data.

Different from existing methods, our proposed FastPSO algorithm exploits finer granularity of parallelism. For example, in the swarm update step, we model the procedure as element-wise multiplication operations on matrices. Based on this modeling, the velocity and the position of a particle are divided into a set of elements. Then, we can allocate the workload to a thread in the unit of elements instead of in the unit of particles. As a result, the workload allocation to each thread is more flexible, and the number of threads to be used can be much larger. Thus, our proposed FastPSO algorithm can make efficient use of the GPU resources, which leads to excellent performance when dealing with high-dimensional data.

Moreover, we propose a series of techniques to make more efficient use of the GPU resources including shared memory and tensor cores. In the *pbest* and *gbest* update step as well as the swarm update step, a large number of threads needs to be created. We develop the GPU resource aware thread creation to prevent creation of too many threads. In the swarm initialization step, we exploit parallel techniques to efficiently generate two coefficient matrices with random numbers corresponding to the velocities and positions. In the whole algorithm, we exploit GPU memory caching to manage swarm information instead of allocating new memory when needed. In the swarm evaluation step, we develop a schema to support customized swarm evaluation functions, which are automatically executed on GPUs in parallel. Next, we elaborate the details of FastPSO.

3.1 Step (i): Swarm initialization

Before the search for the global optimum starts, the PSO algorithm needs to initialize the position and velocity of each particle. This initialization step involves a large number of random number generations which can be done based on Thrust. These random number generation processes include the initialization of velocities, positions and the random weight vectors (i.e., $\boldsymbol{l}_i$ and $\boldsymbol{g}_i$ in Equation (1)) for the swarm. In each iteration of PSO, both $\boldsymbol{l}_i$ and $\boldsymbol{g}_i$ need to be generated for the i-th particle. Therefore, two $d \times n$ random number matrices need to be generated in each iteration of PSO.

3.2 Step (ii): Swarm evaluation

Given the current state of the swarm, we can evaluate its quality. Recall that the main goal of the PSO algorithm is to find the global optimum for the applications of interest. In practice, the global optimum is often unknown, since the ground truth of the given applications is unknown. Nevertheless, practitioners know how to evaluate a swarm for the given problem. To explain this, let us consider using PSO for the neural network architecture search problem as an example. The current state of the swarm in PSO corresponds to a certain neural network architecture. Given a neural network architecture the PSO has found, practitioners can evaluate the architecture to measure the metric of interest like predictive accuracy. The higher the predictive accuracy indicates the higher quality of the swarm.

The swarm evaluation is also computationally expensive, because in every iteration of PSO, a swarm evaluation procedure needs to be performed. On two popular CPU-based PSO libraries (i.e., *pyswarms* and *scikit-opt*), the implementation of the evaluation functions greatly affects the overall time consumption of PSO. If parallelism is not well exploited in the implementation of the evaluation functions, the libraries can be more than 10 times slower [19]. Thus, in order to maximize the performance of PSO on GPUs, we dedicate a GPU thread to evaluate the quality of one or more particles depending on the number of particles in the swarm. Moreover, different applications need to use different functions or metrics to evaluate a swarm. Therefore, our proposed PSO algorithm allows practitioners to customize the function/metric in the evaluation. We design a CUDA schema kernel for user-defined loss functions, which can be seen in code snapshot below.

```
template<typename L>
__global__ void evaluation_kernel(int dim, L lambda){
  for(int i = blockIdx.x * blockDim.x + threadIdx.x;
       i < dim; i += blockDim.x * gridDim.x) {
    lambda(i);
  }
}
```

In the above code snapshot, "lambda(·)" is the function that practitioners can implement by themselves and pass to FastPSO. Furthermore, FastPSO provides a series of built-in evaluation functions with GPU acceleration. The built-in functions include commonly used functions in Swarm Intelligence community, such as Sphere, Griewank and Easom [20].

3.3 Step (iii): The *pbest* and *gbest* update

After the evaluation of the swarm, the next step of the PSO algorithm is to update *pbest* and *gbest*. In the process of updating *pbest*, each particle needs to compare its new target value and its previous target value. If the new target value of the particle leads to a better swarm, the *pbest* value of the particle is updated. Otherwise, the *pbest* value of the particle stays unchanged. The comparison between the new target value and the previous target value and the update of *pbest* involve only the individual particle. Therefore, the update of *pbest* can be done in massively parallel. In FastPSO, we assign a GPU thread to each particle, and the thread is responsible for completing the particle target value comparison and the *pbest* update.

It is worth noting that the GPU thread workload during the *pbest* update process is adapted depending on the size of the optimization problem, so that the number of threads created is not exploded. Specifically, the thread workload of *pbest* update is $\frac{n}{mem}$, where n is the number of particles and mem is the available memory of the GPU. The update for *gbest* of the swarm is a process of finding the minimum and its corresponding index in all the *pbest* of the particles. We implement this update using a GPU-based parallel reduction.

3.4 Step (iv): Swarm update

The last step of the PSO algorithm is to update the velocity and position vectors of each particle (e.g., updating $\boldsymbol{v}_i$ and $\boldsymbol{p}_i$ of the i-th particle). This step is the most time-consuming component of the PSO algorithm. For example, 80% of the algorithm running time of *pyswarms* is consumed in this step [19]. Our experimental results in Section 4 confirm the percentage of the execution time during the process of the PSO algorithm is used for updating the swarms, including the position and velocity of each particle. Based on the Amdahl's law [7], if we can shorten the time of the swarm update, the overall running time of the PSO algorithm can be reduced by about 5 times theoretically. Therefore, this step needs to be carefully accelerated on the GPU foremost. In PSO, the entire swarm can be modeled as two matrices, denoted by $\mathcal{P} \in \mathbb{R}^{n \times d}$ and $\mathcal{V} \in \mathbb{R}^{n \times d}$, respectively, where $\mathcal{P}$ is the matrix containing the position information of all the particles and $\mathcal{V}$ is the matrix with the velocity information of all the particles.

In order to maximize the parallelism of the swarm update, we may assign a thread to each particle which is what the existing methods do. However, such way of workload allocation limits the degree of parallelism, because the number of particle may be much smaller than the number of GPU cores. As swarm update is the most time-consuming component of PSO, we further elaborate the parallelism granularity by modeling the update as element-wise multiplication operations on the two matrices (i.e., $\mathcal{P}$ and $\mathcal{V}$). Based on this idea, we assign the element of each dimension in the particle to a thread for updating the element at that position. The advantage of this is that it can improve the workload of each thread in the GPU and achieve good scalability. As the number of elements in the two matrices may be too large which can lead to extra cost on thread creation or leading to running out of GPU memory, FastPSO can automatically allocate multiple elements to a thread when the matrices are large. Formally, the GPU thread workload of FastPSO can be formulated as the equation below.

$$tw = \frac{n \times d}{mem} \tag{3}$$

where tw stands for the thread workload; n denotes the number of particles; d denotes the particle dimension; mem stands for the available memory of a GPU (e.g., 16GB in a Tesla V100 GPU).

Formally, the update process of the velocities on the whole swarm can be calculated in matrix representation shown in the equation below.

$$\mathcal{V}' = \omega \cdot \mathcal{V} + c_1 \cdot \mathcal{L} \odot (\mathcal{E}_l - \mathcal{P}) + c_2 \cdot \mathcal{G} \odot (\mathcal{E}_g - \mathcal{P}) \tag{4}$$

where ω, c_1 and c_2 are the same as those in Equation (1); $\mathcal{V}$ and $\mathcal{P}$ are the coefficient matrices corresponding to the velocities and positions of all the particles, respectively; $\mathcal{L}$ and $\mathcal{G}$ stand for the matrices of the randomly generated weights, where each row in the matrix $\mathcal{L}$ and $\mathcal{G}$ corresponds to the vector $\boldsymbol{l}$ and $\boldsymbol{g}$, respectively, in Equation (1); $\mathcal{E}_l$ and $\mathcal{E}_g$ are the matrices with values in each row of $pbest$ and $gbest$, respectively. The more detailed representations of $\mathcal{V}$, $\mathcal{P}$, $\mathcal{L}$, $\mathcal{G}$, $\mathcal{E}_l$ and $\mathcal{E}_g$ are shown below.

$$\mathcal{V} = \begin{bmatrix} v_{11} & \cdots & v_{1d} \\ v_{21} & \cdots & v_{2d} \\ \vdots & \ddots & \vdots \\ v_{n1} & \cdots & v_{nd} \end{bmatrix}, \quad \mathcal{P} = \begin{bmatrix} p_{11} & \cdots & p_{1d} \\ p_{21} & \cdots & p_{2d} \\ \vdots & \ddots & \vdots \\ p_{n1} & \cdots & p_{nd} \end{bmatrix},$$

$$\mathcal{L} = \begin{bmatrix} l_{11} & \cdots & l_{1d} \\ l_{21} & \cdots & l_{2d} \\ \vdots & \ddots & \vdots \\ l_{n1} & \cdots & l_{nd} \end{bmatrix}, \quad \mathcal{G} = \begin{bmatrix} g_{11} & \cdots & g_{1d} \\ g_{21} & \cdots & g_{2d} \\ \vdots & \ddots & \vdots \\ g_{n1} & \cdots & g_{nd} \end{bmatrix},$$

$$\mathcal{E}_l = \begin{bmatrix} pbest_1 & \cdots & pbest_1 \\ pbest_2 & \cdots & pbest_2 \\ \vdots & \ddots & \vdots \\ pbest_n & \cdots & pbest_n \end{bmatrix}, \mathcal{E}_g = \begin{bmatrix} gbest & \cdots & gbest \\ gbest & \cdots & gbest \\ \vdots & \ddots & \vdots \\ gbest & \cdots & gbest \end{bmatrix}, \text{ where}$$

v_{ij} and p_{ij} stand for the j-th value of the velocity and position in the i-th particle; l_{ij} and g_{ij} denote the two randomly generated weights used to update the v_{ij} in Equation (1), respectively. We dedicate a GPU thread to update an element in the velocity matrix $\mathcal{V}$. More specifically, let us take the update process of v_{11} in the matrix as an example. Initially, the dedicated GPU thread gets the values of ω, v_{11}, c_1, c_2, l_{11}, g_{11}, p_{11}, $pbest_1$ and $gbest$. Then, the element-wise calculation is performed based on the equation $v_{11}^{'} = \omega v_{11} + c_1 \cdot l_{11} \cdot (pbest_1 - p_{11}) + c_2 \cdot g_{11} \cdot (gbest - p_{11})$ which is derived based on Equation (1).

The update on the velocity has a significant impact on the convergence [2]. Therefore, the change of the velocity of a particle cannot be too large. In our proposed FastPSO solution, we exploit the bound constraint convergence [14] which can be specified by practitioners. With the bound constraint, when the velocity of each particle is updated, an upper and lower bound constraints are applied to limit the updated velocities in a specific range. The constraint for the j-th velocity value of the i-th particle is formulated below.

$$v_{ij} = \begin{cases} lower\_bound_{ij} & \text{if } v_{ij} < lower\_bound_{ij} \\ upper\_bound_{ij} & \text{if } v_{ij} > upper\_bound_{ij} \\ v_{ij} & \text{otherwise} \end{cases} \tag{5}$$

In Equation (5), $lower\_bound_{ij}$ and $upper\_bound_{ij}$ are the user defined lower and upper bounds of j-th velocity value of i-th particle, respectively. The position update of each particle shown in Equation (2) can be expressed in the matrix representation as well, which is similar to the update of velocities, so we omit its matrix representation here.

Although for GPU programs, more threads do not necessarily bring better performance. A study [30] shows that one of the most useful tricks to improve the performance of GPU programs is to increase the number of outputs and reduce the number of independent instructions per thread. As we already know, the position update is dependent on the updated velocity. Thus, the above mentioned method would increase the number of independent instructions that result in more time consumption on thread synchronization and the number of outputs is one per thread. In fact, position update depends on the updated velocity, but the update of each position element can be synchronized.

3.5 Exploiting GPU memory, tensor cores and multi-GPUs

Exploiting the shared memory and the tensor cores can potentially bring improvement in FastPSO. We make use of these GPU resources and study their effect in the overall performance of FastPSO.

Supporting shared memory: The global memory is the biggest-volume memory on GPU, and every thread on GPU is permitted to access the global memory. We can store the velocity and position vectors in the global memory in the PSO optimization process. However, the latency of global memory is the highest in all kinds of GPU memory. Thus, to increase the arithmetic intensity of our kernel, we want to reduce as many accesses to the global memory as possible. One of the most common tricks to further accelerate the GPU programs by replacing global memory with shared memory. In the step of swarm update, the size of matrices to be updated is larger than the shared memory size. Therefore, the matrices are firstly segmented into multiple sub-matrices whose shape is $(TILE\_SIZE, TILE\_SIZE)$. Then the sub-matrices are copied into the shared memory. Finally, the outputs are transferred back to the global memory after the element-wise operation

Supporting tensor cores: We also exploit tensor cores which are Nvidia's newest acceleration technique. Tensor cores enable mixed-precision computing, dynamically adapting calculations to accelerate throughput while preserving accuracy. It is specifically designed for manipulating matrices so it can be utilized for swarm update. When using tensor cores for swarm update, the element-wise matrices multiplication is regarded as the wrap-level matrix multiply. The matrices to be updated are first assigned to the fragment of tenser cores. Then the calculation is done in each fragment, which can be seen as a tiled matrix operation. Finally the outputs are copied to global memory after the tensor core synchronization.

Supporting multiple GPUs: There are two approaches to extend FastPSO to multiple GPUs. The first approach is based on particle splitting. It needs to specify a GPU card to store the global information of the whole particle groups. Then the whole particle group is split into multiple sub-groups. Each sub-group is assigned to a card for optimization, and maintains its own local-global best information. The global information of the whole group can be updated in an asynchronous manner. The second approach is based on tile matrix. It extends the element-wise swarm update operation to multiple GPUs, where each GPU is responsible for computing the updated information in a tile manner.

4 EXPERIMENTAL STUDIES

In this section, we present the experimental evaluation of our proposed FastPSO solution. We also provide a case study on the application on automatic block dimension setting with PSO for a machine learning library of GPUs, namely ThunderGBM [32].

4.1 Experimental setup

We conducted our experiments on a workstation running Linux on two Xeon E5-2640v4 10 core CPUs, 256GB main memory and one Tesla Pascal V100 GPU of 16GB memory. We implemented the FastPSO solution in CUDA-C. For better investigation of FastPSO, we have implemented the sequential version and the parallel version with OpenMP for FastPSO using C++. We denote them using *fastpso-seq* and *fastpso-omp*, respectively. All the programs were compiled with the -O3 option for efficiency optimization. We also conducted experiments for the existing implementations on CPUs including *pyswarms* [19] and *scikit-opt*[1] [23], as well as the existing implementations on GPUs namely *gpu-pso* [10] and *hgpu-pso* [31]. Among them, the *gpu-pso* is a pure-GPU implementation while the *hgpu-pso* heterogeneous multicore CPU and GPU for PSO acceleration. We compare FastPSO with *pyswarms* and *scikit-opt*, because these two systems are the most popular open-source PSO libraries in GitHub, which get more than 700 and 1700 stars respectively. In all of our experiments, unless specified, the default number of particles and dimensions are 5000 and 200, respectively; the maximum number of iterations of PSO is set to 2,000; the ω is set to 0.9; c_1 and c_2 are both set to 2. All experiments were repeated 10 times and the experimental data are the averages.

We evaluate FastPSO and the other implementations presented above in the following four optimization problems.

- **Sphere**: This problem arises from linear algebra and traffic routing, where users need to find the minimum of a given function in the form of $f(\boldsymbol{x}) = \sum_{i=1}^{d} x_i^2$. In our experiments, we used PSO to find the minimum for the Sphere problem with x in the domain of $(-5.12, 5.12)$.
- **Griewank**: The Griewank problem is used for performance test of optimization algorithms. The problem is to find the minimum of the function $f(\boldsymbol{x}) = \frac{1}{4000}\sum_{i=1}^{d} x_i^2 - \prod_{i=1}^{d}\cos(\frac{x_i}{\sqrt{i}})$ $+1$. We used PSO to locate the minimal value of the Griewank problem given x in the domain of $(-600, 600)$.
- **Easom**: The Easom problem is a non-convex function used as a performance test problem for optimization algorithms. The target function is $f(\boldsymbol{x}) = -(-1)^d(\prod_{i=1}^{d}\cos^2(x_i))\exp[-\sum_{i=1}^{d}(x_i - \pi)^2]$ with x in the domain of $(-2\pi, 2\pi)$.
- **ThreadConf**: We used PSO to find the best thread and block configuration for a GPU machine learning library namely ThunderGBM. We aim to compare the default thread and block configuration with the configuration obtained by PSO, and investigate whether PSO can find a better configuration than the default one.

[1]https://github.com/guofei9987/scikit-opt

Next, we first present the experimental results on the overall efficiency of the various implementations. Then, we provide results on sensitivity study by varying the number of particles and by varying the number of dimensions of the optimization problems. We also investigate the breakdown of the elapsed time on different components of FastPSO, in order to provide a better insight of its efficiency and speedup. Finally, we demonstrate the effectiveness of FastPSO by a case study on the ThreadConf problem.

4.2 Overall comparison

In the overall evaluation of FastPSO, we compare the execution time of FastPSO, the sequential version of FastPSO, the parallel version of FastPSO with OpenMP, pyswarms, scikits-opt and *gpu-pso* as well as *hgpu-pso* (i.e., the existing GPU-based PSO systems [10, 31]). The experimental results are shown in Table 1. As can be seen from the "speedup" columns of the table, our proposed FastPSO achieves about 100 times speedup over *pyswarms* and *scikit-opt*. Compared with the existing GPU implementations of PSO, FastPSO is 5 to 7 times faster. When comparing FastPSO with its sequential implementation and OpenMP implementation, FastPSO is an order of magnitude faster. Another observation on the sequential version of FastPSO and the OpenMP version of FastPSO is that the execution time of the sequential version can be reduced by about 50% using OpenMP.

We also evaluate the results found by different PSO implements. We compare the values found by FastPSO and other implementations against the optimal value of the problems. The results are shown in Table 2. As we can see from the table, FastPSO can find the values extremely close to the optimal ones, while the values found by *pyswarms* and *scikit-opt* are much further away from the optimal ones. These results demonstrate that FastPSO is not only faster than the existing libraries, but also produces much higher quality solutions.

In order to get a better insight of our proposed method, we evaluate the FLOPs and memory bandwidth of our method. As shown in Table 3, compared with other implementations (i.e., *gpu-pso* and *hgpu-pso*), the memory bandwidth of our method (which is over 100 GB/s) is much higher than the benchmark GPU implementations. In terms of FLOPs, all the implementations are based on the original PSO algorithm. As a result, the FLOPs of each implementation is similar.

4.3 Effect of the number of particles and dimensions on efficiency

In order to study the scalability of FastPSO, a series of experiments on the four problems were carried out by changing the number of particles and the dimensions of particles.

Table 1: Overall comparison of FastPSO against other implementations

problem	elapsed time (sec)							speedup					
	pyswarms	scikit-opt	gpu-pso	hgpu-pso	fastpso-seq	fastpso-omp	fastpso	pyswarms	scikit-opt	gpu-pso	hgpu-pso	fastpso-seq	fastpso-omp
Sphere	129.67	88.98	4.90	6.01	11.56	8.74	0.67	194.41	133.40	7.34	9.01	17.33	13.10
Griewank	80.94	172.17	5.08	7.32	13.78	9.58	0.66	123.38	262.46	7.74	11.16	21.00	14.60
Easom	126.89	12.77	5.07	7.22	33.91	24.71	0.87	146.35	14.72	5.85	8.33	39.11	28.50
TheadConf	117.670	81.320	4.498	5.477	11.459	6.736	0.47	251.97	174.13	9.63	11.73	24.54	14.42

Table 2: Errors to the optimal values

implementation	reference	Sphere	Griewank	Easom
pyswarms	Miranda [19]	1031.99	2965.27	0.00
scikit-opt	Pedregosa et al. [23]	2483.61	8892.36	0.00
gpu-pso	Hussain et al. [10]	23.72	0.69	0.00
hgpu-pso	Wachowiak et al. [31]	15.06	0.31	0.00
fastpso-seq	ours	26.98	0.66	0.00
fastpso-omp	ours	22.01	0.72	0.00
fastpso	ours	23.62	0.71	0.00

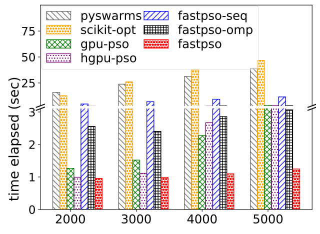
Table 3: FLOPs and memory bandwidth

metrics	dram_read_throughtput (GB/s)	GFLOPs
gpu-pso	61.83	5.82
hgpu-pso	57.41	5.81
fastpso	106.94	5.82

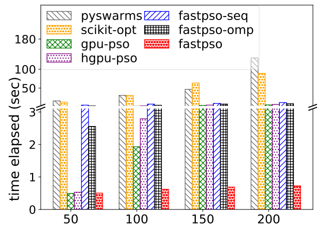
Varying the number of particles: We varied the number of particles from 2000 to 5000, while fixing the number of dimensions at 50. Figure 4a, 4c, 4e and 4g show the effect of the number of particles on the efficiency of FastPSO and other implementations among four problems. As we can see from the figures, FastPSO consistently outperforms the other implementations in different numbers of particles. Moreover, increment of the number of particles greatly increases the execution time of the other implementations, while FastPSO tends to be stable (i.e., the elapsed time is almost unchanged). This demonstrates that FastPSO is suitable to problems with a large number of particles.

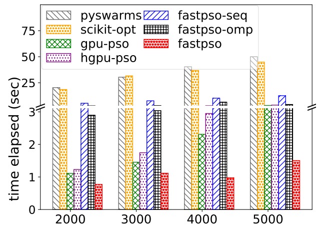
Varying the number of dimensions of the particles: When conducting this set of experiments, we varied the dimensions of the particles from 50 to 200, while fixing the number of particles to 2000. Figures 4b, 4d, 4f and 4h give the results. When varying the dimensions of the particles, FastPSO is rather stable as the number of dimensions of the particles increases. The time consumption of FastPSO is around 0.5 seconds. In comparison, the execution time of the other implementations increases dramatically as the number of dimensions increases. This is due to the fact that when the number of dimensions increases, the number of values in the velocity/position vector to be updated within the swarm increases. As a result, the computation cost increases. For example, it takes *pyswarms* more than 100 seconds to finish the optimization process when the number of dimensions reaches 200.

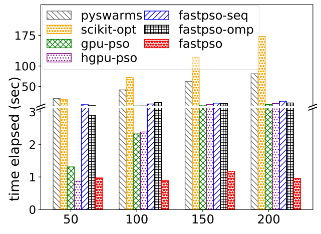
Based on this set of experiments, we demonstrate that FastPSO can not only far exceed other implementations in efficiency, but also scale to large problems with high dimensionality and cardinality. This is manifested in changing the number of particles and dimensions while keeping the total execution time almost unchanged.

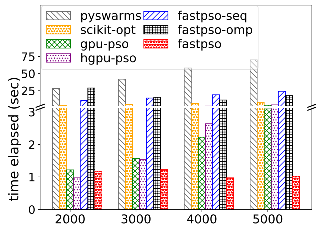


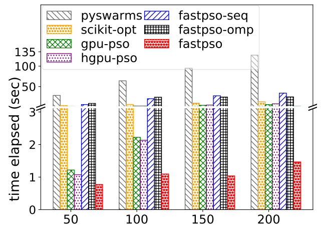
(a) varying # particles (Sphere)

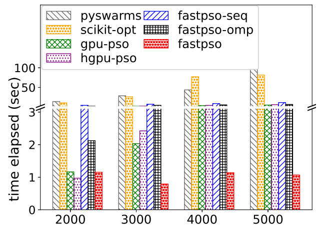

(b) varying #dimensions (Sphere)


(c) varying #particles (Griewank)


(d) varying #dimensions (Griewank)


(e) varying #particles (Easom)


(f) varying #dimensions (Easom)


(g) varying #particles (ThreadConf)

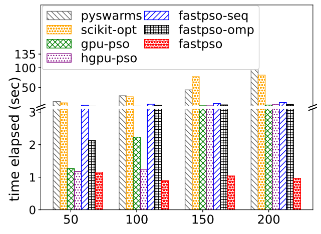
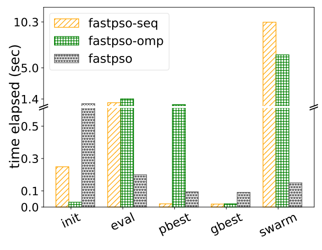
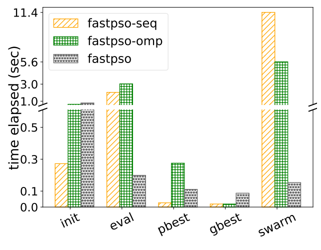

(h) varying #dimensions (ThreadConf)

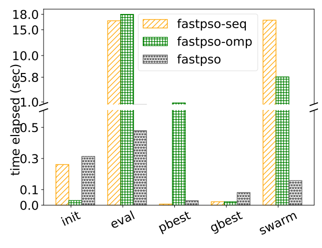
Figure 4: Effect of the number of particles and dimensions

4.4 Elapsed time of each step in FastPSO

In order to investigate more details of the efficiency of FastPSO, we conducted more experiments on measuring the execution time of each step in FastPSO. In this set of experiments, the number of the particles was set to 5000 and the number of dimensions was set


(a) breakdown tests (Sphere)


(b) breakdown tests (Griewank)


(c) breakdown tests (Easom)

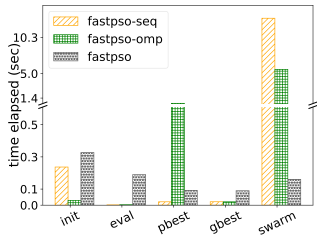

(d) breakdown tests (ThreadConf)

Figure 5: Breakdown tests of FastPSO

to 200. The results are presented in Figures 5a, 5b, 5c and 5d. We decompose the PSO algorithm into 5 steps (cf. Section 3) including (i) swarm initialization, (ii) swarm evaluation, (iii) *pbest* and *gbest* update and (iv) swarm update. The steps are tagged as *init*, *eval*, *pbest*, *gbest* and *swarms* in the figure, where *pbest* and *gbest* update were measured individually. It can be seen that more than 80% of the execution time of the CPU-based FastPSO (both the sequential version and OpenMP version) is taken by the swarm update. It takes the sequential version of FastPSO more than 10 seconds to finish the swarm update process, while FastPSO on GPUs only takes less than 0.1 second. The results and observations on four problems are consistent. Therefore, we can conclude that the key reason why FastPSO can speed up over other implementations is the optimization of swarm update process which is modeled as element-wise multiplication operations on matrices and can be done in massively parallel. In addition, the results also indicate that different optimization problems may slightly affect the execution time of FastPSO.

In order to manage and reuse GPU memory in a better manner, FastPSO exploits memory caching rather than reallocating GPU memory when managing swarm memory. The core idea of our GPU memory management is to allocate a large fixed-size memory on the GPU at the first time of memory allocation. Then when new GPU memory is need to be allocated later, the memory manager can direct the memory allocation request to the earlier allocated memory. Table 4 shows the efficiency of using memory reallocation and using memory caching. As can be seen from the table, the use of the memory caching can further improve FastPSO efficiency by 3.7% to 5% times.

4.5 Different optimizations on swarm update

As we mentioned earlier, swarm update is the key bottleneck of the PSO efficiency. In this subsection, we study different techniques that are used for swarm update acceleration. The techniques include OpenMP, global memory on GPU, shared memory and tensor cores. We used the "for-loop" based implementation on CPUs as the baselinAtificiale in this set of experiments. The results are shown in Figure 6. The elapsed time of the "for-loop" based implementation on CPUs is more than 10 seconds in four problems, while it is reduced to less than 0.3 second by using GPU acceleration techniques. The observation of the results also indicates that the efficiency improvement of using global memory, shared memory and tensor cores are similar in comparison with the baseline.

Table 4: Efficiency of FastPSO with memory caching

problem	w/ caching	w/ reallocation	speedup
Sphere	0.62	0.59	5.08%
Griewank	0.66	0.63	4.76%
Easom	0.84	0.81	3.70%

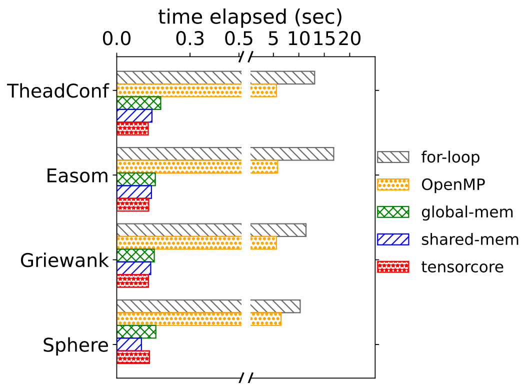


Figure 6: Comparison of different swarm update techniques

4.6 Case study on thread configuration for GPU programs

To verify the effectiveness of FastPSO in finding the best value for practical problems, we conducted a case study that applies FastPSO to automatically tune the numbers of threads and block size of a GPU program. In the CUDA programming model on GPUs, the GPU scheduler manages parallelism at the thread block level. Each GPU kernel function needs to configure the number of blocks to be used as well as the block size, so as to determine how many threads to be used. Generally, different thread configuration leads to different efficiency improvement of a GPU kernel function [25]. In this case study, we used ThunderGBM [32] which is a GPU-based machine learning library as the test GPU program for FastPSO. We set the number of trees to 40 and the tree depth to 6 in ThunderGBM and the other parameters used were the default settings of ThunderGBM. We used FastPSO to automatically set the number of threads for 25 GPU kernel functions of ThunderGBM. Each GPU kernel function needs to set the dimensions of blocks and the number of blocks to determine the total number of threads. In the context of PSO, to configure the GPU kernels, the number of dimensions of the velocity vector and the position vector of each particle in PSO for

ThunderGBM was 50. On the other hand, the number of particles used to search the best thread configuration of ThunderGBM can be any number. In our case study, we used 5000 particles in FastPSO, although other numbers of particles can work just fine. For training ThunderGBM, we used the datasets presented in the original paper which can be found and downloaded in the UCI data repository[2]. The statistical information of the datasets and the experimental results are shown in Table 5.

Table 5: Execution time w/ and w/o FastPSO

data sets			performance		
	# card	# dim	tgbm	tgbm+pso	speedup
covtype	0.58M	54	0.9	0.9	0.96
susy	5M	18	5.6	4.72	1.19
higgs	11M	28	14.51	13.94	1.04
e2006	16K	150361	7.37	5.88	1.25

As we can see from the results in Table 5, we can find that FastPSO can further optimize the training efficiency of ThunderGBM on the three datasets *higgs*, *susy* and *e2006*, by using better thread block configurations for the GPU kernels. However, we notice that using FastPSO in ThunderGBM on the *covtype* dataset leads to the same efficiency as the original ThunderGBM. This is because the default thread configuration of ThunderGBM is as good as the thread configuration found by FastPSO. Overall, FastPSO can help the tested GPU program set a better thread configuration.

5 RELATED WORKS

Swarm Intelligence usually requires a large number of function evaluations to achieve convergence, even in medium and certain small size problems. Since population based swarm intelligence methods are highly parallelizable, many existing algorithms exploit different types of available parallel architectures such as CPUs and GPUs. In this section, we review the related works of parallelized PSO and briefly discuss how the existing studies accelerate the execution time of PSO.

5.1 Acceleration of PSO on CPUs

A CPU-based parallelized PSO was first introduced by Gies and Yahya [6]. They implemented the algorithm containing a swarm with 10 particles, and hence the algorithm ran on 10 independent node cluster. The algorithm converges 8 times faster than its serial counterpart, when 10 computers nodes were used. Further, Schutte et al. [26] implemented a master-worker communication model, in which the master node exclusively performs the algorithm operations and the worker nodes perform the particle evaluation. The communications in the model are achieved with MPI. These studies mentioned above are based on synchronous communications. Another synchronous PSO implementation was proposed by Chusanapiputt et al. [3]. The implementation organizes the swarm into multiple sub-swarms in a hierarchical structure. The velocity update of particles was performed synchronously within the sub-swarms. In this strategy, after exploring all the neighborhoods in the sub-swarm, the sub-swarm sends its best position and corresponding velocity to the high-level sub-swarms in the hierarchical structure. Finally, a subset of the best velocities from the sub-swarms is selected by the root swarm, and the next move of the optimization is decided accordingly by the root swarm.

A few asynchronous versions of parallel PSO were introduced such as asynchronous parallel PSO algorithms [16, 27, 29]. The asynchronous versions of parallel PSO aim to improve the performance of the synchronous PSO by reducing the communication cost. In synchronous implementations, each particle in the parallel process waits for all the particles to complete the process before moving to the next iteration. In asynchronous implementations, the particles do not need to wait for other particles to complete the process. Therefore, no idle CPU processors are left during the process, and the parallel speed are greatly improved in asynchronous versions. McNabb et al. [18] developed PSO on the MapReduce parallel programming model in Hadoop.

5.2 Acceleration of PSO on GPUs

The development of GPU technologies provided an efficient way to accelerate PSO system in the last decade. A survey [1] provides a complete picture of parallel swarm intelligence algorithms and a thorough review of literature up to year 2013. In a more recent survey [28] reviewed a large number of contributions on Swarm Intelligence, particularly algorithms implemented on GPUs, which covers various of Swarm Intelligence methods besides PSO, including Ant Colony Optimization and Bee Algorithms. The survey also describes the implementation issues and discusses the classical performance metrics. Since the contributions on parallelized implementations of Swarm Intelligence are overwhelming, only the most related studies are mentioned in the rest of this section. A special emphasis is devoted to studies that involve PSO acceleration on GPUs. Specifically, Laguna et al. [17] proposed three parallel variants for the PSO algorithm with different degrees of parallelization.

Mussi et al. [21] presented two different parallel implementations of the PSO algorithm: basic and multi-kernels. The first implementation allows the use of a threads block to simulate a swarm, which is efficient from a memory use standpoint but does not fully exploit the GPU resources. The multi-kernel version uses several blocks to simulate several swarms simultaneously, but requires more information exchange at a memory level. Hence, the communication cost is higher compared to the first implementation. Performance was investigated with typical bound constrained benchmark functions with three different kinds of particle number and the results showed better better efficiency comparing with previous implementations. Roberge et al. [24] presented a fully optimized implementation on the GPU. The experimental results showed impressive speedups on the benchmark functions and on an application of interest to modern military aviation. In contrast to previous approaches, they focused on how to utilized GPUs to compute the objective function. Besides, every key step of the standard PSO algorithm is parallelized in this implementation. More recently, Hussain et al. [10] proposed a new GPU-based implementation of PSO. The authors used the coalescing memory in GPUs to further accelerate PSO. The experimental results on different benchmark functions show the efficiency of the implementation compared with the CPU-based

[2]https://archive.ics.uci.edu/ml/datasets.php

implementation. Hussain et al.'s PSO algorithm on GPUs achieves the state-of-the-art results. Therefore, we use it as a baseline in our experiments.

6 CONCLUSION AND FUTURE WORK

GPU accelerations have become a fundamental research topic for improving the efficiency of machine learning and artificial intelligence algorithms. This paper develops a novel GPU-based parallel Particle Swarm Optimization (PSO) algorithm namely "FastPSO" to speed up the optimization process of the PSO algorithm, which is a key Swarm Intelligence algorithm and has been widely used in many applications. Although GPUs have much higher computational power and memory bandwidth than CPUs, it is non-trivial to fully exploit GPUs for the PSO algorithm. In this paper, we have managed to use GPUs to address the efficiency bottleneck of the PSO algorithm. Our key idea is by treating the optimization process as element-wise multiplication operations on matrices and by increasing the utilization of the GPU resources. We have conducted a series of experiments to fully test the efficacy of FastPSO. The experimental results show that FastPSO can achieve the best speed on common optimization problems comparing with the existing implementations. FastPSO is 5 to 7 times faster than the existing GPU-based algorithm, and is two orders of magnitude faster than the existing CPU-based libraries. To further investigate the speedup of FastPSO, we have implemented the sequential version of FastPSO and the parallel version of FastPSO with OpenMP. Our study has shown that FastPSO on the GPU is an order of magnitude faster than the CPU-based versions, which indicates our efficient use of the GPU resources. As the new hardware (e.g., FPGA) and optimization technologies on GPUs (e.g., tensor core) are gradually emerging, we aim to use more new hardware features to optimize the PSO algorithm in the future.

7 ACKNOWLEDGMENTS

This work was supported by Project 61902333 supported by National Natural Science Foundation of China, by the Shenzhen Institute of Artificial Intelligence and Robotics for Society (AIRS). This research is also supported by Oracle for Research, Australia.

REFERENCES

[1] Enrique Alba, Gabriel Luque, and Sergio Nesmachnow. 2013. Parallel metaheuristics: recent advances and new trends. *International Transactions in Operational Research* 20, 1 (2013), 1–48.

[2] Emilio Fortunato Campana, Matteo Diez, Giovanni Fasano, and Daniele Peri. 2013. Initial particles position for PSO, in bound constrained optimization. In *International Conference in Swarm Intelligence*. Springer, 112–119.

[3] Songsak Chusanapiputt, Dulyatat Nualhong, Sujate Jantarang, and Sukumvit Phoomvuthisarn. 2005. Relative velocity updating in parallel particle swarm optimization based lagrangian relaxation for large-scale unit commitment problem. In *TENCON 2005-2005 IEEE Region 10 Conference*. IEEE, 1–6.

[4] Ashraf Darwish, Aboul Ella Hassanien, and Swagatam Das. 2020. A survey of swarm and evolutionary computing approaches for deep learning. *Artificial Intelligence Review* 53, 3 (2020), 1767–1812.

[5] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. 2006. Ant colony optimization. *IEEE Computational Intelligence Magazine* 1, 4 (2006), 28–39.

[6] Dennis Gies and Yahya Rahmat-Samii. 2003. Reconfigurable array design using parallel particle swarm optimization. In *IEEE Antennas and Propagation Society International Symposium. Digest. Held in conjunction with: USNC/CNC/URSI North American Radio Sci. Meeting (Cat. No. 03CH37450)*, Vol. 1. IEEE, 177–180.

[7] John L Gustafson. 1988. Reevaluating Amdahl's law. *Commun. ACM* 31, 5 (1988), 532–533.

[8] Hashim A Hashim and Mohammad A Abido. 2019. Location management in LTE networks using multi-objective particle swarm optimization. *Computer Networks* 157 (2019), 78–88.

[9] Roger A Horn. 1990. The hadamard product. In *Proc. Symp. Appl. Math*, Vol. 40. 87–169.

[10] Md Maruf Hussain, Hiroshi Hattori, and Noriyuki Fujimoto. 2016. A CUDA implementation of the standard particle swarm optimization. In *2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 219–226.

[11] Jing Jiang, Fei Han, Qinghua Ling, Jie Wang, Tiange Li, and Henry Han. 2020. Efficient network architecture search via multiobjective particle swarm optimization based on decomposition. *Neural Networks* 123 (2020), 305–316.

[12] Francisco Erivaldo Fernandes Junior and Gary G Yen. 2019. Particle swarm optimization of deep neural networks architectures for image classification. *Swarm and Evolutionary Computation* 49 (2019), 62–74.

[13] Dervis Karaboga. 2010. Artificial bee colony algorithm. *Scholarpedia* 5, 3 (2010), 6915.

[14] Massimiliano Kaucic. 2013. A multi-start opposition-based particle swarm optimization algorithm with adaptive velocity for bound constrained global optimization. *Journal of Global Optimization* 55, 1 (2013), 165–188.

[15] James Kennedy and Russell Eberhart. 1995. Particle swarm optimization. In *Proceedings of ICNN'95-International Conference on Neural Networks*, Vol. 4. IEEE, 1942–1948.

[16] Byung-Il Koh, Alan D George, Raphael T Haftka, and Benjamin J Fregly. 2006. Parallel asynchronous particle swarm optimization. *Internat. J. Numer. Methods Engrg.* 67, 4 (2006), 578–595.

[17] Gerardo A Laguna-Sánchez, Mauricio Olguín-Carbajal, Nareli Cruz-Cortés, Ricardo Barrón-Fernández, and Jesús A Álvarez-Cedillo. 2009. Comparative study of parallel variants for a particle swarm optimization algorithm implemented on a multithreading GPU. *Journal of Applied Research and Technology* 7, 3 (2009), 292–307.

[18] Andrew W McNabb, Christopher K Monson, and Kevin D Seppi. 2007. Parallel PSO using MapReduce. In *2007 IEEE Congress on Evolutionary Computation*. IEEE, 7–14.

[19] Lester James Miranda. 2018. PySwarms: a research toolkit for particle swarm optimization in Python. *Journal of Open Source Software* 3, 21 (2018), 433.

[20] Marcin Molga and Czesław Smutnicki. 2005. Test functions for optimization needs. *Test Functions for Optimization Needs* 101 (2005), 48.

[21] Luca Mussi, Fabio Daolio, and Stefano Cagnoni. 2011. Evaluation of parallel particle swarm optimization algorithms within the CUDA-TM architecture. *Information Sciences* 181, 20 (2011), 4642–4657.

[22] Anand Nayyar and Nhu Gia Nguyen. 2018. Introduction to swarm intelligence. *Advances in Swarm Intelligence for Optimizing Problems in Computer Science* (2018), 53–78.

[23] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of Machine Learning Research* 12 (2011), 2825–2830.

[24] Vincent Roberge and Mohammed Tarbouchi. 2012. Parallel particle swarm optimization on graphical processing unit for pose estimation. *WSEAS Trans. Comput* 11, 6 (2012), 170–179.

[25] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 73–82.

[26] JF Schutte, BJ Fregly, RT Haftka, and AD George. 2003. *A parallel particle swarm optimizer.* Technical Report. FLORIDA UNIV GAINESVILLE MECHANICAL AND AEROSPACE ENGINEERING.

[27] Andrea Serani, Cecilia Leotardi, Umberto Iemma, Emilio F Campana, Giovanni Fasano, and Matteo Diez. 2016. Parameter selection in synchronous and asynchronous deterministic particle swarm optimization for ship hydrodynamics problems. *Applied Soft Computing* 49 (2016), 313–334.

[28] Ying Tan and Ke Ding. 2015. A survey on GPU-based implementation of swarm intelligence algorithms. *IEEE Transactions on Cybernetics* 46, 9 (2015), 2028–2041.

[29] Gerhard Venter and Jaroslaw Sobieszczanski-Sobieski. 2006. Parallel particle swarm optimization algorithm accelerated by asynchronous evaluations. *Journal of Aerospace Computing, Information, and Communication* 3, 3 (2006), 123–137.

[30] Vasily Volkov. 2010. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC*, Vol. 10. San Jose, CA, 16.

[31] Mark P Wachowiak, Mitchell C Timson, and David J DuVal. 2017. Adaptive particle swarm optimization with heterogeneous multicore parallelism and GPU acceleration. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 2784–2793.

[32] Zeyi Wen, Hanfeng Liu, Jiashuai Shi, Qinbin Li, Bingsheng He, and Jian Chen. 2020. ThunderGBM: Fast GBDTs and Random Forests on GPUs. *Journal of Machine Learning Research* 21, 108 (2020), 1–5.