GPU-based Parallel Single and Multi-objective Particle Swarm Optimization for Large Swarms and High Dimensional Problems

PhD Thesis

GPU-based Parallel Single and Multi-objective Particle Swarm Optimization for Large Swarms and High Dimensional Problems

By
Md Maruf Hussain

Supervisor
Noriyuki Fujimoto

February 2020
Department of Mathematics and Information Sciences
Graduate School of Science
Osaka Prefecture University

Abstract

The social learning process of birds and fishes inspired the development of the heuristic Particle Swarm Optimization (PSO) search algorithm. The advancement of Graphics Processing Units (GPU) and the Compute Unified Device Architecture (CUDA) platform plays a significant role to reduce the computational time in search algorithm development. This doctoral thesis paper presents a good implementation for the Standard Particle Swarm Optimization (SPSO) on a GPU based on the CUDA architecture, which uses coalescing memory access. Here, we also present an analysis of the performance of the various Pseudorandom Number Generators (PRNGs) on a GPU SPSO on the CUDA architecture. The algorithm is evaluated on a suite of well-known benchmark optimization functions. The experiments are performed on an NVIDIA GeForce GTX 980 GPU and a single core of 3.20 GHz Intel Core i5 4570 CPU and the test results demonstrate that the GPU algorithm runs about maximum 170 times faster than the corresponding CPU algorithm. The success of the PSO algorithm as a single objective optimizer has motivated us to extend its use in multi-objective optimization problems (MOOPs). During the last couple of years, parallel MOPSO (Multi-objective Particle Swarm Optimization) with two or more objectives has gained a lot of attention in the literature on GPU computing. A number of implementations have been published for MOPSO on a GPU. However, none of them have been able to capture good enough Pareto fronts fast. In addition, the authors have pointed out their limitations in various aspects such as archive handling, picking up fewer nondominated solutions and so on. Previous literature also lacks evaluation of its MOPSO implementation with large swarms and high dimensional problems. This thesis paper presents a faster implementation of parallel MOPSO on a GPU based on the CUDA architecture. We achieved our faster implementation by using coalescing memory access, a fast pseudorandom number generator, Thrust library, CUB library, an atomic function, parallel archiving and so on. The proposed parallel implementation of MOPSO using a master-slave model provides up to 157 times speedup compared to the corresponding CPU implementation.

As the proposed implementations perform very highly even with increased size of problem dimensionality and swarm population, it can be widely used in real world optimization problems.

Acknowledgement

Foremost, I would like to express my sincere gratitude to my supervisor Prof. Noriyuki Fujimoto for the continuous support of my Ph.D study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Yushi Uno and Prof. Kazuhisa Seta, for their encouragement, insightful comments, and hard questions.

In particular, I am grateful to my labmats Hiroyuki Kobayashi and Hiroshi Hattori for enlightening me the first glance of research.

Last but not the least, I would like to thank my family: my parents Md Omar Ali and Mrs Nurun Nahar, my brother Md Obaej Tareq, my sister Shohelly Akther Shampa and my wife Mahfuza Sharmin for supporting me spiritually throughout my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Particle Swarm Optimization (PSO) algorithm has been first introduced by Eberhart and Kennedy in 1995 [1], which is one of the most important population based non-deterministic optimization algorithms for single objective optimization problems. Since then, many successful applications of PSO have been reported. In many of those applications, the PSO algorithm has shown several advantages over other swarm intelligence based optimization algorithms due to its robustness, efficiency and simplicity. Moreover, compared to other stochastic algorithms, it usually requires less computational effort and resources [2].

The PSO algorithm maintains a swarm of particles, where each of which represents a potential solution. Here a swarm can be identified as the population and a particle as an individual. In a PSO system, each particle flows through a multidimensional search space and adjusts its position based on its own experience with neighboring particles.

On a CPU, this process is implemented based on task scheduling into serial processing, whereas on a GPU, many particles can reach to their positions simultaneously, which improves the PSO efficiency significantly. In recent years, a GPU becomes a very popular platform for the realization of parallel computing, mainly due to changes in architecture and development of CUDA and OpenCL languages. Previously reported works have shown that the PSO implementation on a GPU provides a better performance than CPU-based implementations [3] which makes us interested in this study.

At first, we implemented a good implementation for the Standard Particle Swarm Optimization (SPSO) on a GPU based on the CUDA architecture, which uses atomic function, a fast pseudorandom number generator, coalescing memory access. The algorithm is evaluated on a suite of well-known benchmark optimization functions. The experiments are performed on an NVIDIA GeForce GTX 980 GPU and a single core of 3.20 GHz Intel Core i5 4570 CPU and the test results demonstrate that the GPU algorithm runs about maximum 170 times faster than the corresponding CPU algorithm [4]. Therefore, this proposed algorithm can be used to improve required

time to solve optimization problems. After that, we conducted experiments for testing the effect of the Pseudorandom Number Generators on the SPSO on a GPU [5]. By using a single step `TausStep` of the combined Tausworthe generator, the proposed parallel implementation of SPSO provides up to 307 times speedup compared to a serial SPSO implementation. Speedup is greatly accelerated for high dimensional problems, large particles and complex benchmark functions.

The success of the PSO algorithm as a single objective optimizer has motivated us to extend its use in other areas. One of such areas is multi-objective optimization. Multi-objective optimization problems (MOOPs) are very common in real-world optimization fields, where the objectives to be optimized are normally in conflict with each other. Moore and Chapman proposed the first extension of the PSO strategy for solving MOOPs (Multi-Objective PSO, MOPSO) in an unpublished manuscript in 1999 [6].

With big data becoming more important as time goes by, the necessity for faster methods is growing [7]. The previous implementations of serial and parallel cases of MOPSO do not meet the requirements of big data. In addition, those implementations could only handle a limited number of dimensions. The necessity for a better method is sorely needed.

This thesis paper presents a new GPU-parallelized implementation of MOPSO (GPU MOPSO) based on a master-slave model for large swarms and high dimensional optimization problems. This paper also presents a new serial implementation of MOPSO (CPU MOPSO). Our CPU program uses a single core only although our CPU has multiple cores. Our CPU MOPSO achieves faster performance for large swarms and high dimensional optimization problems. The experimental results show that the proposed GPU MOPSO increases the processing speed compared to previously proposed approaches on a GPU based on the CUDA architecture. The proposed parallel implementation of MOPSO using a master-slave model provides up to 157 times speedup compared to the corresponding CPU implementation. Here, we investigate a large number of iterations to reach good nondominated solutions which achieve good Pareto fronts. Pareto fronts of both CPU MOPSO and GPU MOPSO implementations match very closely to the true Pareto fronts.

Performance of MOPSO is dependent upon an archiving technique. We propose a simple parallel archiving technique which significantly speeds up the process. Our serial archiving technique is the same as the parallel archiving except that it is executed in serial. In our GPU MOPSO, the used PRNG and coalescing memory access have a positive impact which improve computational time.

In the literature, several models for parallel MOPSO have previously been proposed. Some of these models are suited for costly platforms. For example, the island model is suitable for clusters and grids. The diffusion model of multi-objective evolutionary algorithms is also suitable for another costly platform, massively parallel processors. In contrast, there are models for more affordable platforms. The master-slave model [8] and the hierarchical model are both suitable for GPUs, but

the hierarchical model tends to be slower than the master-slave model [9].

The doctoral thesis is organized as follows. In Chapter 1, we introduce our implementations. In Chapter 2, we sketch out briefly PSO, SPSO, MOOPs, MOPSO, GPU computing, an overview of CUDA architecture coalescing memory access, random number generators, Thrust library, CUB library and parallel models. In Chapter 3, we present our CUDA Implementation of SPSO. In Chapter 4, we conduct experiments for testing the effect of ten Pseudorandom Number Generators on the SPSO on a GPU. In Chapter 5, we provide our MOPSO implementations on a CPU and a GPU, analyze experimental results and compare our implementation with the previous implementation in terms of execution time and speedup. Finally, in Chapter 6, we give some concluding remarks and point out directions for future work.

Chapter 2

Background Information

2.1 The Particle Swarm Optimization (PSO)

A PSO system simulates the behaviors of a bird flock. In a bird flock, some birds are randomly searching food in an area [10]. If there is only one piece of food in the area, all the birds do not know the exact location of the food. However, with each iteration, they know how far the food is. For an individual bird the best effective way to find the food source is to follow the bird which is nearest to the food.

A PSO algorithm follows the similar principle, where it learns from the past scenario and uses it to solve the optimization problems. Compared to the bird scenario, each single solution in PSO is a bird in the search space. We call it a *particle*. In our paper, the number of particles is represented by n and the number of dimensions is represented by d. All particles are evaluated by their *fitness values* which are evaluated by the *fitness function* to be optimized, and each particle has a velocity to direct the flying of the particle.

Initially, a PSO starts with a group of random particles (solutions). Each particle then searches for optima by updating *generations*. Each particle is updated by following two "best" values in every iteration. The first best value is the *personal best* which is the best fitness value it has achieved so far and called as *pBest*. The second "best" value is the value that is obtained by any particle in the population. This best value is a *global best* and called *gBest*.

In a PSO, the position and velocity of a particle within the domain of the fitness function are computed by two following equations:

Velocity update :

$$\begin{aligned} v_{ij}(t+1) = w(&v_{ij}(t) \\ &+c_1 * rand_1() * (pBestPos_{ij} - x_{ij}(t))) \\ &+c_2 * rand_2() * (gBestPos_j - x_{ij}(t)))) \end{aligned}$$

Position update:

$$x_{ij}(t+1) = x_{ij}(t) + v_{ij}(t+1)$$

where i is the identifier of a particle and t is time in generation. $v_{ij}(t)$ is the jth component of the current velocity vector and $x_{ij}(t)$ is the jth component of the current position vector of the ith particle. $pBestPos_{ij}$ is jth component of the personal best position of the ith particle and $gBestPos_j$ is jth component of the position of the global best. $v_{ij}(t+1)$ is the jth component of modified velocity and $x_{ij}(t+1)$ is the jth component of updated position of the ith particle. $rand_1()$ and $rand_2()$ are random real numbers in [0, 1]. c_1 and c_2 are constants and w is also a constant called *inertia weight*. $rand_1()$ and $rand_2()$ are independent each other. Moreover, $rand_1()$ and $rand_2()$ are independent among particles respectively. Therefore, many independent random sequences are required.

A particle as well as its topological neighbors is considered as part of the population. In that case, the best value in the part is a local best and is called *lBest* [10]. In a PSO lBest version, the position and velocity of a particle within the domain of the fitness function are computed by two following equations:

Velocity update:

$$\begin{aligned} v_{ij}(t+1) = w(&v_{ij}(t) \\ &+c_1 * rand_1() * (pBestPos_{ij} - x_{ij}(t))) \\ &+c_2 * rand_2() * (lBestPos_j - x_{ij}(t)))) \end{aligned}$$

Position update:

$$x_{ij}(t+1) = x_{ij}(t) + v_{ij}(t+1)$$

where $lBestPos_j$ is jth component of the local best position.

2.2 The Standard Particle Swarm Optimization

In 2007, by Bratton and Kennedy [11], the Standard Particle Swarm Optimization (SPSO) is defined which is designed by a straightforward extension of the PSO original algorithm while taking into account more recent developments that can be expected to improve performance on standard measures. This standard algorithm is intended for use both as a baseline for performance testing of improvements to the technique and introduce PSO to the wider optimization community.

In SPSO, every particle only uses a local best particle for velocity updating, which is chosen from its left and right neighbors and itself. We call this a ring topology, as shown in Fig. 2.1.

In the last couple of years, many inertia weight calculation methods were proposed [12]. The SPSO introduced an inertia weight method. This method introduced a new parameter χ, known as the constriction factor. χ is derived from the existing constants

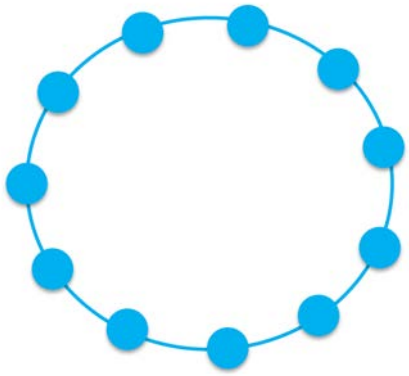

Fig. 2.1: Ring topology

in the velocity update equation:

$$\chi = \frac{2}{| \, 2 - \varphi - \sqrt{\varphi^2 - 4\varphi} \, |}$$

$$\varphi = c_1 + c_2, \varphi > 4$$

It was found that when $\varphi < 4$, the swarm would slowly "spiral" toward and around the best found solution in the search space with no guarantee of convergence, while for $\varphi > 4$ convergence would be quick and guaranteed. Using the constant $\varphi = 4.1$ to ensure convergence, the values $\chi = 0.72984$ and $c_1 = c_2 = 2.05$ are obtained. But there are other possible choices for the constriction coefficients.

This constriction factor is applied to the entire velocity update equation:

$$\begin{aligned} v_{ij}(t+1) = \chi(&v_{ij}(t) \\ &+c_1 * rand_1() * (PbestPos_{ij} - x_{ij}(t)) \\ &+c_2 * rand_2() * (lBestPos_j - x_{ij}(t))) \end{aligned}$$

If $x_{ij}(t+1)$ exceeds the boundary limitation X_{max} or X_{min}, it will be directly set to X_{max} or X_{min}[9].

2.3 Multi Objective Optimization Problems (MOOPs)

An MOOP [13][14][15] is a problem with solutions evaluated along two or more incomparable or conflicting criteria. The solution of an MOOP minimizes or maximizes the values of objective functions simultaneously. In this paper, we assume to minimize all objective functions without loss of generality (see Fig. 2.2). An MOOP is also called a vector optimization, multi-performance, or multi-criteria problem. In general an MOOP can be expressed as follows.

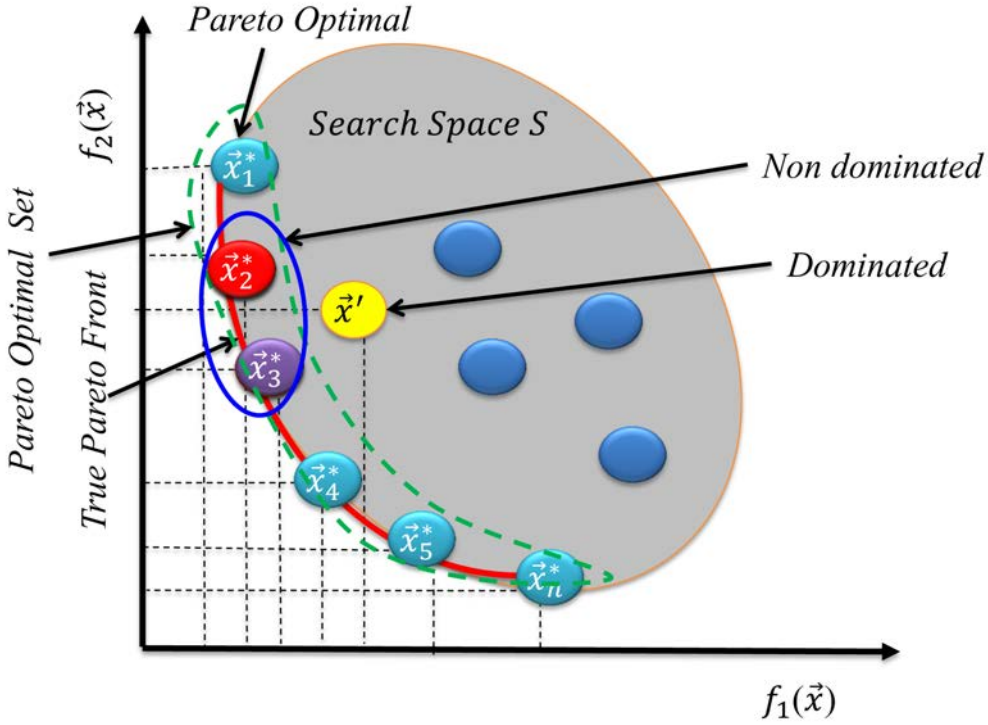

Fig. 2.2: An MOOP.

Finding the vector $\vec{x}^* = [x_1^*, x_2^*, \cdots, x_d^*]^T$ which satisfies the p inequality constraints

$$g_i(\vec{x}) \geq 0, i = 1, 2, 3, \cdots, p$$

and the q equality constraints

$$h_i(\vec{x}) = 0, i = 1, 2, 3, \cdots, q$$

and optimizes the vector function,

$$\vec{f}(\vec{x}) = [f_1(\vec{x}), f_2(\vec{x}).....f_N(\vec{x})]^T$$

where $\vec{x} = [x_1, x_2, x_3, \cdots, x_d]^T$ is the vector of decision variables. In this paper, we focus on an MOOP without any constraint.

In this paper, we use the following basic terminologies.

- Feasible solution (or solution for short): A vector $\vec{x}$ that satisfies all the constraints and variable bounds is known as a feasible solution.
- Search space: A set of all feasible solutions is called a search space which is denoted by S.
- Pareto optimality: A solution $\vec{x}^*$ is pareto optimal iff there does not exist

another solution $\vec{x}$ such that $f_i(\vec{x}) \leq f_i(\vec{x}^*)$ for $i = 1, 2, 3, \cdots, N$ and $f_i(\vec{x}) < f_i(\vec{x}^*)$ for at least one i.

- Pareto optimal set: A Pareto optimal set is a set of the solutions that cannot be improved in one objective function without deteriorating their performance in at least one of the rest.

- Pareto front: A plot of the entire Pareto optimal set in the objective space is called a Pareto front.

- Pareto dominance: A solution $\vec{x}$ dominates another solution $\vec{x}'$ (denoted by $\vec{x} \prec \vec{x}'$) if $\vec{x}$ is better than $\vec{x}'$. We say that a solution is better than other if it is better in at least one objective and in the others are better or equal.

2.4 Multi Objective Particle Swarm Optimization (MOPSO)

PSO is an efficient and simple population-based technique. Therefore, it can be naturally extended to deal with an MOOP. In MOPSO, it is necessary to modify the PSO to solve an MOOP[14]. In general, when solving an MOOP, there are three main goals to achieve [16] : Maximize the number of elements in the Pareto optimal set; Minimize the distance of the Pareto optimal set produced by an algorithm with respect to the Pareto front; Maximize the spread of solutions investigated to obtain a distribution of vectors as smooth and uniform as possible. Two new basic terminologies are introduced in MOPSO.

- Leader: Generally, a nondominanted solution called a leader is used to guide the particles.

- Archive: In each iteration, solutions nondominanted with each other are stored to detect a Pareto optimal set. Those solutions are stored in a data structure called an archive.

In MOPSO, the position and velocity of a particle within the domain of the fitness function are computed by the following two equations [9]:
Velocity update:

$$\begin{aligned} v_{ij}(t+1) = w(&v_{ij}(t) \\ &+c_1 * rand_1() * (pBestPos_{ij} - x_{ij}(t)) \\ &+c_2 * rand_2() * (leaderPos_j - x_{ij}(t))) \end{aligned}$$

Position update:

$$x_{ij}(t+1) = x_{ij}(t) + v_{ij}(t+1)$$

where $leaderPos_j$ is jth component of the position of the leader. The leader is selected in some way from the archive. Several ways to select a leader from the archive were proposed in the literature. Here, SPSO constriction factor χ is also applied to the entire velocity update equation.

2.5 GPU Computing

Over the years, GPUs [17] have been evolved into highly parallelized, multi-threaded, and multi-core processors with tremendous computational horsepower and very high memory bandwidth due to the insatiable demand for real-time high definition 3D graphics. GPUs are specialized in compute intensive, highly parallel computation unerringly. Therefore, GPUs are well-suited to meet the demand of simultaneous processing of large data and repetitive operation with high arithmetic intensity. Due to those advantages, in recent years GPUs have entered into the mainstream application and have successfully been implemented in many applications such as Voronoi diagram and neural network computation, etc.[18],[19].

Over the years, many platforms and programming models have been proposed for GPU computing, of which the most important platforms are CUDA and OpenCL. Both platforms are based on C/C++ language and have very similar platform models, execution models, memory models and programming models. In MOPSO, a large number of data needs to be processed with the shortest time to find a set of nondominated solutions (a Pareto front). A GPU is very well suited to handle this kind of task and can provide superior performance than a conventional CPU.

2.6 An Overview of CUDA Architecture

This section illustrates the CUDA architecture. CUDA is a parallel computing platform and programming model introduced by NVIDIA. CUDA programming model is a multithreaded programming model which utilizes the multi-core parallel processing power of a GPU to solve complex computational problems without the need of mapping them into a graphics API by the programmer. In a CUDA program, the threads are categorized into two hierarchy structure, a grid and thread blocks. A *thread block* is a set of threads and has dimensionality 1, 2, or 3. A *grid block* is a set of blocks with the same size and dimensionality. A *kernel function* call generates threads as a grid with given dimensionality and size. The threads in a thread block can share data efficiently via shared memory. However, the maximum number of threads per block is limited to 1024. So, if more than 1024 threads are required, we have to partition threads into several thread blocks with the same size. Some applications have already been developed based on CUDA, for example, matrix multiplication, real-time visual hull computation, image denoising, and so on [20][21][22]. In this

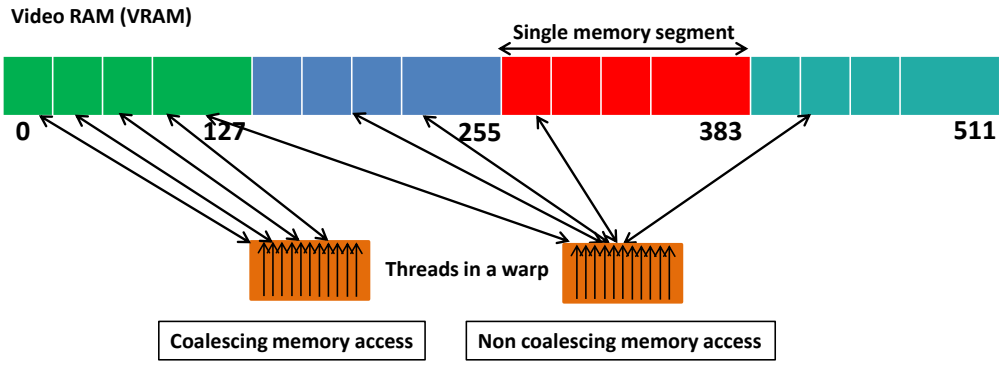

Fig. 2.3: Memory access on a GPU.

paper, we intend to implement SPSO, MOPSO on a GPU in parallel to accelerate the running speed of it.

2.7 Coalescing Memory Access

GPUs provide high-bandwidth memory. It is important to follow the right memory access pattern to get maximum memory bandwidth. The most efficient way to access global memory in a GPU is the coalescing memory access pattern.

During the coalescing memory access, the Video RAM (VRAM) memory is used. VRAM bandwidth is most efficiently used when a single memory segment of 32, 64, 128 or 256 bytes is simultaneously accessed by threads in a warp (during the execution of a single read or write instruction). This is called coalesced memory access [23] (See Fig. 2.3). On the other hand, VRAM bandwidth is inefficiently used and also time consuming when the simultaneous memory access by threads in a warp is non-coalesced. In this case, at least two memory segments are accessed by threads in a warp. If the threads access the different segments of the VRAM, the bandwidth efficiency is dramatically decreased.

2.8 Random Number Generators

Random number generators are used to deterministically generate a sequence of numbers that is difficult to distinguish from a natural random sequence. Random numbers are essential elements in a great number of solutions in computer science. Randomized algorithms require a random source to ensure computational complexity bounds and sampling methods often require randomness to accurately represent the terms they are surveying.

```
1 // S1, S2, S3, and M are all constants,
2 //      and z is part of the private per-thread generator state.
3 unsigned TausStep(unsigned &z, int S1, int S2, int S3,
4             unsigned M){
5         unsigned b=(((z << S1) ^ z) >> S2);
6         return z = (((z & M) << S3) ^ b);
7 }
```

Fig. 2.4: A single step of the combined Tausworthe generator [24].

According to the source of randomness, random number generators (RNGs) can be classified into three groups [24]: true random number generators (TRNGs), quasi-irandom number generators (QRNGs) and pseudo number generators (PRNGs). The most common type of random number generator is the PRNG. PRNGs are designed to look as random as a TRNG, but PRNGs can be implemented in deterministic software because the state transition function can be predicted completely. In our implementation of MOPSO on a GPU, we consider only the following type of generator - a generator that is a single step `TausStep` of the combined Tausworthe generator[24].

2.8.1 Combined Tausworthe Generator

The combined Linear Feedback Shift Register (LFSR) or tausworthe generator provides a fast implementation and makes use of exclusive-or operations to combine the results of two or more independent binary matrix derived streams, providing a stream of longer period and much better quality. Each independent stream is generated using `TausStep` shown in Fig. 2.4, in six bitwise instructions.

A popular version of this procedure is the four component LFSR113 generator[25] which produces a random stream with a period of approximately 2^{113}. However, statistical tests show that even the four-component LFSR113 produces significant correlations across 5-tuples and 6-tuples for relatively small sample sizes. Also the periods are not sufficiently long enough. However in terms of speed, these are highly competitive to those which are currently available from software libraries. In our implementation, we used as a random number generator a single step `TausStep` with `S1 = 6, S2 = 13, S3 = 18,` and `M = 4294967294UL`, which is the first component of the four component LFSR113.

2.8.2 Linear Congruential Generator

One of classic generators is the linear congruential generator (LCG) which was introduced by Knuth in 1969 [26]. It uses a transition function of the form shown below

$x_{n+1} = (ax_n + c) mod\ m$

The maximum period of the generator is m(assuming the triple (a, c, m) has certain properties), but this means that in a 32-bit integer, the period can be at most 2^{32}, which is far too short. LCGs also have known statistical flaws, making them unsuitable for modern simulations.

2.8.3 Xorshift RNG

Xorshift RNG was proposed as a class of very fast, good-quality PRNG by Marsaglia[27]. It produces a sequence of $2^{32} - 1$ integers, or a sequence of $2^{64} - 1$ pairs of integers or a sequence of $2^{96} - 1$ triples of integers etc., by means of repeated use of a simple construction: exclusive-or (xor) a computer word with a shifted version of itself. In C, the basic operation is y xor operation on $(y << a)$ for shifts left, y xor operation on $\hat{}(y >> a)$ for shifts right. Combining such xorshift operations for various shifts and arguments provide extremely fast and simple RNGs that seem to do very well on tests of randomness. Several weaknesses of such generators was showed by Panneton and L'Ecuye[28]. Subsequent analysis by Vigna[29], explores the space of possible generators obtained by multiplying the result of a xorshift generator by a suitable constant.

2.8.4 Multiply-with-carry(MWC)

Multiply-with-carry (MWC)[27] was invented by Marsaglia to generate a sequences of random integers based on an initial set from two to many thousands of randomly chosen seed values. The MWC methods provides a very fast generation of random number sequences with immense periods ranging from 2^{60} to $2^{2000000}$ using a very simple arithmetic.

2.8.5 cuRAND Library

The cuRAND library[30] can be used for simple and efficient generation of high -quality pseudorandom and quasirandom number sequences. CUDA libraries can be directly called from kernel code. A quasirandom sequence of n dimensional points is generated by a deterministic algorithm designed to fill an n dimensional space evenly. Configuring the device cuRAND API requires four options: an RNG algorithm with which to generate a random sequence, a distribution to which the returned values will adhere, a seed from which to start and an offset into the random number sequence at which to begin sampling. The device API requires the explicit specification of each of these parameters. The device API includes functions for pseudorandom generation and quasirandom generation. The functions for pseudorandom sequences support bit generation and generation from distributions. Bit generations are given below:

Bit Generation with XORWOW generator (curand XORWOW): Following a call to *curand_init*(), *curand*() returns a sequence of pseudorandom numbers with a period greater than 2^{190}.

```
__device__ unsigned int
       curand (curandStateXORWOW_t *state)
__device__ void curand_init
     ( unsigned long long seed,
        unsigned long long sequence,
        unsigned long long offset,
        curandStateXORWOW_t *state)
```

Different seeds are guaranteed to produce different starting states and different sequences. The same seed always produces the same state and the same sequence.

Bit Generation with Philox generator (curand Philox4_32_10_t): Following a call to *curand_init*(), *curand*() returns a sequence of pseudorandom numbers with a period 2^{128}.

```
__device__ unsigned int
curand (curandStatePhilox4_32_10_t *state)
__device__ void curand_init
     ( unsigned long long seed,
        unsigned long long sequence,
        unsigned long long offset,
        curandStatePhilox4_32_10_t *state)
```

Subsequence and offset together define offset in a sequence with period 2^{128}. Offset defines offset in subsequence of length 2^{64}. When last element from subsequence is generated, then the next random number is first element from consecutive subsequence. The same seed always produces the same state and the same sequence. Sequences generated with different seeds usually do not have statistically correlated values, but some choices of seeds may give statistically correlated sequences.

2.9 Thrust Library

Thrust [31] is a C++ template library for both GPUs and CPUs which is based on the Standard Template Library (STL). Mainly, Thrust allows to implement high performance parallel applications with minimal programming effort through a high-level interface that is fully interoperable with CUDA C/C++. These high-level abstractions provide Thrust with the freedom to select the most efficient implementation automatically. As a result, Thrust can be utilized in rapid prototyping of CUDA applications, where programmer's productivity matters most, as well as in production, where robustness and absolute performance are crucial. Thrust provides a rich collection of data parallel primitives such as scan, sort, and reduce, which can be

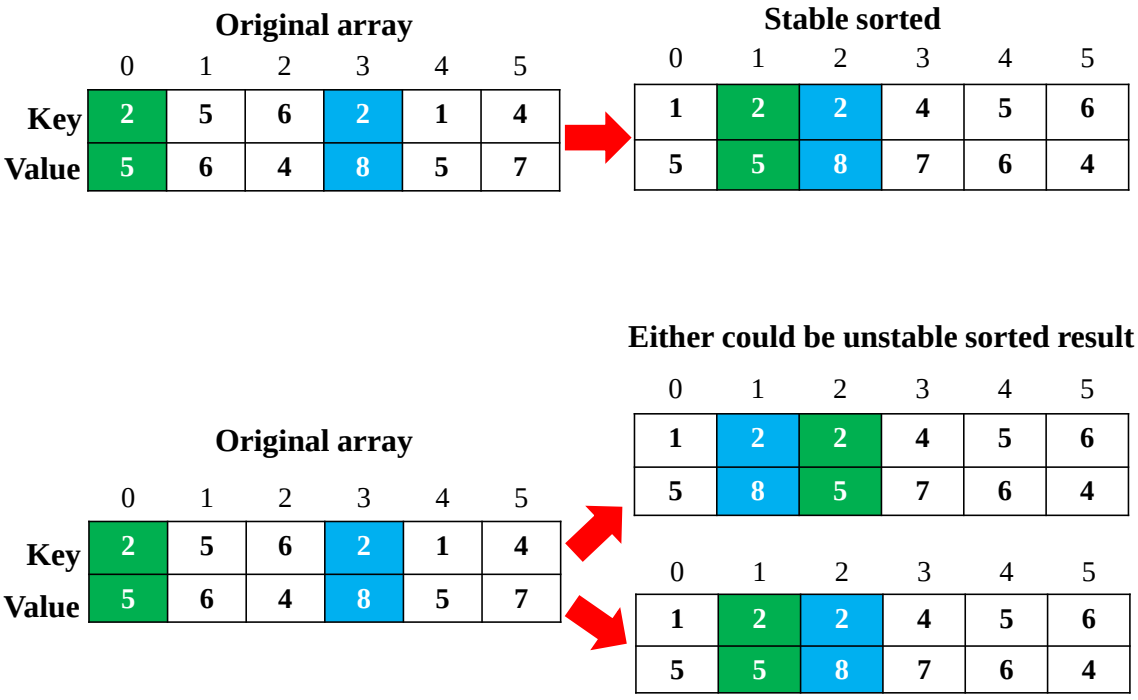

Fig. 2.5: Stable and unstable sort.

composed together to implement complex algorithms with concise and readable source code. Prefix-sums or scan operations are important building blocks in many parallel algorithms such as stream compaction and radix sort. `thrust::exclusive_scan` function provides an efficient parallel implementation of prefix-sums. Thrust offers several functions to sort data or rearrange data according to a given criterion. The `thrust::sort` and `thrust::stable_sort` functions are direct analogs of sort and stable sort in the STL. In our implementation, we used `thrust::stable_sort` function due to stability of sorting.

A sorting algorithm[32] is claimed to be stable if two objects with equal keys appear in sorted output in the same order as the order they appear in the unsorted input array. Whereas a sorting algorithm is considered to be unstable if there are two or more objects with equal keys that don't appear in the same order before and after sorting. For instance, in Fig. 2.5, key 2 appears twice at position "0" and "3" and their order is preserved in unsorted and sorted array i.e, key 2 at position "0" appear first in unsorted and sorted array (before and after sorting). In case of unstable sort this order of appearance before and after sorting is not preserved.

2.10 CUB Library

The CUB library[33] is developed as an open-source project by NVIDIA Research. It is a C++ template library, which provides state-of-the-art and reusable software components for every layer of the CUDA programming model. In particular, it includes warp-wide, block-wide and device-wide collective primitives. Warp-wide "collective" primitives such as cooperative warp-wide prefix scan, reduction, etc. are safely specialized for each underlying CUDA architecture. Block-wide "collec tive" primitives such as cooperative I/O, sort, scan, reduction, histogram, etc. are compatible with arbitrary thread block sizes and types. Device-wide primitives such as parallel sort, prefix scan, reduction, histogram, etc. are compatible with CUDA dynamic parallelism.

As a SIMT library[34] and software abstraction layer, CUB provides simplicity of composition, high performance, performance portability, simplicity of performa nce tuning, robustness, durability, reduced maintenance burden, and a path for language evolution.

CUB's device-wide primitives can be performance-tuned to match the processor resources provided by each CUDA processor architecture. As a result, CUB implementations demonstrate much better performance-portability when compared to more traditional, rigidly-coded parallel libraries such as Thrust[31] .

In our implementation, we used CUB library for computing a prefix-sums. In particular, we used `cub::DeviceScan` which provides device-wide parallel operations for computing a prefix-sums across a sequence of data items residing within device-accessible memory.

CUB and Thrust share some similarities, for example they both provide similar device-wide primitives for CUDA. CUB and Thrust are complementary to each other and they can be used together. In fact, the CUB project arose out of a maintenance need to achieve better performance-portability within Thrust by using reusable block-wide primitives to reduce maintenance and tuning effort.

2.11 Parallel Models

For the parallel implementation of multi-objective evolutionary algorithms (MO EAs), several models were provided by Van Veldhuizen, Zydallis and Lamont [35].

$Master - slave\ model$: The master-slave model is one of the simplest ways to parallelize an MOEA. Here, a master processor executes the MOEA and the objective function evaluations are distributed among a number of slave processors. As soon as the slaves complete the evaluations they return the objective function values to the master and remain idle until the next generation.

$Diffusion\ model$: An individual is evaluated and mutated on a single proc essor. Selection and crossover are limited to a few neighbors. Such an approach is very useful

when we execute MOPSO on massively parallel processors.

Island model : This model is very popular among researchers, but it requires many parameters and design decisions. The population is divided into several small sub-populations, called islands, which evolve independently of each other.

Hierarchical model : This model combines a coarse-grained parallel scheme at a high level island model with a fine-grained scheme at a low level diffusion model. It was a slower model.

From the knowledge about the above models, we come to the conclusion that the master-slave model is suitable for our parallel implementation of MOPSO. The master-slave model is most suited for stream processors such as NVIDIA's GPU.

Chapter 3

A CUDA Implementation of the Standard Particle Swarm Optimization

3.1 Introduction

This chapter presents a new parallelized implementation of the Standard Particle Swarm Optimization (SPSO), an extension of PSO, partially using coalescing memory access. The experimental results show that the proposed parallelized SPSO increases the processing speed compared to previously proposed approach on a GPU based on the CUDA architecture. We investigate a large number of iterations to reach a good optimization solution. We also investigate the impact of fine grained parallelism in high-dimensionality problems. A large swarm of particles is analyzed to achieve a desired SPSO solution with improved computational time compared to a CPU.

The remainder of this chapter is organized as follows. In Section 3.2, summarizes some related works. In Section 3.3 we provide the SPSO and its implementation on a GPU. Subsequently, in Section 3.4, we present and analyze the obtained results and compare it to the previous implementation in terms of execution time, speedup and fitness values. Finally, in Section 3.5, we give some concluding remarks and point out directions for next work.

3.2 Related Works

Y. Zhou and Y. Tan [36] presented parallel approach to run SPSO on a GPU. Some experiments are conducted by running SPSO both on a GPU and a CPU, respectively. The running time of the SPSO based on a GPU is greatly shortened compared to that of the SPSO on a CPU. Running speed of GPU SPSO can be more than 11 times as fast as that of CPU SPSO.

Calazan et al. [37] proposed GPU based Parallel Dimension Particle Swarm Optimization (PDPSO). For optimization problems with low computational complexity i.e. low dimensions, CPU based PDPSO gives better performance than GPU based PDPSO. A GPU provides positive impact on large optimization problems. Fine grained model is used i.e. distribute one dimension to one thread. GPU PDPSO is 85 times faster than CPU PDPSO.

V. K. Reddy and S. Reddy [38] implemented a parallel asynchronous version and synchronous version of PSO on a GPU and compare the performance in terms of execution time and speedup with their sequential version on a CPU. Mussia et al. [39] proposed two different ways of exploiting GPU parallelism. The execution speeds of the two parallel algorithms are compared, on functions which are typically used as benchmarks for PSO, with a sequential implementation of SPSO.

Li and Zhang [40] proposed a CUDA based Multichannel particle swarm algo rithm. The optimization experiments results of 4 benchmark functions like sphere, rastrigin, griewank and rosenbrock, it showed that the CUDA-based parallel algor ithm can greatly save computing time and improve computing accuracy. Compar ison of results on GeForce GTX 480 GPU with Intel Core i7 860 also showed, as population gradually increases, speedup also increases.

Calazan et al. implementation [41] of a Cooperative Parallel Particle Swarm Optimization (CPPSO) for high-dimension problems on GPUs results showed that the proposed architecture is up to 135 times and not less than 20 times faster in terms of optimization time when compared to the direct software execution of the algorithm.

Zhou and Tan [9] compared with the CPU based sequential Multiobjective Particle Swarm Optimization (MOPSO) and GPU based parallel MOPSO. Imple mentation of GPU based parallel MOPSO is much more efficient in terms of running time, and the speedups range from 3.74 to 7.92 times.

Zhu et al. [42] presented a faster parallel Euclidean Particle Swarm Optim ization (pEPSO). Five benchmark functions had been employed to examine the perfor mance of the pEPSO. Experimental results showed that the average proc essing time of calculating fitness had been accelerated to maximum 16.27 times the original algorithm (EPSO).

Silva and Filho [43] proposed to use multiple sub-swarms. Each sub-swarm is executed in a GPU block aiming at maximizing data alignments and avoiding instructions bifurcations and also provided two communication mechanisms and two topologies in order to allow the sub-swarm to exchange information and collaborate by using the GPU global memory. They showed speedups up to 100 and 5 times when compared to the serial implementation and PSO start-of-art implementation for CUDA.

Bali et al. [44] illustrated a novel parallel approach to run SPSO on GPUs and applied to TSP (GPU-PSO-A-TSP). Results showed that running speed of GPU-PSO is four times as fast as that of CPU PSO.

Algorithm 1: CPU SPSO algorithm

```
for i = 1 to n do
    initialize the position & velocity of particle i randomly
    initialize the pBest & lBest of particle i to infinity
for j = 1 to iterations do
    for i = 1 to n do
        compute fitness[i]
        if fitness[i] < pBest[i] then
            pBest[i] := fitness[i]
    for i = 1 to n do
        l:= 1 + (n + l - 2) % n
        r:= 1+r % n
        lBestIndex[i]:=i
        if pBest[r] < pBest[lBestIndex[i]] then
            lBestIndex[i]:=r
        if pBest[l] < pBest[lBestIndex[i]] then
            lBestIndex[i]:=l
    for i = 1 to d do
        update velocity & position of particle i
gBest := infinity
for i = 1 to n do
    if pBest[i] < gBest then
        gBest := pBest[i]
```

3.3 Implementing SPSO Using CUDA

The CPU SPSO algorithm is described in Algorithm 1. Here the personal best (pBest) position and value are obtained by adjusting the pervious personal position and fitness value. After that, the local best position and value are updated by comparing the right (pBest[r]) and left (pBest[l]) neighbor respectively. Finally, it updates the velocity and position of the particle (shown in algorithm 1). In a parallel processing system the individual particle can find their best position simultaneously and increases the SPSO efficiency significantly. During our development of SPSO on CUDA, fitness function, pBest, lBest, gBest, updated position and velocity of particles in the swarm can be computed on a GPU while initialization also can be done on a GPU. In this section we show the main parts of the SPSO-code developed for a GPU. The SPSO algorithm, expressed in a CUDA-based pseudocode, is given in Algorithm 2.

We used seven kernel functions. The first kernel allocates memory on a GPU with 1 block of 1 thread. In order to generate random numbers on a GPU, we used a single step `TausStep` of the combined Tausworthe generator for each thread with an independent seed number.

The second kernel generates random number seeds using a single step `TausStep`

Algorithm 2: GPU SPSO algorithm

let n = number of particles; let d = number of dimensions
allocate memory on a GPU with 1 block of 1 thread.
generate random number seeds for TausStep & initialize velocity
& position of each particles of each particle on a GPU with n blocks of d threads.
for $i = 1$ to iterations do
    calculate fitness values & pBests on a GPU with $(n + 32 - 1)/32$ blocks of 32 threads.
    calculate lBest on a GPU with $(n + 32 - 1)/32$ blocks of 32 threads.
    update velocities & positions on a GPU with n blocks of d threads.
end;
calculate gBest by atomicMin() on a GPU with $(n + 32 - 1)/32$ blocks of 32 threads.
transfer gBest to a CPU
free seeds using 1 block of 1 thread
free memory on a GPU
return the best result & corresponding position

and initializes the basic information of each particle such as position and velocity on a GPU with n blocks of d threads. For this initialization, we used coalescing memory access. The arrays on VRAM are arranged to realize coalescing memory access, in particular the array of random number seeds.

The third kernel generates $(n + 32 - 1)/32$ blocks of 32 threads to compute the fitness function. This kernel performs the reduction process to get the fitness value. When this process is completed, the current pBest value of each particle is compared with the previous pBest value. If the current pBest value is smaller than the previous pBest value, then it is updated. When the pBest is updated, the threads of this kernel also update the coordinates of the pBest value accordingly. The CUDA pseudo-code for kernel fitness and pBest calculator is given in Fig. 3.1.

The next kernel computes the local best (lBest) value by comparing the previous pBest of neighboring (left and right) particles. The details of kernel lBest calculator are shown in Fig. 3.2. Here the neighbors of a particle are particles (tid+1) and (tid-1). During this process, in each terminal of the array, it will cause illegal access (out of array) if no special consideration is given.

Therefore, we set exception handling for particles number “0” and “n-1”. For number “0” the left neighbor was set as “n-1” and for number “n-1”, the right neighbor was set as “0”. This implementation enables us to implement ring topology. The local best position of the particle is calculated based on the definition of right neighborhood and left neighborhood.

The fifth kernel generates n blocks of d threads, which compute velocities and positions for the next iteration. In this function, we also used coalescing memory access. Our experiment showed that coalesced access is faster than non-coalesced access. Therefore in our experiment, coalescing memory access has an important effect that our implementation is faster than other related work [36].

We also used an atomic function [34] which performs a read-modify-write atomic

```
1  __global__ void evaluate_particles(int d, int n, float *x,
2  float *pValue, float *pBestValue, float *pBestPos, int *lBestIdx)
3  {
4      int i = blockDim.x * blockIdx.x + threadIdx.x;
5      if (i >= n) return;
6      pValue[i] = fitness_function(d, &x[d * i]);
7
8      if (pValue[i] < pBestValue[i]) {
9          pBestValue[i] = pValue[i];
10         pBestPos[d * i ..d * i + d - 1] = x[d * i ..d * i + d-1];
11     }
12 }
```

Fig. 3.1: Our kernel function for computing fitness values and personal bests.

operation on one 32-bit or 64-bit word residing in global or shared memory. Atomic operations are an advanced feature provided by the GPU vendor NVIDIA. The operation is atomic in that no other thread can access this address until the operation is complete. The atomic operations guarantee the correct calculation result.

Among atomic functions, atomicMin function computes the minimum of given values. atomicMin(addr, val) reads the 32-bit or 64-bit word old located at the address addr in global or shared memory, computes the minimum of old and val, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old. atomicMin functions are provided only for type int, unsigned, and unsigned long long. However, in our proposed method, the expected pBest values are of float type. Therefore, we need the atomicMin function for float type. Fortunately, we can simply implement it using the atomicMin function for unsigned type as shown in Fig. 3.3.

Our implementation merely casts float type into unsigned type. Why this simple implementation works correctly can be explained as follows. In CUDA C, both unsigned type and float type are of size 32 bits (shown in Fig. 3.4). Notice that sign bit, exponent, and mantissa in a float value are allocated to more significant bit(s) in this order. Hence, for any given float value x and y, the magnitude correlation of x and y is equivalent to that of x and y as unsigned integers if x and y are non-negative integers. If only non-positive values are given, our atomicMin() implementation behaves as atomicMax().

The most important kernel computes the global best value based on the all particle positions in the swarm. During this kernel process, we used atomicMin() function for good fitness values. Finally, our last kernel is used to free the array for seeds. In our experiments, better performance can be achieved if all the kernel functions use coalesceing memory access.

```
1  __global__ void calculate_localBest(int n, float *pBestValue,
2            int *lBestIdx)
3  {
4       int tid = blockDim.x * blockIdx.x + threadIdx.x;
5       if(tid >= n) return;
6
7       int right = (tid == (n - 1)) ?      0   : tid+1;
8       int left  = (tid ==      0 ) ? (n - 1) : tid-1;
9       int lBestCandidate = tid;
10
11      if(pBestValue[right] < pBestValue[lBestCandidate])
12         lBestCandidate = right;
13      if(pBestValue[left]  < pBestValue[lBestCandidate])
14         lBestCandidate = left;
15      lBestIdx[tid] = lBestCandidate;
16 }
```

Fig. 3.2: Our kernel function for local best.

```
1  __device__ inline float atomicMin(float *addr, float val)
2  {
3       unsigned old=atomicMin((unsigned*)addr,*((unsigned *)&val));
4       return *((unsigned *) &old);
5  }
```

Fig. 3.3: A simple and efficient implementation of atomicMin function for non-negative float values.

3.4 Experiments and Analysis

In this section, we present our experimental results which have been obtained using a CPU and a GPU platforms described below. The results are compared with other previous implementation [36] in terms of speedup, execution time, loop time per iteration and fitness values. In our experiment, seven classical benchmark functions [45] (shown in Table 3.1) were used to evaluate the performance of the implementations. The six functions except Easom always return non-negative values. In contrast, Easom always returns a non-positive value. Since our atomicMin() implementation behaves as atomicMax() for non-positive values as explained in Section 3.3, the aims of our experiments are to minimize the six functions and to maximize Easom function although all the seven functions are originally for

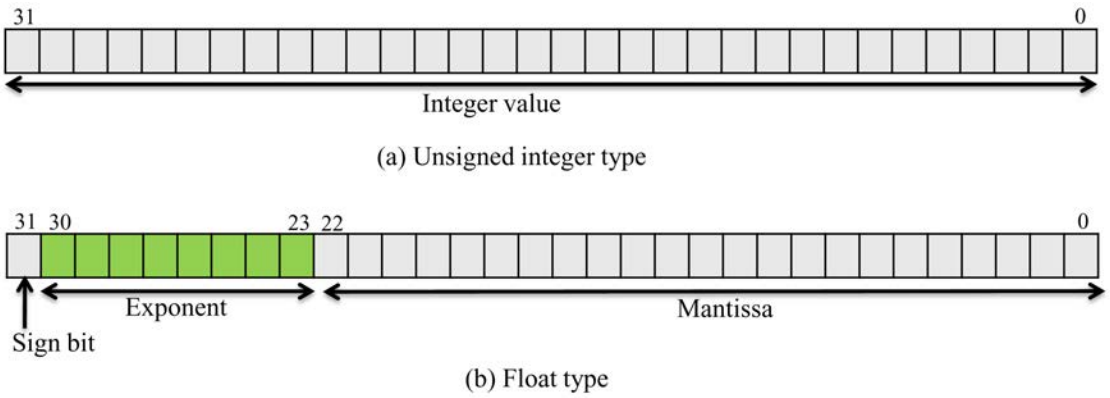

Fig. 3.4: Difference between integer type and float type.

minimization. Hence, the optimal solutions of all the seven functions are 0.

One of the most common measure used by the parallel computing community to compare the test results is speedup. Speedup is defined as the ratio of the execution time $Time_{CPU}$ of the sequential implementation to the execution time $Time_{GPU}$ of the parallel implementation:

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

We run the CPU SPSO and the GPU SPSO using the same configuration of parameters n and d, which are the number of particles and the dimensions respectively.

Our tests were conducted using an NVIDIA GeForce GTX 980 GPU [46] and an Intel(R) Core i5(TM) 4570 @ 3.20GHz with 8 GB RAM. The operating system was Windows 7 Professional SP1. For compilation, we used Microsoft Visual Studio 2012 Professional Edition and CUDA 7.5 SDK.

In all experiments the number of dimensions and particles were respectively set from 50 to 200 and from 2000 to 20000. Each experiment was run until the maximum number of iterations has been reached, which was set at 2000. We carried out another set of simulations to evaluate convergence speed with respect to the number of iterations. In all cases, we got good solutions for seven functions (See Figs. 3.5 and 3.6).

The average results using some complex functions are shown in Tables 3.2 through 3.8. In these cases, the number of particles ranges from 2000 to 5000, the number of dimensions is 50 and the number of iterations is 2000. The GPU SPSO and the CPU SPSO were run on from f_1 to f_7 functions for 50 times independently with different seeds.

Analyzing the data of the tables, we can observe that in Table 3.8, a GPU SPSO can reach a maximum speedup of greater than 170 times when the swarm population

Table 3.1: Benchmark Test Functions

Name	Equation	Bounds
Sphere	$f_1 = \sum_{i=0}^{d} \ x_i^2$	$(-5.12, 5.12)^d$
Rosenbrock	$f_2 = \sum_{i=0}^{d-1} \ (100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$	$(-10, 10)^d$
Rastrigin	$f_3 = \sum_{i=0}^{d}[x_i^2 - 10 * \cos(2\pi x_i) + 10]$	$(-5.12, 5.12)^d$
Griewank	$f_4 = \frac{1}{4000} \ \sum_{i=1}^{d} x_i^2 - \prod_{i=1}^{d} \cos(\frac{x_i}{\sqrt{i}}) + 1$	$(-600, 600)^d$
Ackley	$f_5 = -20 \exp[-\frac{1}{5}\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}] - \exp[\frac{1}{d}\sum_{i=1}^{d} \cos(2\pi x_i)] + 20 + e$	$(-32.768, 32.768)^d$
De Jong	$f_6 = \sum_{i=1}^{d} \mid x_i \mid^{(i+1)}$	$(-1, 1)^d$
Easom	$f_7 = -(-1)^d(\prod_{i=1}^{d} \cos^2(x_i)) \exp[-\sum_{i=1}^{d}(x_i - \pi)^2]$	$(-2\pi, 2\pi)^d$

size is 5000 and dimension size is 50, running on complex function f_7. For more complex functions speedup may be even greater (see Figs. 3.7 and 3.8).

However, when optimized by a GPU SPSO, the time needed is almost the same among the seven functions under the same dimension and population. Therefore, the curves for f_1 to f_7 by a GPU SPSO in Figs. 3.9 to 3.12 are overlapped with each other.

In the next test, the swarm population was set at 2000 and the dimension was changed from 50 to 200. The results shown in Tables 3.9 to 3.15 demonstrate that running PSO on a CPU to optimize high dimensional problems is slow, but the speed can be greatly accelerated if we run it on a GPU (see Figs. 3.9 to 3.12). In these cases, in Table 3.15, a GPU SPSO can reach a maximum speedup of greater than 139 times when the dimension size is 200 and the swarm population size is 2000, running on complex function f_7. For more complex functions speedup may be even greater (See Figs. 3.13 and 3.14).

We observed that speedup is increased when swarm population and dimension size are large due to the coalescing memory access. Due to difference between random number sequence on a CPU and that on a GPU, the gBest values may be slightly different. Moreover, execution time depends on function types and how

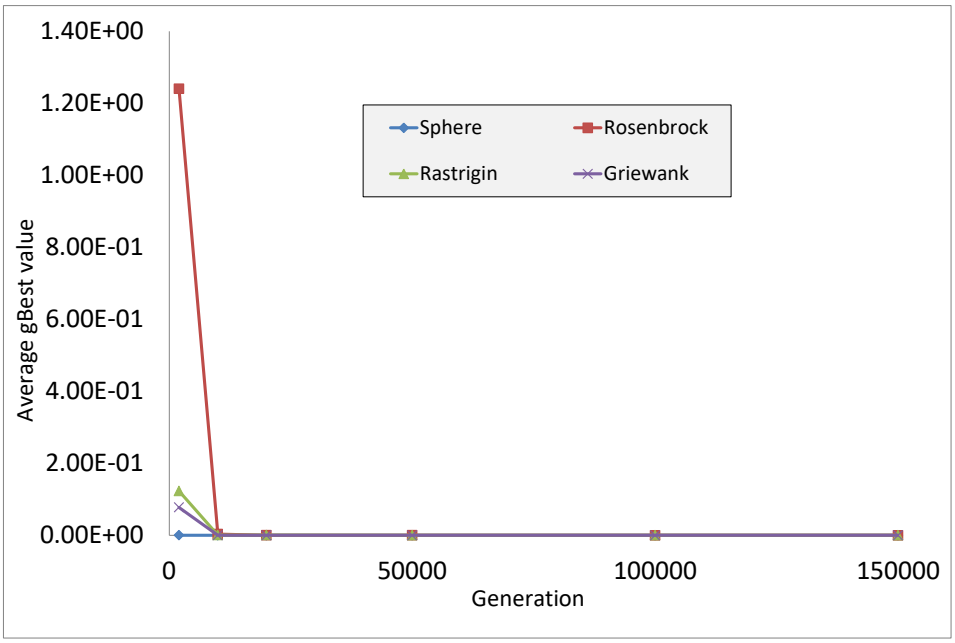

Fig. 3.5: gBest and generation for f_1 *to* f_4 functions

many operators that have been used inside a function. We focused on execution time and speedup for fixed number of iterations.

However, we also investigate on execution time and speedup for the stop condition when an acceptable optimization value has been found. The results shown in Table 3.16 are obtained when running on simple function f_1. The speedup can be improved significantly using complex function. The other implementation [36] of SPSO is slower than our implementation under the same dimension and the same population size (see Table 3.17). In our implementation the best key point is that we try to implement good parallelization. In the other implementation of SPSO, the random numbers were generated inside a CPU whereas, in our implementation, all the random numbers were generated inside a GPU. The implementation of a single step `TausStep` of the combined Tausworthe generator on a CPU is same as a GPU implementation and not time-consuming. However, the transfer of generated random numbers from a CPU to a GPU is time-consuming. This increases the overall processing time of the other implementation. In our work, the random numbers were generated by TausStep and initialized the basic information of each particle on a GPU for SPSO application to reduce the processing time significantly. Moreover, we used some kernels that are using coalescing memory access which increase the processing speed.

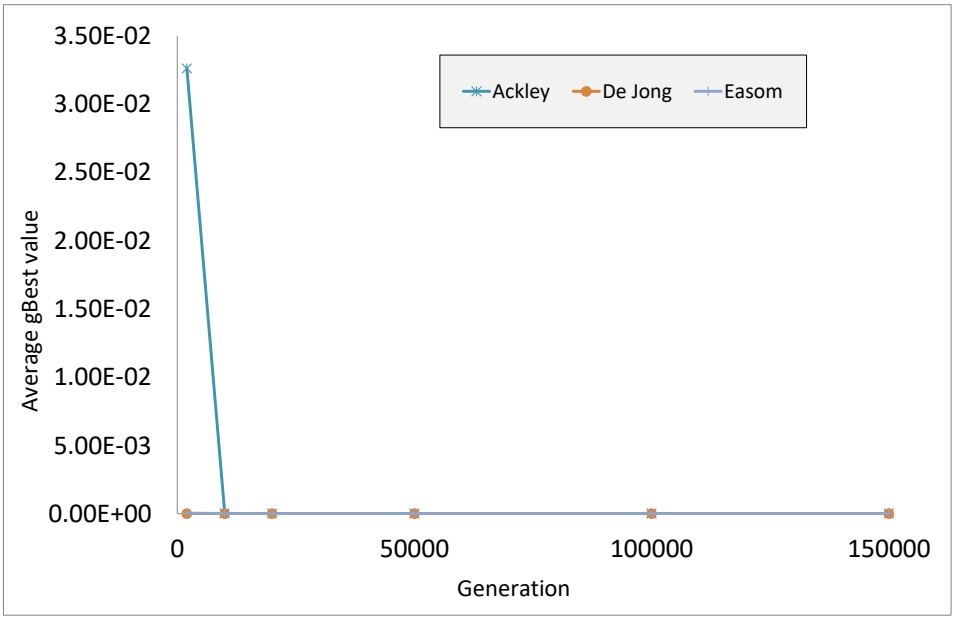

Fig. 3.6: gBest and generation for f_5 *to* f_7 functions

The "atomicMin" function was used to calculate good solutions in SPSO on a GPU. On the other hand, in the previous approach, solutions were calculated by using a complex algorithm.

3.5 Summary

This chapter has presented an implementation of the SPSO on the CUDA architecture. The proposed GPU SPSO significantly reduces execution time compared to previous development. We have achieved a good fitness value with short execution time and kernel loop time simultaneously. Moreover, the implementation has a significant speedup when compared to the CPU serial implementation. The proposed imple mentation is 170 times faster.

In this paper we focused on realizing good implementation of the original SPSO. In our future work, we want to investigate the effect of the Pseudorandom Number Generators on the SPSO on a GPU and parallel implementation of the extensions [37][41][9] of SPSO to improve the efficiency of other implementations. We also want to improve the SPSO performance by implementing all kernel functions using

Table 3.2: GPU SPSO and CPU SPSO on f_1 (number of dimensions $d = 50$)

n	CPU Time(s)	GPU Time(s)	CPU loop Time(s)	GPU loop Time(s)	CPU gBest Value	GPU gBest Value	Speedup
2000	1.3888	0.1498	0.00069	0.00003	1.32E-04	1.33E-04	9.2
3000	2.0852	0.1962	0.00104	0.00005	1.14E-04	1.07E-04	10.6
4000	2.7791	0.2214	0.00138	0.00006	1.10E-04	9.47E-05	12.5
5000	3.4822	0.2561	0.00174	0.00007	9.84E-05	8.18E-05	13.5

Table 3.3: GPU SPSO and CPU SPSO on f_2 (number of dimensions $d = 50$)

n	CPU Time(s)	GPU Time(s)	CPU loop Time(s)	GPU loop Time(s)	CPU gBest Value	GPU gBest Value	Speedup
2000	1.6302	0.1607	0.00081	0.00004	1.26E+00	1.24E+00	10.1
3000	2.4432	0.1955	0.00122	0.00005	1.15E+00	1.12E+00	12.4
4000	3.2494	0.2224	0.00162	0.00006	1.11E+00	1.08E+00	14.6
5000	4.0665	0.2588	0.00203	0.00007	1.03E+00	9.62E-01	15.7

coalescing memory access which should improve SPSO performance significantly.

Table 3.4: GPU SPSO and CPU SPSO on f_3 (number of dimensions $d = 50$)

n	CPU Time(s)	GPU Time(s)	CPU loop Time(s)	GPU loop Time(s)	CPU gBest Value	GPU gBest Value	Speedup
2000	5.3248	0.1779	0.00266	0.00004	1.82E-01	1.23E-01	29.9
3000	8.0448	0.2201	0.00402	0.00006	8.05E-02	8.20E-02	36.5
4000	10.7043	0.2437	0.00535	0.00007	7.36E-02	7.24E-02	43.9
5000	13.4521	0.2691	0.00672	0.00008	6.07E-02	6.25E-02	49.9

Table 3.5: GPU SPSO and CPU SPSO on f_4 (number of dimensions $d = 50$)

n	CPU Time(s)	GPU Time(s)	CPU loop Time(s)	GPU loop Time(s)	CPU gBest Value	GPU gBest Value	Speedup
2000	5.9915	0.1984	0.00299	0.00005	7.66E-02	7.75E-02	30.1
3000	8.9916	0.2385	0.0077	0.00007	5.94E-02	6.24E-02	37.7
4000	11.9985	0.2625	0.0102	0.00008	5.30E-02	5.74E-02	45.7
5000	15.0037	0.2879	0.0128	0.00009	4.72E-02	5.30E-02	52.1

Table 3.6: GPU SPSO and CPU SPSO on f_5 (number of dimensions $d = 50$)

n	CPU Time(s)	GPU Time(s)	CPU loop Time(s)	GPU loop Time(s)	CPU gBest Value	GPU gBest Value	Speedup
2000	4.4402	0.1774	0.00222	0.00004	3.38E-02	3.26E-02	25.0
3000	6.6004	0.2179	0.00329	0.00006	3.03E-02	2.92E-02	30.2
4000	8.8019	0.2416	0.00400	0.00007	2.88E-02	2.75E-02	36.4
5000	10.9857	0.2661	0.00549	0.00008	2.72E-02	2.52E-02	41.2

Table 3.7: GPU SPSO and CPU SPSO on f_6 (number of dimensions $d = 50$)

n	CPU Time(s)	GPU Time(s)	CPU loop Time(s)	GPU loop Time(s)	CPU gBest Value	GPU gBest Value	Speedup
2000	4.1840	0.1829	0.00209	0.00005	8.62E-24	8.42E-24	22.8
3000	6.2895	0.2185	0.00314	0.00006	7.64E-25	5.07E-25	28.7
4000	8.3929	0.2422	0.00420	0.00007	5.36E-25	8.03E-26	34.6
5000	10.5018	0.2689	0.00525	0.00008	7.43E-26	5.49E-26	39.0

Table 3.8: GPU SPSO and CPU SPSO on f_7 (number of dimensions $d = 50$)

n	CPU Time(s)	GPU Time(s)	CPU loop Time(s)	GPU loop Time(s)	CPU gBest Value	GPU gBest Value	Speedup
2000	19.2378	0.1904	0.00961	0.00005	0.00E+00	0.00E+00	101.0
3000	28.6848	0.2324	0.01434	0.00007	0.00E+00	0.00E+00	123.4
4000	38.2021	0.2547	0.01910	0.00008	0.00E+00	0.00E+00	149.9
5000	47.8445	0.2800	0.02392	0.00009	0.00E+00	0.00E+00	170.8

Table 3.9: GPU SPSO and CPU SPSO on f_1 (number of particles $n = 2000$)

d	CPU Time(s)	GPU Time(s)	CPU loop Time(s)	GPU loop Time(s)	CPU gBest Value	GPU gBest Value	Speedup
50	1.3888	0.1498	0.00069	0.00003	1.32E-04	1.33E-04	9.2
100	2.8172	0.2404	0.00140	0.00007	1.00E-02	9.89E-03	11.7
150	4.3661	0.3187	0.00218	0.00010	5.49E-02	5.58E-02	13.6
200	6.0485	0.3903	0.00302	0.00013	1.59E-01	1.60E-01	15.4

Table 3.10: GPU SPSO and CPU SPSO on f_2 (number of particles $n = 2000$)

d	CPU Time(s)	GPU Time(s)	CPU loop Time(s)	GPU loop Time(s)	CPU gBest Value	GPU gBest Value	Speedup
50	1.6302	0.1607	0.00081	0.00004	1.26E+00	1.24E+00	10.1
100	3.4275	0.2403	0.00171	0.00007	3.36E+01	3.33E+01	14.2
150	5.3585	0.3189	0.00267	0.00010	1.01E+02	1.01E+02	16.8
200	7.8724	0.3944	0.00393	0.00013	2.58E+02	2.58E+02	19.9

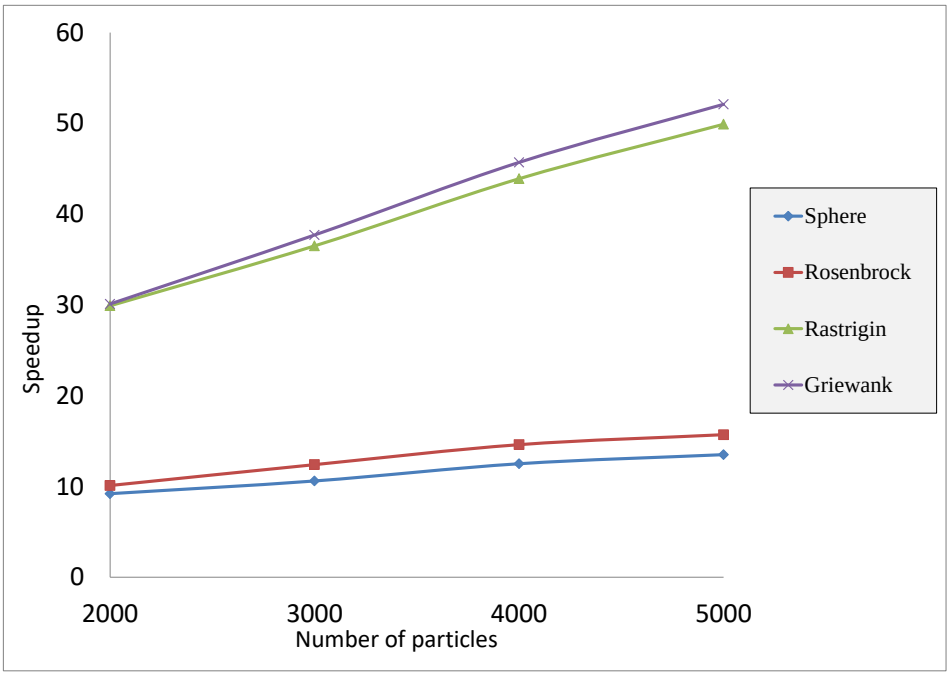

Fig. 3.7: Speedup and swarm population for f_1 *to* f_4 functions

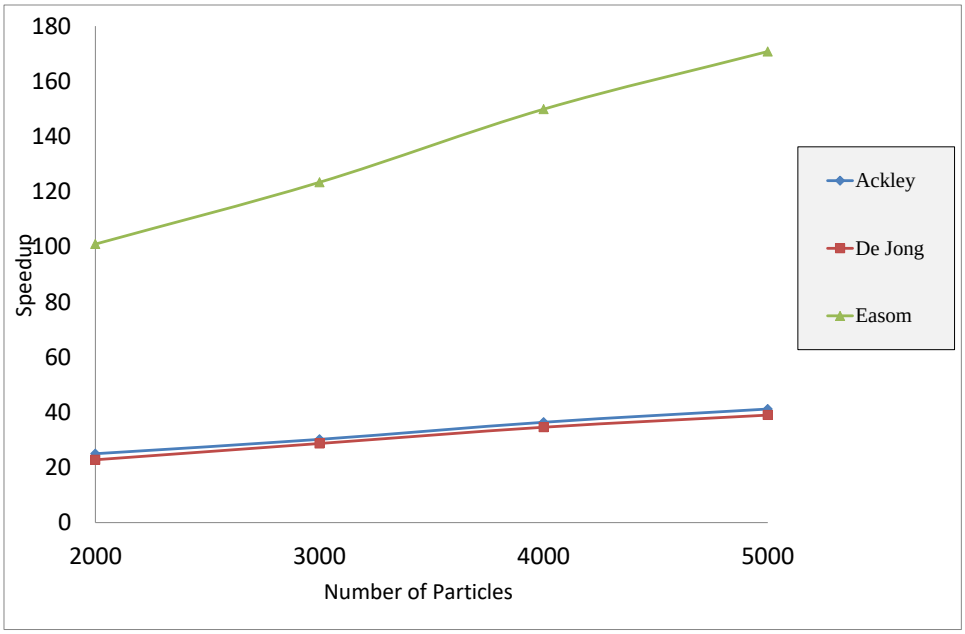

Fig. 3.8: Speedup and swarm population for f_5 *to* f_7 functions

Table 3.11: GPU SPSO and CPU SPSO on f_3 (number of particles $n = 2000$)

d	CPU Time(s)	GPU Time(s)	CPU loop Time(s)	GPU loop Time(s)	CPU gBest Value	GPU gBest Value	Speedup
50	5.3248	0.1779	0.00266	0.00004	1.18E-01	1.23E-01	29.9
100	10.5850	0.2971	0.00529	0.00009	4.38E+00	4.11E+00	35.6
150	16.7847	0.4159	0.00839	0.00014	2.18E+01	2.16E+01	40.3
200	23.3476	0.5021	0.01167	0.00018	5.57E+01	5.42E+01	46.4

Table 3.12: GPU SPSO and CPU SPSO on f_4 (number of particles $n = 2000$)

d	CPU Time(s)	GPU Time(s)	CPU loop Time(s)	GPU loop Time(s)	CPU gGest Value	GPU gBest Value	Speedup
50	5.9915	0.1984	0.00299	0.00005	7.66E-02	7.75E-02	30.1
100	11.8170	0.3345	0.00590	0.00011	5.33E-01	5.35E-01	35.3
150	19.6374	0.4658	0.00981	0.00015	6.95E-01	6.90E-01	42.1
200	24.5178	0.5716	0.01325	0.00019	1.07E+00	1.06E+00	42.8

Table 3.13: GPU SPSO and CPU SPSO on f_5 (number of particles $n = 2000$)

d	CPU Time(s)	GPU Time(s)	CPU loop Time(s)	GPU loop Time(s)	CPU gBest Value	GPU gBest Value	Speedup
50	4.4402	0.1774	0.00222	0.00004	3.38E-02	3.26E-02	25.0
100	8.3446	0.2934	0.00417	0.00009	3.47E-01	3.44E-01	28.4
150	12.4729	0.4086	0.00623	0.00014	7.49E-01	7.54E-01	30.5
200	16.6014	0.4943	0.00829	0.00018	1.07E+00	1.07E+00	33.5

Table 3.14: GPU SPSO and CPU SPSO on f_6 (number of particles $n = 2000$)

d	CPU Time(s)	GPU Time(s)	CPU loop Time(s)	GPU loop Time(s)	CPU gBest Value	GPU gBest Value	Speedup
50	4.1840	0.1829	0.00209	0.00004	8.62E-24	8.42E-24	22.8
100	10.2486	0.3141	0.00512	0.00010	8.08E-22	1.87E-21	32.6
150	17.7679	0.4540	0.00888	0.00015	1.12E-17	8.22E-19	39.1
200	25.3917	0.5668	0.01269	0.00019	8.60E-19	2.15E-19	44.7

Table 3.15: GPU SPSO and CPU SPSO on f_7 (number of particles $n = 2000$)

d	CPU Time(s)	GPU Time(s)	CPU loop Time(s)	GPU loop Time(s)	CPU gBest Value	GPU gBest Value	Speedup
50	19.2378	0.1904	0.00961	0.00005	0.00E+00	0.00E+00	101.0
100	38.2600	0.3239	0.01912	0.00011	0.00E+00	0.00E+00	118.1
150	57.2342	0.4507	0.02775	0.00016	0.00E+00	0.00E+00	126.9
200	76.7703	0.5518	0.0383	0.00020	0.00E+00	0.00E+00	139.1

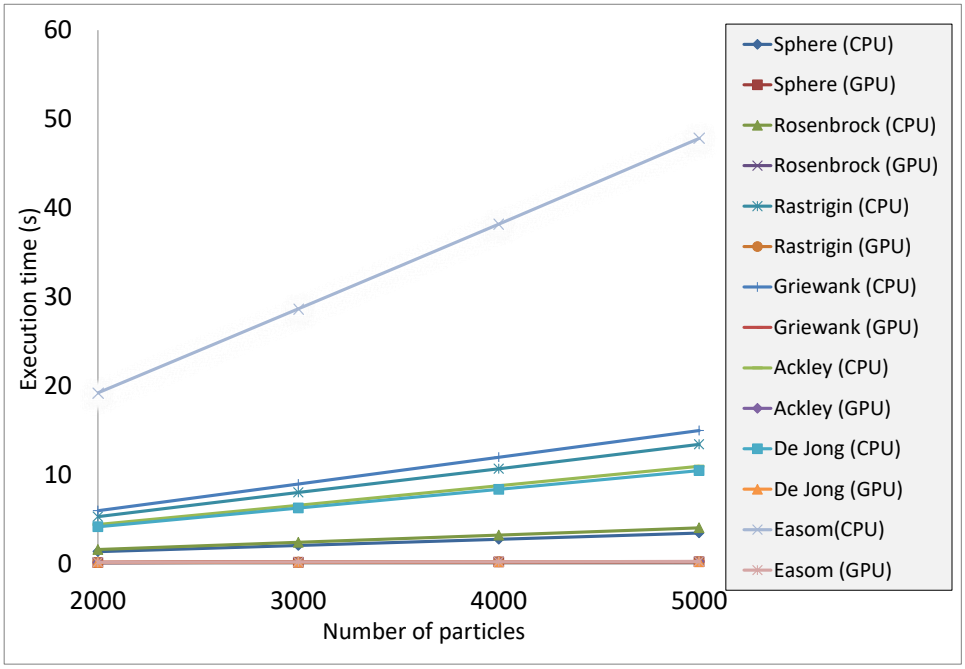

Fig. 3.9: Overlap of computation time as a function of swarm population

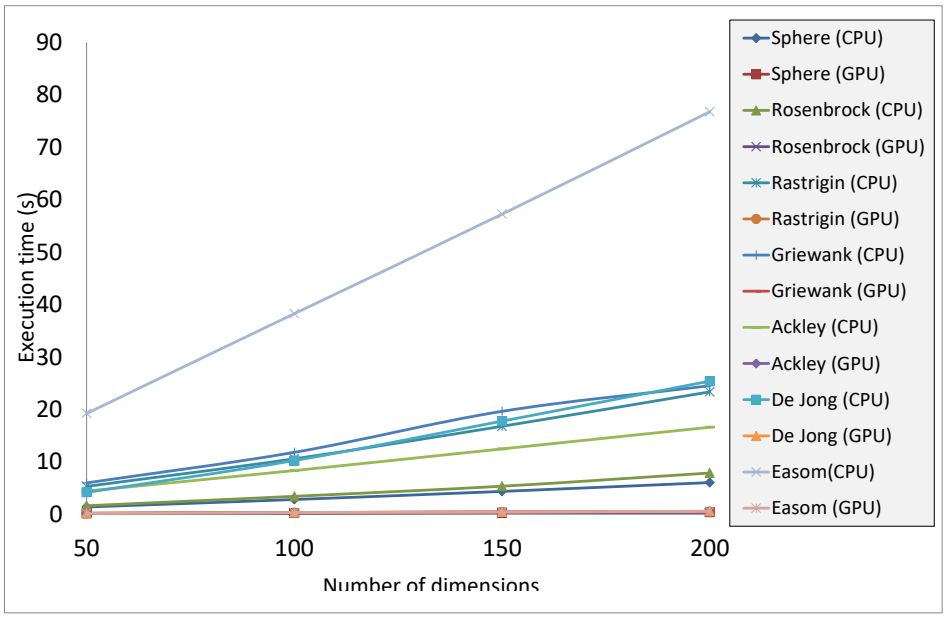

Fig. 3.10: Overlap of computation time as a function of dimension

Table 3.16: GPU SPSO and CPU SPSO on f_1 (number of particles $n = 10000$, acceptable optimization value 0.0001 and optimal value for f_1 is 0

d	CPU Time(s)	GPU Time(s)	CPU loop Time(s)	GPU loop Time(s)	Speedup
50	1.1038	0.2030	0.0045	0.0002	5.4
100	5.9410	0.4024	0.0089	0.0004	14.7
150	15.4436	1.0243	0.0133	0.0006	15.0
200	32.1552	1.7544	0.0182	0.0007	18.3

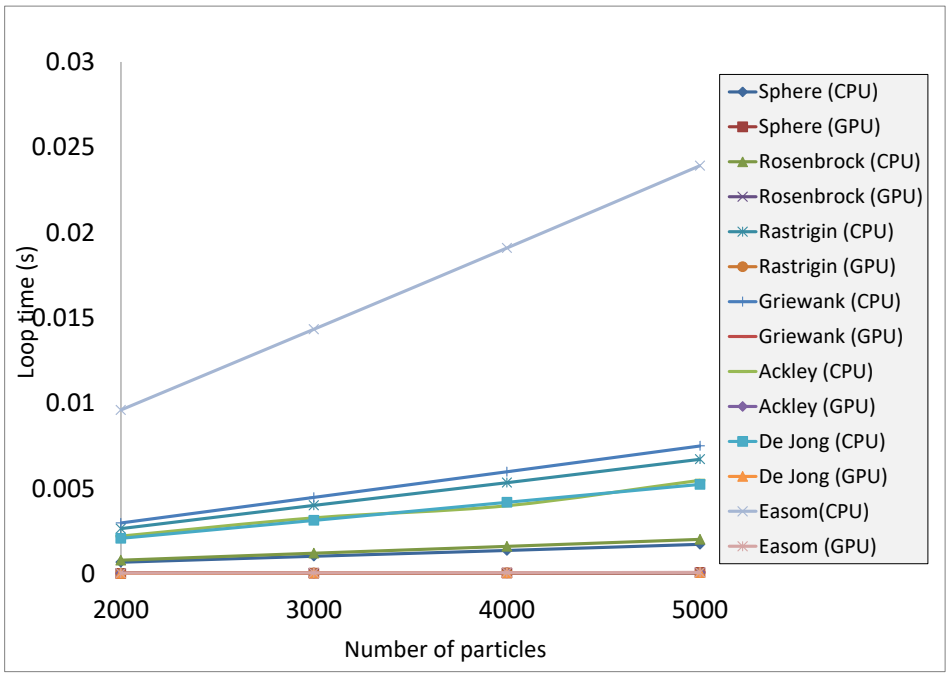

Fig. 3.11: Overlap of loop time as a function of swarm population

Table 3.17: A Comparison between [36] and ours on f_4 (number of dimensions $d = 50$)

		[36]			ours		
n	iterations	CPU Time(s)	GPU Time(s)	Speedup	CPU Time(s)	GPU Time(s)	Speedup
2000	2000	481.5713	57.6654	8.3	5.9915	0.1984	30.1
10000	10000	1269.7554	113.1295	11.2	241.5331	1.7695	136.4
20000	10000	2537.7515	221.9755	11.4	646.2705	4.3384	148.9

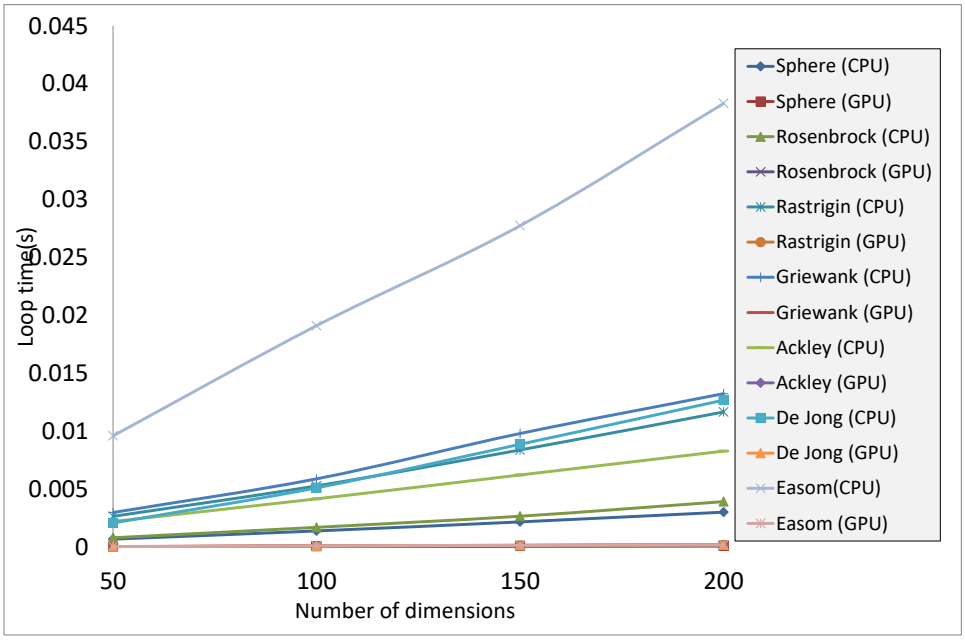

Fig. 3.12: Overlap of loop time as a function of dimension

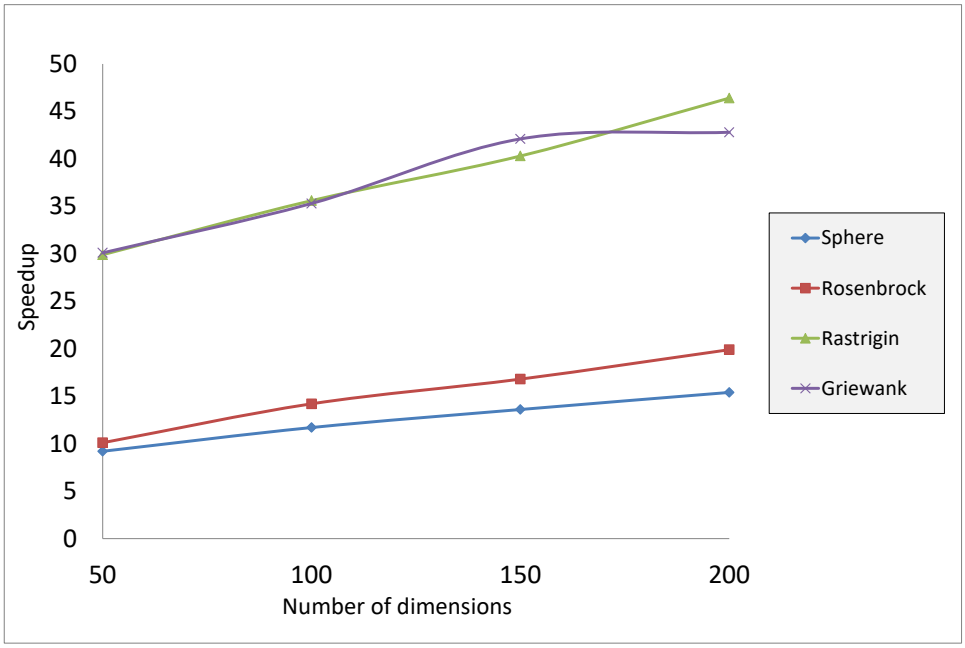

Fig. 3.13: Speedup and dimension for f_1 *to* f_4 functions

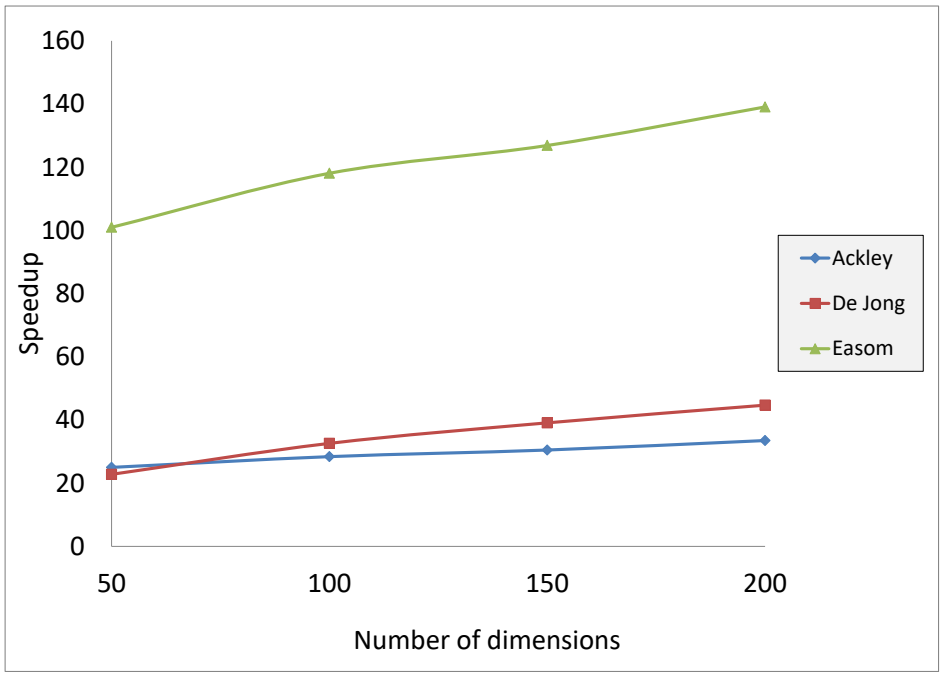

Fig. 3.14: Speedup and dimension for f_5 *to* f_7 functions

Chapter 4

Effect of the Pseudorandom Number Generators on the Standard Particle Swarm Optimization on a GPU

4.1 Introduction

A key component of particle swarm optimization algorithms is PRNGs which provide random numbers to drive the stochastic search process. The performance of SPSO algorithms is influenced by the quality of the PRNGs running on a GPU.

In our previous research paper[4], the SPSO algorithm was implemented based on task scheduling on a GPU, where many particles can reach to their best position simultaneously by using ring topology and it could significantly improved the PSO efficiency. Some random variables included in the SPSO implementation were used to initiate the positions and the velocities of the particles and to evaluate the velocity update equation as well on a GPU. These random variables were generated by a PRNG called *curandXORWOW* [30]. These random variables help to guide a more effective exploration during the search process. We achieved 46 times speedup compared to CPU SPSO on large swarm population and high dimensional problems. However, the *curandXORWOW* takes some time to setup for the first time. Therefore, large swarm population and high dimensional problems require more computation time to initialize velocity and position on a GPU.

In this chapter, we present an analysis of the performance of the various PRNGs on a GPU SPSO on the CUDA architecture. We have implemented ten PRNGs. By using a single step `TausStep` of the combined Tausworthe generator, we have achieved a desire SPSO solution with improved computational time on a GPU which have been evaluated with SPSO implemented on a CPU by using the same PRNG. The experimental results show a significant speedup of SPSO with large swarm population

and high dimensional problems.

The remainder of this paper is organized as follows. Section 4.2 our PRNG imple mentation is presented. Subsequently, experimental evaluations, obtained results and analysis are presented in Section 4.3. Finally, Section 4.4 gives some concluding remarks and open directions for future work.

4.2 Our PRNG Implementation

We implemented ten PRNGs on a GPU SPSO. Each of them is simple, qualityful and extremely fast. Between ten of them, three types xorshift were implemented by combining xorshift operations in different ways which produce periods $2^{32} - 1$ for 32-bit words, period $2^{64} - 1$ for 64-bit words and period $2^{128} - 1$ for 128-bit words. We also implemented variations of xorshift operations likes xorshift*, xorwow, multiply-with-carry (mwc). Other PRNGs of linear congruential generator (LCG), a single step `TausStep`, curandPhilox and curandXORWOW were also implemented on GPU SPSO. The important part of our random number generator implementation is that GPU SPSO does not need initial seeds from the host. All are completely CPU free. On the other hand, for CPU SPSO, all are completely CPU oriented. Our a single step `TausStep` is presented on Fig. 4.1 which is based on a CPU and on a GPU respectively. Inside this PRNG, four functions have been implemented. In these four functions, $fsrand(i, j, k, s)$ is used to set a seed s of each dimension where parameter i,j,k are a particle number, dimension, and 0 or 1 to distinguish $rand_1()$ or $rand_2()$ in the velocity update equation. Subsequently, each thread calls $frand(i, j, k)$ with the same prameters i, j, k as the parameters for $fsrand()$ call in order to obtain a uniformly random real number in [0, 1].

Each thread uses an individual seed with its thread index as a parameter. Due to the fact that the random number generators are running in each thread independently, the numbers can be generated on demand by each particle.

4.3 Experimental Evaluations

In this section, by using ten PRNGs the experimental results of an SPSO imple mentation on a GPU and on a CPU are presented in terms of speedup. Performance comparisons between PRNGs on a GPU SPSO and the same PRNGs on a CPU SPSO are made based on six classical benchmark test functions [45] shown in Table 4.1. The six functions except Easom always return non-negative values. In contrast, Easom always returns a non-positive value. Since our atomicMin() implementation behaves as atomicMax() for non-positive values as explained in Section 3.3, the aims of our experiments are to minimize the six functions and to maximize Easom function although all the seven functions are originally for minimization. For all PRNGs on

```
1  __host__ __device__void taus_rand_seed(unsigned*seed,unsigned s)
2  {
3      *seed =s;
4  }
5  __host__ __device__ unsigned taus_rand(unsigned* seed)
6  {
7      unsigned int b = (((*seed << 6) ^ *seed) >> 13);
8      *seed = (((*seed &4294967294UL) << 18) ^ b);
9      return *seed;
10 }
11 __host__ __device__ unsigned *seeds ;
12 __host__ __device__ int seeds_d ;
13 __host__ __device__ void fsrand(int i,int j,int k,unsigned seed)
14 {
15     taus_rand_seed( &seeds[(i * 2 * seeds_d+j+k* seeds_d)],seed);
16     // set seed for frand()
17 }
18 __host__ __device__ float frand(int i, int j, int k)
19     // return a uniformly random real number in [0, 1]
20 {
21     return (float) taus_rand((unsigned (*))
22      &seeds[ (i * 2 * seeds_d + j + k * seeds_d)]) / UINT_MAX;
23 }
```

Fig. 4.1: a single step `TausStep` of the combined Tausworthe generator implementation on a GPU and on a CPU.

a GPU SPSO, good fitness values are obtained which shows the reliability of our results.

Table 4.1: Benchmark Test Functions[45] for minimization

Name	Equation	Bounds	Optimal value
Sphere	$f_1(x_1, x_2, \cdots, x_d) = \sum_{i=1}^{d} \ x_i^2$	$(-5.12, 5.12)^d$	0
Rosenbrock	$f_2(x_1, x_2, \cdots, x_d) = \sum_{i=1}^{d} \ (100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$	$(-10, 10)^d$	0
Rastrigin	$f_3(x_1, x_2, \cdots, x_d) = \sum_{i=1}^{d} [x_i^2 - 10 * \cos(2\pi x_i) + 10]$	$(-5.12, 5.12)^d$	0
Griewank	$f_4(x_1, x_2, \cdots, x_d) = \frac{1}{4000} \ \sum_{i=1}^{d} x_i^2 - \prod_{i=1}^{d} \cos(\frac{x_i}{\sqrt{i}}) + 1$	$(-600, 600)^d$	0
Ackley	$f_5(x_1, x_2, \cdots, x_d) = -20 \exp[-\frac{1}{5}\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}] - \exp[\frac{1}{d}\sum_{i=1}^{d} \cos(2\pi x_i)] + 20 + e$	$(-32.768, 32.768)^d$	0
Easom	$f_6(x_1, x_2, \cdots, x_d) = -(-1)^d(\prod_{i=1}^{d} \cos^2(x_i)) \exp[-\sum_{i=1}^{d}(x_i - \pi)^2]$	$(-2\pi, 2\pi)^d$	0

Table 4.2: Impact of the PRNGs on SPSO speedup (number of particles $n = 2000$, number of iteration 2000) for f_1

d	xorshift xor32	xorshift xor64	xorshift xor64*	xorshift xor128	xorshift xorwow	mwc	LCG	a single `TausStep`	curand Philox	curand XORWOW
50	35.8	36.2	37.7	23.1	15.0	24.1	38.1	37.3	12.7	12.5
100	46.6	48.1	47.4	28.1	17.7	29.0	48.4	48.0	14.4	6.6
150	53.9	54.0	54.4	31.5	18.3	33.3	55.4	55.4	14.8	5.3
200	67.4	68.5	68.7	37.5	22.7	39.8	68.7	69.5	17.7	5.5

Table 4.3: Impact of the PRNGs on SPSO speedup (number of dimensions $d = 50$, number of iteration 2000) for f_1

n	xorshift xor32	xorshift xor64	xorshift xor64*	xorshift xor128	xorshift xorwow	mwc	LCG	a single step `TausStep`	curand Philox	curand XORWOW
2000	35.8	36.2	37.7	23.1	15.0	24.1	38.1	37.3	12.7	12.5
3000	43.2	44.0	43.9	27.4	16.9	29.1	45.3	44.5	13.9	9.5
4000	50.7	52.6	51.7	30.4	18.0	31.3	53.0	52.3	14.7	6.5
5000	54.8	55.8	56.0	31.9	18.6	32.9	57.0	56.5	15.1	5.4

4.3.1 Experimental Result

In all experiments, we considered large particles and dimensions. The number of dimensions and particles were respectively set from 32 to 256 and 2000 to 10000. Each experiment was run until the maximum number of iterations has been reached, which was set to 2000. By using each PRNG, the GPU SPSO was run on from f_1 to f_6 functions for 50 times independently with different seeds. In the same way, the CPU SPSO was also run on from f_1 to f_6 functions with the same PRNG for 50 times independently with different seeds. The average speedup result of f_1 and f_6 functions are shown in from Tables 4.2 to 4.5.

4.3.2 Experimental Analysis

Analyzing the data of the tables, we can observe that in Table 4.4, a GPU SPSO can reach maximum 175 times speedup using a single step `TausStep` of PRNGs when the

Table 4.4: Impact of the PRNGs on SPSO speedup (number of particles $n = 2000$, number of iteration 2000) for f_6

d	xorshift xor32	xorshift xor64	xorshift xor64*	xorshift xor128	xorshift xorwow	mwc	LCG	a single step `TausStep`	cuRAND Philox	curand XORWOW
50	121.4	121.0	123.9	80.5	56.3	85.3	123.9	125.6	47.1	46.2.
100	149.3	150.0	152.7	97.2	67.1	104.1	151.8	152.2	55.0	27.0
150	159.5	159.8	161.6	105.4	66.8	112.3	161.2	161.7	54.6	21.1
200	172.6	173.3	174.5	109.0	71.6	116.9	174.5	175.0	56.7	19.0

Table 4.5: Impact of the PRNGs on SPSO speedup (number of dimensions $d = 50$, number of iteration 2000) for f_6

n	xorshift xor32	xorshift xor64	xorshift xor64*	xorshift xor128	xorshift xorwow	mwc	LCG	a single step `TausStep`	curand Philox	curand XORWOW
2000	121.4	121.0	123.9	80.5	56.3	85.3	123.9	125.6	47.1	46.2
3000	149.4	150.2	153.6	100.1	64.4	105.7	152.9	154.0	53.3	37.1
4000	183.0	183.2	186.9	114.2	70.4	121.8	184.7	187.6	57.5	26.2
5000	207.2	207.6	211.3	123.9	74.0	132.7	207.6	212.8	60.0	22.1

Table 4.6: SPSO gBest value (number of particles $n = 2000$, number of iteration 2000) for f_1

d	xorshift xor32	xorshift xor64	xorshift xor64*	xorshift xor128	xorshift xorwow	mwc	LCG	a single step `TausStep`	curand Philox	curand XORWOW
50	2.76E-15	7.66E-13	9.33E-12	3.09E-12	1.64E-13	3.27E-02	1.64E-14	1.33E-04	8.99E-12	8.65E-12
100	2.07E-08	1.63E-05	1.21E-04	5.37E-05	7.44E-06	7.40E-01	2.56E-07	9.89E-03	1.36E-04	1.39E-04
150	1.63E-05	5.63E-03	6.24E-02	2.55E-02	4.29E-03	2.23E+00	1.34E-04	5.58E-02	6.94E-02	6.52E-02
200	2.89E-04	9.42E-02	1.75+00	7.03E-01	1.34E-01	4.81E+00	3.65E-03	1.60E-01	2.40E+00	2.30E+00

swarm population and dimension size are respectively set to 2000 and 200, running on a complex function. In the next test, the swarm population is set from 2000 to 5000 and the dimension is set to 50. The results shown in Table 4.5 demonstrate that running a GPU-SPSO can reach maximum 212 times speedup for a single step `TausStep` of PRNGs when the swarm population size is set to 5000 running on a complex function. For more complex functions speed up may be even greater.

Analyzing the data of the tables, we have been obtained that $curandXORWOW$, $curandPhilox4\_32\_10\_t$ of PRNGs on a GPU were taken more time to optimize large particles and high dimensional problems. On the others hand, the speed can be greatly accelerated for $xorshift$ RNGs, linear congruential generator, a single step `TausStep` on a GPU. Moreover, a single step `TausStep` is faster than $xorshift$ RNGs and linear congruential generator (see Figs. 4.2 to 4.3) to be more exact. In GPU SPSO, we achived good optimization value (see Tables 4.6 to 4.9) and the speedup was greatly accelerated in the case of high dimension and large particles running on a single step `TausStep` of PRNG. However, when dimension and particle size are small, speedups were also accelerated moderately (see Table 4.10).

Table 4.7: SPSO gBest value (number of dimensions $d = 50$, number of iteration 2000) for f_1

n	xorshift xor32	xorshift xor64	xorshift xor64*	xorshift xor128	xorshift xorwow	mwc	LCG	a single step `TausStep`	curand Philox	curand XORWOW
2000	2.76E-15	7.66E-13	9.33E-12	3.09E-12	1.64E-13	3.27E-02	1.64E-14	1.33E-04	8.99E-12	8.65E-12
3000	1.96E-15	6.94E-13	8.30E-12	2.58E-12	1.27E-13	3.00E-02	1.42E-14	1.07E-04	7.88E-12	8.20E-12
4000	1.88E-15	6.62E-13	7.63E-12	2.73E-12	9.63E-14	2.82E-02	1.36E-14	9.47E-05	7.76E-12	7.65E-12
5000	1.81E-15	6.21E-13	7.24E-12	2.43E-12	1.03E-13	2.63E-02	1.28E-14	8.18E-05	7.32E-12	7.46E-12

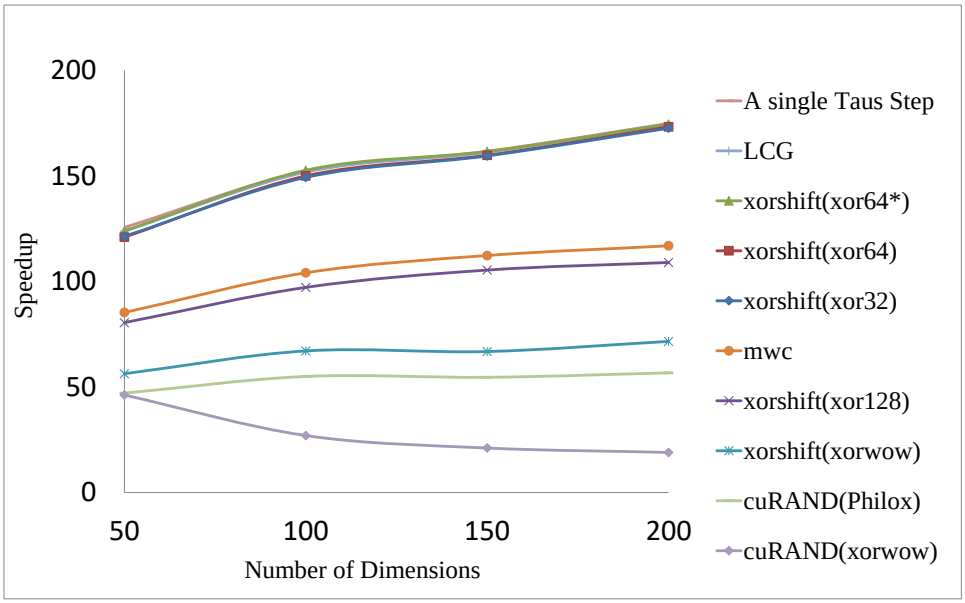

Fig. 4.2: Speedup (number of Particles = 2000, number of iterations= 2000)

In this chapter, we presented an analysis of the performance of the PRNGs on GPU SPSO in terms of the PRNGs statistical quality and good parallelized implementation on a SPSO. We showed that a single step `TausStep` of PRNG for GPU presents enough quality to be used by the SPSO algorithm. We also achieved speedup and execution time on a GPU SPSO are increased due to the overhead of memory access. This proposed GPU SPSO algorithm can be used to improve required time to solve optimization problems on large swarm population and high dimensional.

4.4 Summary

This chapter has presented an investigation about the influence of the quality of PRNGs on the performance of the SPSO algorithm running on a GPU based on CUDA architecture. The experimental results demonstrated that a single step `TausStep` of PRNGs on GPU SPSO significantly reduces execution time compared to previous development. By using a single step `TausStep` of PRNGs, we have achieved a good fitness value with short execution time. The proposed implementation is 307

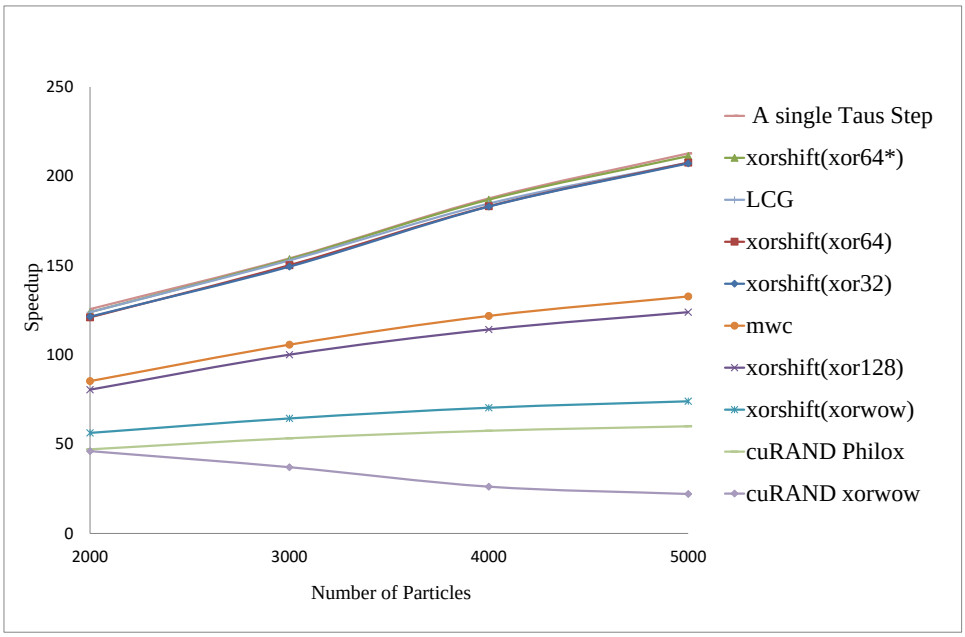

Fig. 4.3: Speedup (number of dimensions = 50, number of iterations= 2000)

times faster than CPU SPSO. This implementation has a positive impact on the performance of the optimization of large dimension and large swarm population problems. In our future work, we want to investigate about the influence of the quality of PRNGs on the performance of the multi-objective particle swarm optimization (MOPSO) and many objective particle swarm optimization.

Table 4.8: SPSO gBest value (number of particles $n = 2000$, number of iteration 2000) for f_6

d	xorshift xor32	xorshift xor64	xorshift xor64*	xorshift xor128	xorshift xorwow	mwc	LCG	a single step `TausStep`	curand Philox	curand XORWOW
50	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
100	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
150	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
200	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00

Table 4.9: SPSO gBest value (number of dimensions $d = 50$, number of iteration 2000) for f_6

n	xorshift xor32	xorshift xor64	xorshift xor64*	xorshift xor128	xorshift xorwow	mwc	LCG	a single step `TausStep`	curand Philox	curand XORWOW
2000	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
3000	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
4000	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
5000	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00

Table 4.10: speedup for large swarm population and high dimensional problems on a SPSO (number of iterations = 10000)

d	n	$Sphere(f_1)$	$Rosenbrock(f_{2)}$	$Rastrigin(f_3)$	$Griewank(f_4)$	$Ackley(f_5)$	$Easom(f_6)$
32	128	10.4	4.1	7.5	8.2	8.0	12.1
64	1024	56.4	53.6	61.6	62.8	66.9	118.1
256	10000	104.7	124.3	127.7	134.3	130.9	307.9

Chapter 5

GPU-based Parallel Multi-objective Particle Swarm Optimization for Large Swarms and High Dimensional Problems

5.1 Introduction

This chapter presents a new serial implementation of MOPSO (CPU MOPSO) and a new GPU-parallelized implementation of MOPSO (GPU MOPSO) based on a master-slave model for large swarms and high dimensional optimization problems. The proposed parallel implementation of MOPSO 157 times speedup compared to the corresponding CPU implementation. We achieved our faster implementation by using coalescing memory access, a fast pseudorandom number generator, Thrust library, CUB library, an atomic function, parallel archiving and so on.

The remainder of this chapter is organized as follows. In Section 5.2, we provide our MOPSO implementations on a CPU and a GPU. Subsequently, in Section 5.3, we present and analyze experimental results and compare our implementation with the previous implementation in terms of execution time and speedup. After that, Section 5.4 summarizes some related works. Finally, in Section 5.5, we give some concluding remarks and point out directions for future work.

5.2 Our Implementations of MOPSO

A CPU MOPSO is implemented to evaluate its performance and execution time. In the CPU MOPSO, the velocity and position of each particle are updated by using the update equations in Section 2.4. The personal best (`pBest`) position and fitness value are obtained by adjusting the previous personal best position and fitness value. At

each iteration, an archive is updated with respect to the contents of the personal best repository. At the beginning of each iteration, the archive is empty. The personal best of the first particle is added to the archive when the archive is empty. After that, other personal bests nondominanted by the personal best of the first particle are added to the archive. In our CPU MOPSO implementation, this is called our archiving technique.

After completion of all iterations, the archive is returned as an outcome from which a Pareto optimal set is constructed. Our CPU MOPSO uses six loops which require a lot of time to run. Implementing this part of the program in parallel using a GPU platform can reduce execution time significantly. In our CPU MOPSO implementation, `thrust::stable_sort` is used for sorting in the ascending order of f_2 [47]. The Pareto fronts generated by our CPU MOPSO are quite close to the true Pareto fronts for large swarms and high dimensional problems.

We utilize a simple, effective and quality archiving technique. This technique starts with the first particle selection and finishes with the Pareto optimal set construction from the final archive. The details of the same technique on a GPU is described in the following explanation of our GPU MOPSO.

We use the master-slave model in our GPU MOPSO implementation. A CPU handles a master and a GPU handles slaves. Our GPU MOPSO is delineated in Fig. 5.1 and Fig. 5.2.

The main process of MOPSO implementation on a GPU by using the master-slave model is as follows.

Step 1: This step allocates memory on a GPU with 1 block of 1 thread and initializes the basic information on each particle such as the position and the velocity with n blocks of d threads. In order to generate random numbers on a GPU, we use `TausStep` for each thread with an independent seed number. In our implementation each particle is mapped onto a distinct thread block and each dimension is mapped onto a distinct thread. Fig. 5.3 shows our kernel code for initialization where function `fsrand(i, j, k, seed)` (resp. `frand(i, j, k)`) sets a seed of the PRNG (resp. generates a random number) for the thread allocated to (particle `i`, dimension `j`). Each thread requires two sequences of random numbers. The parameter `k` distinguishes the two sequences (`k` $\in \{0, 1\}$). In Fig. 5.3, `x[ ]` , `v[ ]`, and `pBestValue[ ]` store positions, velocities, and the personal best values of particles respectively.

In our previous work [4] of the Standard Particle Swarm Optimization (SPSO) on a GPU, the curandXORWOW [30] was used for each thread with an independent seed number. The curandXORWOW focuses on efficient generation of high-quality pseudorandom and quasirandom numbers. However, it is a little bit complex and generating a random number requires more computations. Therefore, updating the XORWOW state takes more time. Our new implementation uses the PRNG TausStep. TausStep has a positive impact on the performance to solve high dimensional two objective optimization problems with a large swarm population. For

MASTER

let n = number of particles; let d = number of dimensions
allocate memory on a GPU with 1 block of 1 thread.
generate random number seeds for `TausStep` & initialize velocity and position of each particle on a GPU with n blocks of d threads. } **Step1**

for $i = 1$ to iterations do
  calculate fitness values and pBests on a GPU with $(n + 256 - 1)/256$ blocks of 256 threads. } **Step 2**
  select all personal bests nondominated by the personal best of the first particle on a GPU with $(n + 256 - 1)/256$ blocks of 256 threads. } **Step 3**
  using cub::DeviceScan::ExclusiveSum in advance, compact the selected personal bests into a dense array `archive` on a GPU with $(n + 256 - 1)/256$ blocks of 256 threads. } **Step 4**
  update velocities and positions on a GPU with n blocks of d threads. } **Step 5**
end;
stable_sort `archive` in the ascending order of f_2 using thrust::stable_sort
construct a Pareto optimal set from `archive` on a GPU with 1 block of 1 thread. } **Step 6**
free seeds using 1 block of 1 thread

Fig. 5.1: The master in our GPU MOPSO.

this initialization, coalescing memory access is used. The arrays on the VRAM are arranged to realize coalescing memory access, in particular the array for random number seeds.

Step 2 : This step calculates fitness values and personal bests on a GPU with $(n + 256 - 1)/256$ blocks of 256 threads by adjusting the previous personal best positions and values of the two objectives. Here, each slave is mapped on a thread. Fig. 5.4 shows our kernel code for calculating fitness values and personal bests where `pBestPos[ ]` stores personal best positions. Function `poscpy(int d, float *dst, float *src)` copies `src[0..d-1]` to `dst[0..d-1]` in serial. Functions `f1` and `f2` denote the first and the second of the two objective functions respectively.

Step 3 : This step selects the personal bests to be placed in the archive by computing a flag array `archiveflag` on a GPU with $(n + 256 - 1)/256$ blocks of 256 threads. In this process, the personal best of the first particle is compared with the personal bests of the other particles. After comparisons, the personal bests nondominated by the first particle and the first particle itself are marked with the flag "1" which means that their values are stored in Step 4 to a dense array `archive` and finally returned to the master. The kernel code of this step is shown in Fig. 5.5 where function `dominates(x, y)` returns 1 if a solution `x` dominates a solution `y` otherwise returns 0.

Step 4 : In this step, on a GPU with $(n + 256 - 1)/256$ blocks of 256 threads, from a sparse array `archiveflag`, a dense array `archive` of type `key = thrust::tuple (float, float, float *)` is computed. The dense array `archive` stores together with pBestPos[ ] the fitness values and positions of the personal bests selected in Step 3. The first (resp. second) element of a tuple stores the fitness value of the first (resp. second) objective function. The third element of a tuple stores the pointer to the corresponding element in `pBestPos[ ]`. This array implements the archive on a

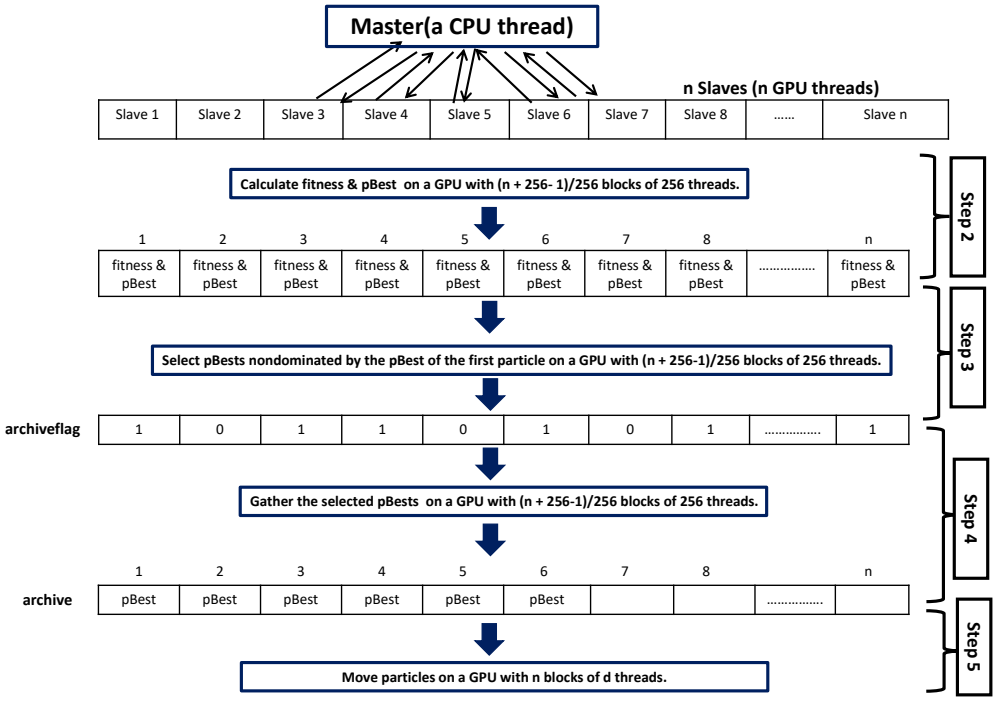

Fig. 5.2: The slaves in our GPU MOPSO.

GPU. Together with Step 4, Step 3 is called our GPU MOPSO archiving technique. In this process, each flag value has an important role. The personal bests only with flag value "1" are stored in `archive`. Here, prefix-sums is used to find out the location of each personal best in `archive`. In our implementation, device-wide exclusive prefix-sum `cub::DeviceScan::ExclusiveSum` in the CUB library is used. `archive` is used in Step 5 for computing the new velocity and position of each particle. In each iteration, `archive` is updated from the beginning. The kernel code is shown in Fig. 5.6 where `archivedst[i]` stores the index of `archive` at which the particle `i` should be placed and `archivesrc[i]` stores the index of a particle such that the particle `archivesrc[i]` is placed in `archive[i]`. Here, `atomicAdd` [17] library function is used for counting the number of the solutions in the archive where `archiveSize` is the counter.

Step 5 : This step moves all particles according to the velocity update equation and the position update equation in Section 2.4 on a GPU with n blocks of d threads. In our implementation, the leader is selected for each particle randomly from the archive. Fig. 5.7 shows our kernel function for computing the new velocity and position of each particle on a GPU where `taus_rand(i, j, k)` returns a random unsigned integer using the same state as `frand(i, j, k)`.

Step 6 : After completion of all iterations, one of the slaves collects nondominated

```
1  __global__ void initialize(int d, int n, float *x, float *v,
2                float *pBestValue, int seed, float domain)
3  {
4      int i = blockIdx.x, int j = threadIdx.x;
5
6      fsrand(i, j, 0, 1 + 2 * (i * d + j) + 0 + seed);
7      fsrand(i, j, 1, 1 + 2 * (i * d + j) + 1 + seed);
8
9      x[i * d + j] = domain * frand(i, j, 0);
10     v[i * d + j] = domain * frand(i, j, 1);
11
12     if (j == 0) {
13         pBestValue[i * 2 + 0] = FLT_MAX;
14         pBestValue[i * 2 + 1] = FLT_MAX;
15     }
16 }
```

Fig. 5.3: Our kernel function for initialization (Step 1).

solutions from the final `archive` on a GPU with 1 block of 1 thread, assuming that `archive` is sorted in the ascending order of f_2. Nondominated solutions are returned as the outcome and are used to construct a Pareto front.

In our implementation,`thrust::stable_sort` is used for sorting in the ascending order of f_2 on a GPU. Fig. 5.8 shows our kernel code for generating the `Pareto optimal set` where `ParetoOptimalSetValue[ ]` and `ParetoOptimalSetPos[ ]` store the values and positions of the Pareto optimal set respectively. Here `count` stores the number of solutions in the Pareto optimal set.

5.3 Experimental Evaluations

In this section, we present our experimental results which have been obtained using a CPU platform and a GPU platform. The experimental results are obtained in terms of speedup. The speedup is defined as the ratio of the execution time of the sequential implementation to the execution time of the parallel implementation. Our GPU MOPSO has been implemented using a single step of the combined Tausworthe generator as a PRNG and the results have been compared with our CPU MOPSO which uses the same PRNG. Performance comparisons are conducted based on four classical benchmark test functions with two objectives shown in Table 5.1 [16][48][49]. The CPU MOPSO and the GPU MOPSO are run using the same configuration of parameters n and d, which are the numbers of particles and the dimensions respectively.

```
1  __global__ void evaluate_particles(int d, int n, float *x,
2            float *pBestValue, float *pBestPos)
3  {
4      int i = blockDim.x * blockIdx.x + threadIdx.x;
5      if (i >= n) return;
6
7      float pValue0 = f1(d, &x[d * i]), pValue1= f2(d, &x[d * i]);
8
9      if (dominates(pValue0, pValue1, &pBestValue[i * 2])) {
10         pBestValue[i * 2 + 0] = pValue0;
11         pBestValue[i * 2 + 1] = pValue1;
12         poscpy(d, &pBestPos[d * i], &x[d * i]);
13     }
14 }
```

Fig. 5.4: Our kernel function for computing fitness values and personal bests (Step 2).

5.3.1 Environment

Tests were conducted using our GPU and CPU server. Our GPU server has an NVIDIA TITAN V (5120 cores, 12GB VRAM)[50], a 2.8GHz Intel Core i7-6700T (8MB L3 cache, 4 physical cores), 16GB main memory, and Windows 10 Professional. Our CPU server has a 3.0GHz Intel Xeon E3-1220V5 (8MB L3 cache, 4 physical cores)[51], 16GB main memory, and Windows Server 2012 R2. For compilation, we used Microsoft Visual Studio 2017 Professional Edition and CUDA 10 SDK.

5.3.2 Experimental Result

In all experiments the numbers of dimensions and particles were respectively set from 30 to 1024 and from 1024 to 32768. Each experiment was run until the maximum number of iterations has been reached, which was set from 250 to 2500. The GPU MOPSO and CPU MOPSO were run for the four test functions with two objectives (f_1 and f_2) 10 times independently with different seeds. The average speedup results of the four functions are shown in Tables 5.2 to 5.11.

5.3.3 Experimental Analysis

Analyzing the data in Tables 5.2 through 5.5 and Figs. 5.9 to 5.12 we observe that a large swarm improves the quality and size of the obtained nondominated solutions which creates a good Pareto front. In these cases, the numbers of particles, dimensions, and iterations are 1024 to 32768, 30, and 2500 respectively. On the

```
1  __global__ void select_pBests(int n, float *pBestValue,
2          int *archiveflag))
3  {
4       int i = blockDim.x * blockIdx.x + threadIdx.x;
5       if (i >= n) return;
6
7       if ((pBestValue[0] == pBestValue[i * 2 + 0])
8             && (pBestValue[1] == pBestValue[i * 2 + 1])) return;
9       if (!dominates(&pBestValue[0], &pBestValue[i * 2]))
10            archiveflag[i] = 1;
11      else archiveflag[i] = 0;
12 }
```

Fig. 5.5: Our kernel function for selecting personal bests to be placed in the archive (Step 3).

other hand, in Tables 5.6 through 5.9, we observe that the number of nondominated solutions is decreased with an increase of the number of dimensions. It's a common phenomena that we have difficulty if the number of dimensions is increased while the number of particles is fixed. However, Pareto fronts of both CPU MOPSO and GPU MOPSO implementations are very close to the true Pareto fronts as shown in Figs. 5.13 and 5.14. Here, the numbers of dimensions, particles, and iterations are 256, 8192, and 2500 respectively. In this experiment, the CPU MOPSO and the GPU MOPSO were run for Test functions 1 to 4. Moreover, we include Figs. 5.15 through 5.18 where the numbers of dimensions and particles are large at the same time. Here, we observe that the obtained Pareto fronts are very close to the true Pareto fronts.

More remarkable point of our implementation is that we can handle large volume cases faster as well which are more important for optimization problems. For this reason, analyzing the data of the tables, we observe that in Table 5.11, the GPU MOPSO can reach maximum 157 times speedup for Test function 3 when the swarm population and the number of dimensions are respectively large.

The experimental results have been compared with an existing GPU parallel implementation [9] using the same configuration to obtain a better comparative result. However, their tests were conducted by using an NVIDIA GeForce 9800 GT [52] GPU and an Intel Core 2 Duo, 2.20 GHz CPU. The operating system was Windows XP Professional. Both platforms are older than our experimental platforms. The comparative results are presented in Table 5.10 and they demonstrate that our GPU implementation is faster when the swarm population size is 4096 and the number of dimensions are increased from 100 to 200. Our GPU implementation increases the speed up-to 28 times for Test function 4.

Our implementation runs fast even when the swarm size and the number of

```
1  __global__ void make_archive(int d, int n, float *pBestValue,
2  float *pBestPos,key *archive, int *archivedst, int *archivesrc,
3            int *archiveflag)
4  {
5      int i = blockDim.x * blockIdx.x + threadIdx.x;
6      if (i >= n) return;
7      if (archiveflag[i] == 1) {
8          archive[archivedst[i]].get<0>()
9            = pBestValue[archivesrc[archivedst[i]] * 2 + 0];
10         archive[archivedst[i]].get<1>()
11           = pBestValue[archivesrc[archivedst[i]] * 2 + 1];
12         archive[archivedst[i]].get<2>()
13           = &pBestPos[archivesrc[archivedst[i]] * d];
14         atomicAdd(&archiveSize, 1);
15     }
16 }
```

Fig. 5.6: Our kernel function for archiving on a GPU (Step 4).

dimensions are increased simultaneously (See Tables 5.2 to 5.10). Furthermore, our CPU MOPSO is also fast due to a simple archiving technique. Our CPU implementation works well for large swarms and high dimensional problems. In our CPU MOPSO, the execution time depends on the function type and how many loops are used inside the implementation. In addition, it depends on the number of iterations. In our GPU MOPSO, the execution time depends on kernels and the number of iterations. Moreover, coalescing memory access has an important effect which makes our GPU MOPSO implementation faster for large swarms and high dimensional problems. The test results for the four test functions demonstrated that the execution times are almost the same when the number of iterations is fixed. Therefore, the execution time curves for the four test functions by our GPU MOPSO in Figs. 5.19 and 5.20 are overlapped with each other. Our GPU MOPSO implementation effectively speeds up for large swarms and high dimensional problems. (See Figs. 5.21 and 5.22)

In a nutshell, in our implementation the best key point is that we achieved efficient parallelization. We presented a new approach to archive handling. In our implementation, a single step of the combined Tausworthe generator of PRNGs for GPUs presents enough quality Pareto fronts. We found more speedup with good nondominated solutions when the swarm size and the number of dimensions are simultaneously larger (See Table 5.11). The proposed GPU MOPSO can be used to improve execution time to solve high dimensional optimization problems on a large swarm population.

```
1  __global__ void move_particles(int d, int n, float *x, float *v,
2            float *pBestPos, int *archivesrc)
3  {
4      const float W = 0.729f, C1 = 1.4595f, C2 = 1.4595f;
5      int i = blockIdx.x, int j = threadIdx.x;
6
7      __shared__ int leader;
8      if (threadIdx.x == 0)
9          leader = archivesrc[taus_rand(i, 0, 1) % archiveSize];
10     __syncthreads();
11
12     v[d * i + j] = W * v[d * i + j] + C1 * frand(i, j, 0) *
13                    (pBestPos[d * i + j] - x[d * i + j])
14                                        + C2 * frand(i, j, 1) *
15                    (pBestPos[leader * d + j] - x[d * i + j]);
16     x[d * i + j] += v[d * i + j];
17
18     if (x[d * i + j] < 0.0f) x[d * i + j] = 0.0f;
19     else if (x[d * i + j] > 1.0f) x[d * i + j] = 1.0f;
20 }
```

Fig. 5.7: Our kernel function for computing a new velocity and position of each particle on a GPU (Step 5).

5.3.4 The Bottleneck on a CPU and on a GPU

In this section, we show profiling results of our CPU and GPU programs for Test function 4 as two tables. For these experiments, the numbers of dimensions, particles and iterations were respectively set to 30, 2000, and 2500. Table 5.12 is the profiling result by the Microsoft Visual Studio CPU profiling tool. Here we observe that the bottleneck of our CPU MOPSO is the second objective function `f2`. Table 5.13 is the profiling result by the NVIDIA Nsight profiling tool which provides a breakdown of the execution time of our GPU MOPSO implementation. Here, we observe that `cub::DeviceScanKernel` is the bottleneck of our GPU MOPSO implementation.

```
1  __global__ void calculate_ParetoOptimalSet(float *Pareto
2  OptimalSetValue, float *ParetoOptimalSetPos,key *archive, int d)
3  {
4      // precondition: archive[] is sorted in the ascending
5              order of f2.
6      ParetoOptimalSetValue[0] = archive[0].get<0>();
7      ParetoOptimalSetValue[1] = archive[0].get<1>();
8      poscpy(d, &ParetoOptimalSetPos[0], archive[0].get<2>());
9      count = 1;
10     for (int i = 1; i < archiveSize; i++) {
11         if (archive[i].get<0>() <
12                 ParetoOptimalSetValue[(count - 1) * 2 + 0]) {
13             ParetoOptimalSetValue[count * 2 + 0]
14                                 = archive[i].get<0>();
15             ParetoOptimalSetValue[count * 2 + 1]
16                                 = archive[i].get<1>();
17             poscpy(d, &ParetoOptimalSetPos[count * d],
18                                   archive[i].get<2>());
19             if (!((ParetoOptimalSetValue[(count - 1) * 2 + 0]
20                     == ParetoOptimalSetValue[count * 2 + 0])
21                 && (ParetoOptimalSetValue[(count - 1) * 2 + 1]
22                     == ParetoOptimalSetValue[count * 2 + 1])))
23                 count++;
24         }
25     }
26 }
```

Fig. 5.8: Our kernel function for generating a Pareto optimal set on a GPU (Step 6).

Table 5.1: Four classical benchmark test functions

$Test function$	$Objective\ 1$	$Objective\ 2$	$Bounds$
1	$f_1 = x_1$	$f_2 = gh;$ $where, g = 1 + 9.0 \sum_{i=2}^{d} (\frac{x_i}{d-1});$ $h = 1 - (\frac{f_1}{g})^2$	$[0, 1]$
2	$f_1 = x_1$	$f_2 = gh;$ $where, g = 1 + 9.0 \sum_{i=2}^{d} (\frac{x_i}{d-1});$ $h = 1 - \sqrt{\frac{f_1}{g}}$	$[0, 1]$
3	$f_1 = x_1$	$f_2 = gh;$ $where, g = 1 + 9.0 \sum_{i=2}^{d} (\frac{x_i}{d-1});$ $h = 1 - \sqrt{\frac{f_1}{g}} - (\frac{f_1}{g}) sin(10\pi * f_1)$	$[0, 1]$
4	$f_1 = x_1$	$f_2 = gh;$ $where, g = 1 + 9.0 \sum_{i=2}^{d} (\frac{x_i}{d-1});$ $h = 1 - \sqrt[4]{\frac{f_1}{g}} - (\frac{f_1}{g})^4$	$[0, 1]$

Table 5.2: Speedup of our GPU MOPSO (number d of dimensions = 30, number of iterations = 2500) for Test function 1.

n	CPU Time(s)	# of Nondominated Solutions	GPU Time(s)	# of Nondominated Solutions	Speedup
1024	0.83	37	0.15	42	5.5
2048	1.59	86	0.15	112	10.6
4096	4.49	143	0.16	160	28.0
8192	6.98	269	0.20	280	34.9
16384	12.18	537	0.26	566	46.8
32768	35.68	1299	0.38	1355	93.8

Table 5.3: Speedup of our GPU MOPSO (number d of dimensions = 30, number of iterations = 2500) for Test function 2.

n	CPU Time(s)	# of Nondominated Solutions	GPU Time(s)	# of Nondominated Solutions	Speedup
1024	0.92	208	0.16	268	5.7
2048	1.91	338	0.17	468	11.2
4096	3.03	838	0.18	856	16.8
8192	6.25	1436	0.21	1452	29.7
16384	12.94	2464	0.28	2563	46.2
32768	32.05	4227	0.40	4369	80.1

Table 5.4: Speedup of our GPU MOPSO (number d of dimensions = 30, number of iterations = 2500) for Test function 3.

n	CPU Time(s)	# of Nondominated Solutions	GPU Time(s)	# of Nondominated Solutions	Speedup
1024	0.74	228	0.16	279	4.6
2048	1.57	503	0.17	503	9.2
4096	3.17	834	0.18	837	17.6
8192	6.28	1309	0.22	1500	28.5
16384	12.37	2423	0.28	2450	44.1
32768	33.25	4213	0.40	4237	83.1

Table 5.5: Speedup of our GPU MOPSO (number d of dimensions = 30, number of iterations = 2500) for Test function 4.

n	CPU Time(s)	# of Nondominated Solutions	GPU Time(s)	# of Nondominated Solutions	Speedup
1024	0.92	467	0.16	492	5.7
2048	1.87	765	0.17	826	11.0
4096	3.82	1188	0.18	1328	21.2
8192	7.16	2171	0.21	2180	34.0
16384	14.51	3382	0.28	3337	51.8
32768	32.74	4701	0.40	4801	81.8

Table 5.6: Speedup of our GPU MOPSO (number n of particles = 8192, number of iterations = 2500) for Test function 1.

d	CPU Time(s)	# of Nondominated Solutions	GPU Time(s)	# of Nondominated Solutions	Speedup
32	5.98	313	0.20	230	29.9
64	12.06	286	0.27	294	46.6
128	30.77	198	0.43	196	71.5
256	69.94	207	0.74	322	94.5
512	144.90	109	1.37	292	105.7
1024	362.26	365	2.73	569	132.6

Table 5.7: Speedup of our GPU MOPSO (number n of particles = 8192, number of iterations = 2500) for Test function 2.

d	CPU Time(s)	# of Nondominated Solutions	GPU Time(s)	# of Nondominated Solutions	Speedup
32	8.29	1114	0.21	1202	39.4
64	16.63	488	0.29	1147	57.3
128	33.31	371	0.45	1141	74.0
256	63.48	335	0.77	828	82.4
512	136.66	305	1.42	315	96.2
1024	358.29	226	2.71	246	132.2

Table 5.8: Speedup of our GPU MOPSO (number n of particles = 8192, number of iterations = 2500) for Test function 3.

d	CPU Time(s)	# of Nondominated Solutions	GPU Time(s)	# of Nondominated Solutions	Speedup
32	8.57	1242	0.21	1255	40.8
64	14.22	1012	0.28	1216	50.7
128	33.72	987	0.45	1077	74.9
256	68.22	1024	0.78	1033	87.4
512	136.62	912	1.43	922	95.5
1024	351.37	828	2.73	838	128.7

Table 5.9: Speedup of our GPU MOPSO (number n of particles = 8192, number of iterations = 2500) for Test function 4.

d	CPU Time(s)	# of Nondominated Solutions	GPU Time(s)	# of Nondominated Solutions	Speedup
32	8.06	1680	0.21	1765	38.3
64	15.80	1513	0.29	1546	54.4
128	31.96	1395	0.46	1406	69.4
256	68.01	1214	0.79	1389	86.0
512	163.11	488	1.43	545	114.0
1024	368.46	128	2.72	148	135.4

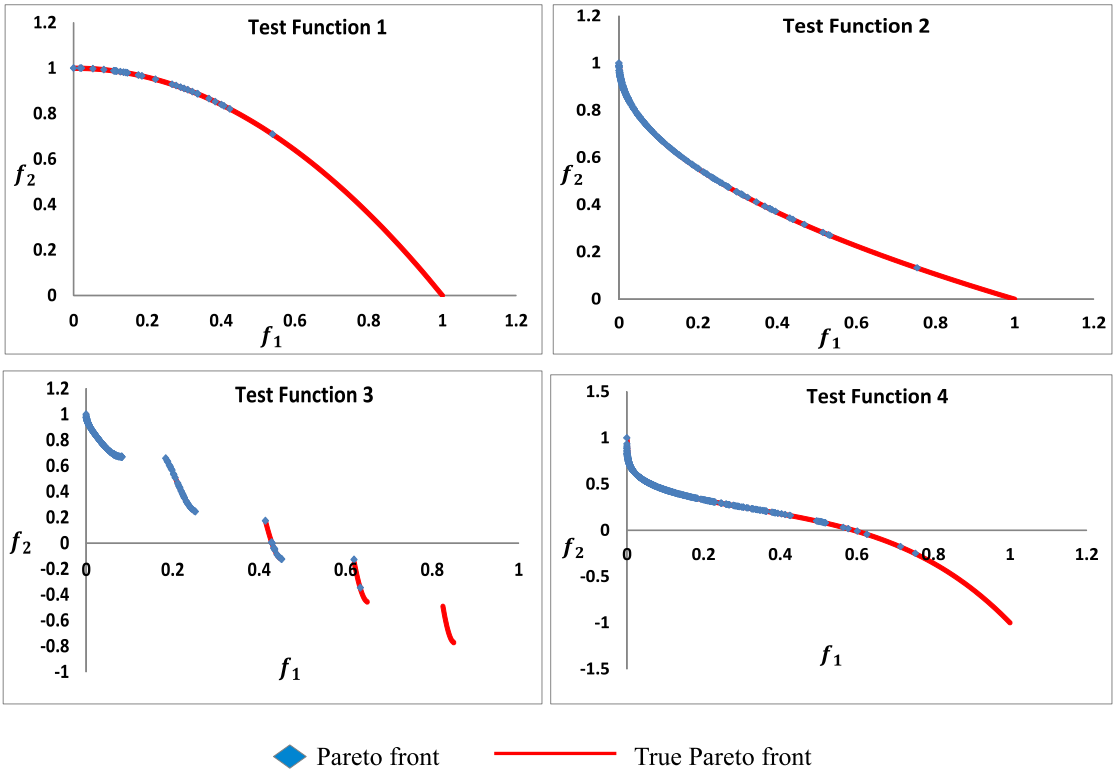

Fig. 5.9: Pareto fronts constructed on a CPU (n = 1024,d= 30, number of iterations = 2500).

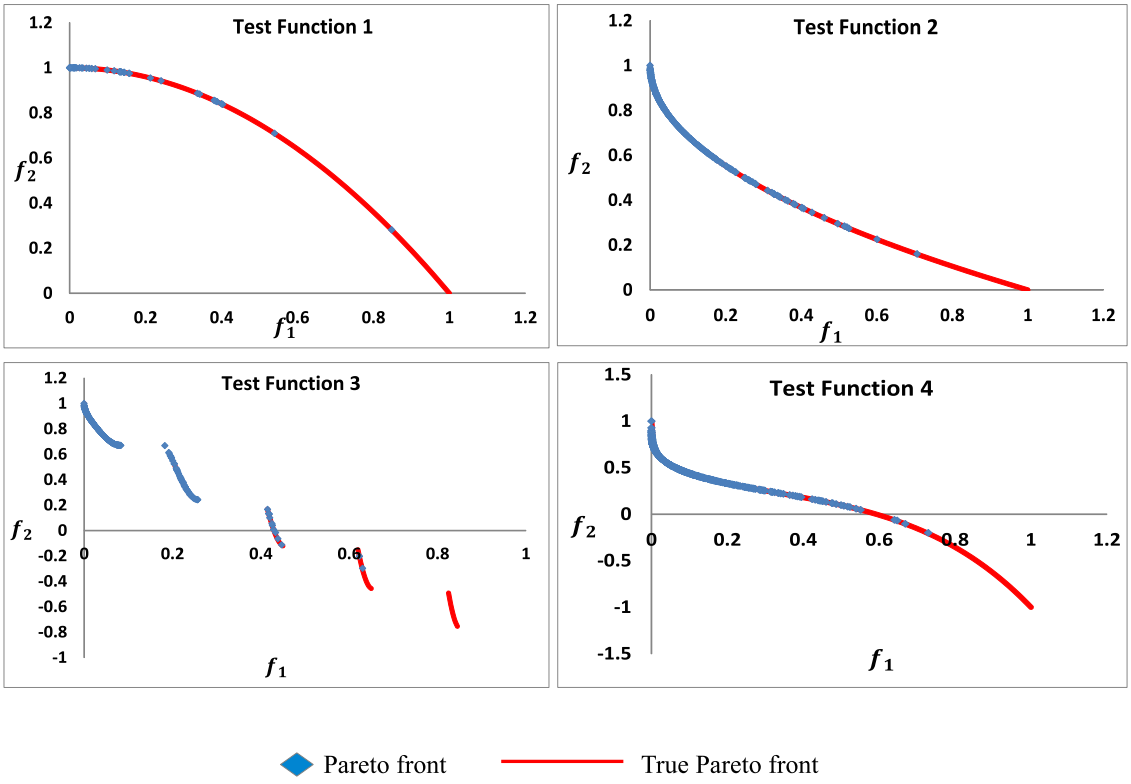

Fig. 5.10: Pareto fronts constructed on a GPU ($n = 1024$,d= 30, number of iterations = 2500).

Table 5.10: Speedup comparison between [9] and our implementation (number n of particles = 4096, number of iterations = 250) for Test function 4.

	[9]			Our implementation		
d	CPU Time(s)	GPU Time(s)	Speedup	CPU Time(s)	GPU Time(s)	Speedup
100	29.23	2.80	10.4	1.15	0.07	16.4
200	60.63	5.67	10.6	2.54	0.09	28.2

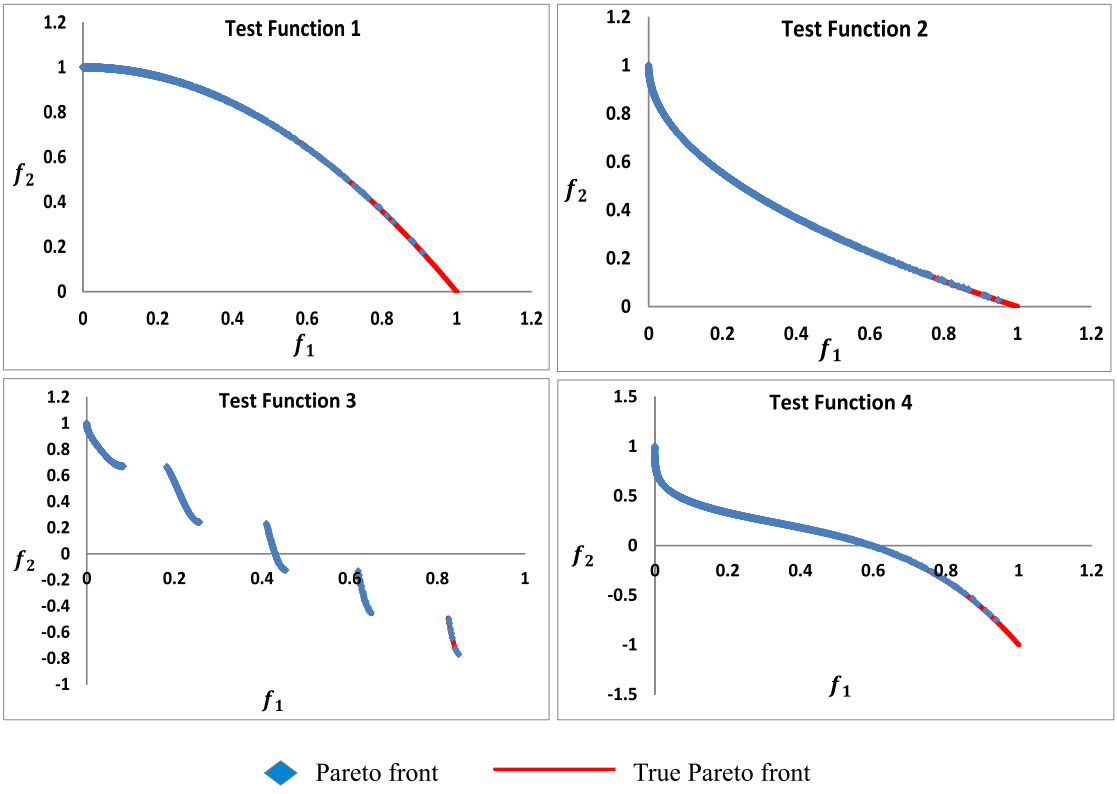

Fig. 5.11: Pareto fronts constructed on a CPU ($n = 32768$, $d = 30$, number of iterations $= 2500$).

Table 5.11: More nondominated solutions and speedup are found when swarm and dimension are simultaneously larger (number of dimensions $= 1024$, number of iteration $= 2500$, Test function 3).

n	CPU Time(s)	# of Nondominated Solutions	GPU Time(s)	# of Nondominated Solutions	Speedup
10000	340.30	946	3.08	946	110.4
20000	688.87	1283	4.88	1284	141.1
30000	1031.42	1510	6.79	1522	151.9
40000	1371.48	1713	8.73	1799	157.0

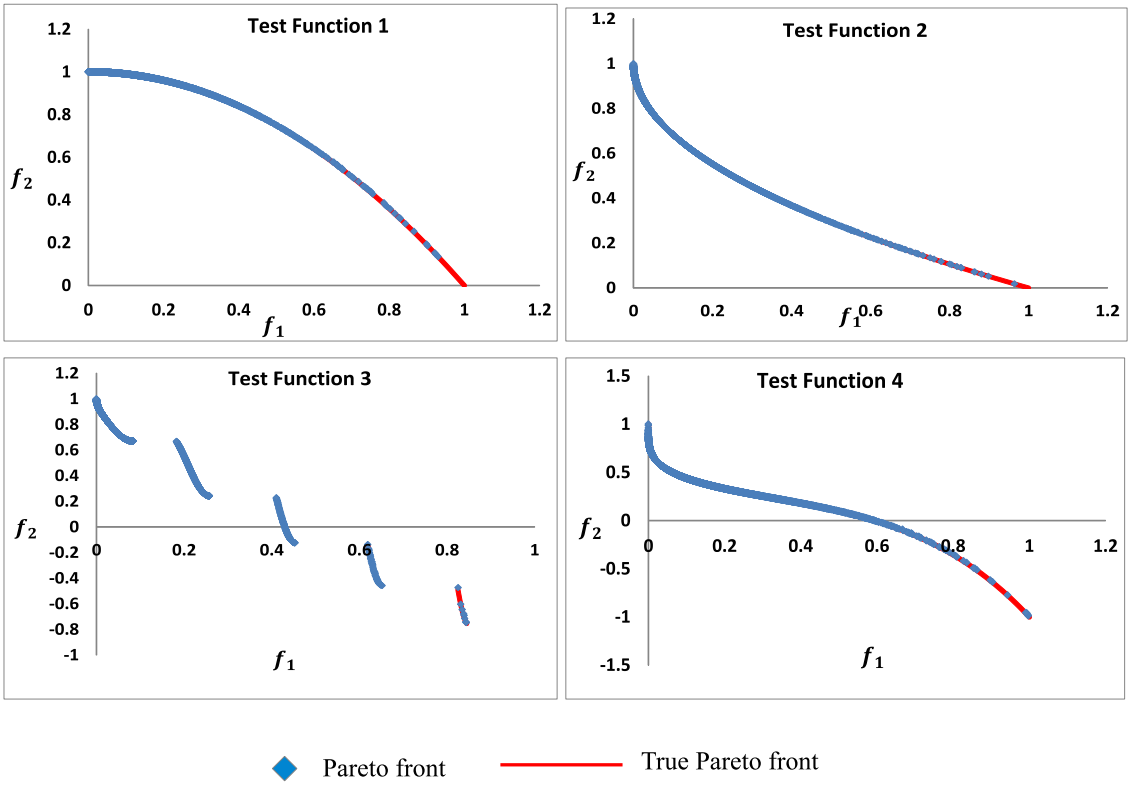

Fig. 5.12: Pareto fronts constructed on a GPU (n =32768,d = 30, number of iterations = 2500).

Table 5.12: The bottleneck on a CPU (number d of dimensions = 30, number n of particles = 2000, number of iterations = 2500) for Test function 4.

Time (%)	# of calls	Name of the function
55.61%	5000000	`float f2`
31.82%	2500	`void move_particles`
11.96%	1	`int PSO`
0.60%	2500	`void select_pBests`
0.001%	1	`void initialize`

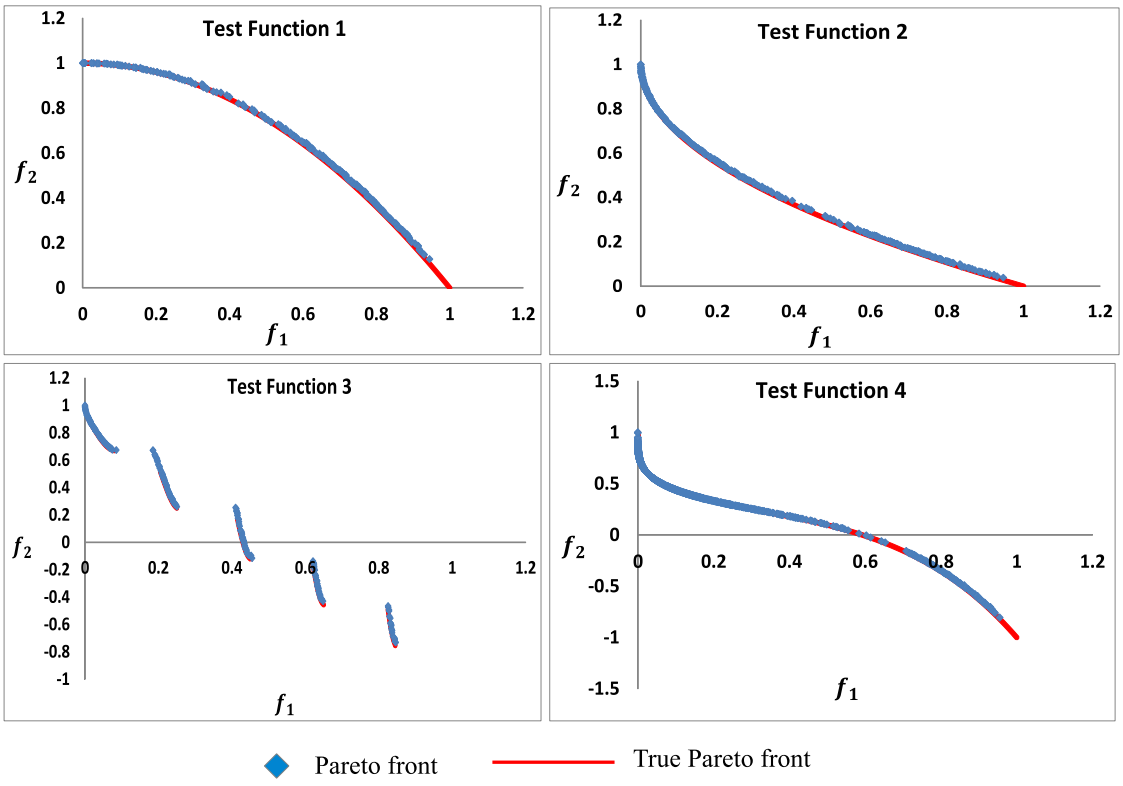

Fig. 5.13: Pareto fronts constructed on a CPU ($n = 8192$, $d = 256$, number of iterations $= 2500$).

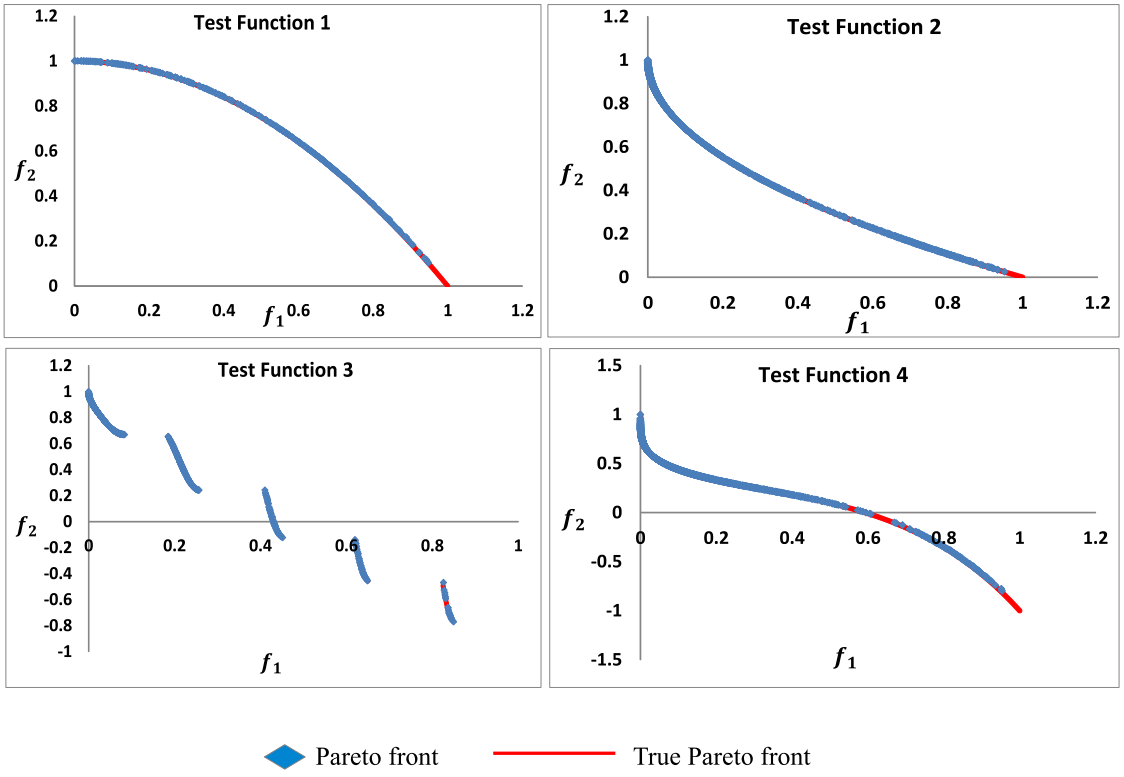

Fig. 5.14: Pareto fronts constructed on a GPU ($n = 8192$, $d = 256$, number of iterations $= 2500$).

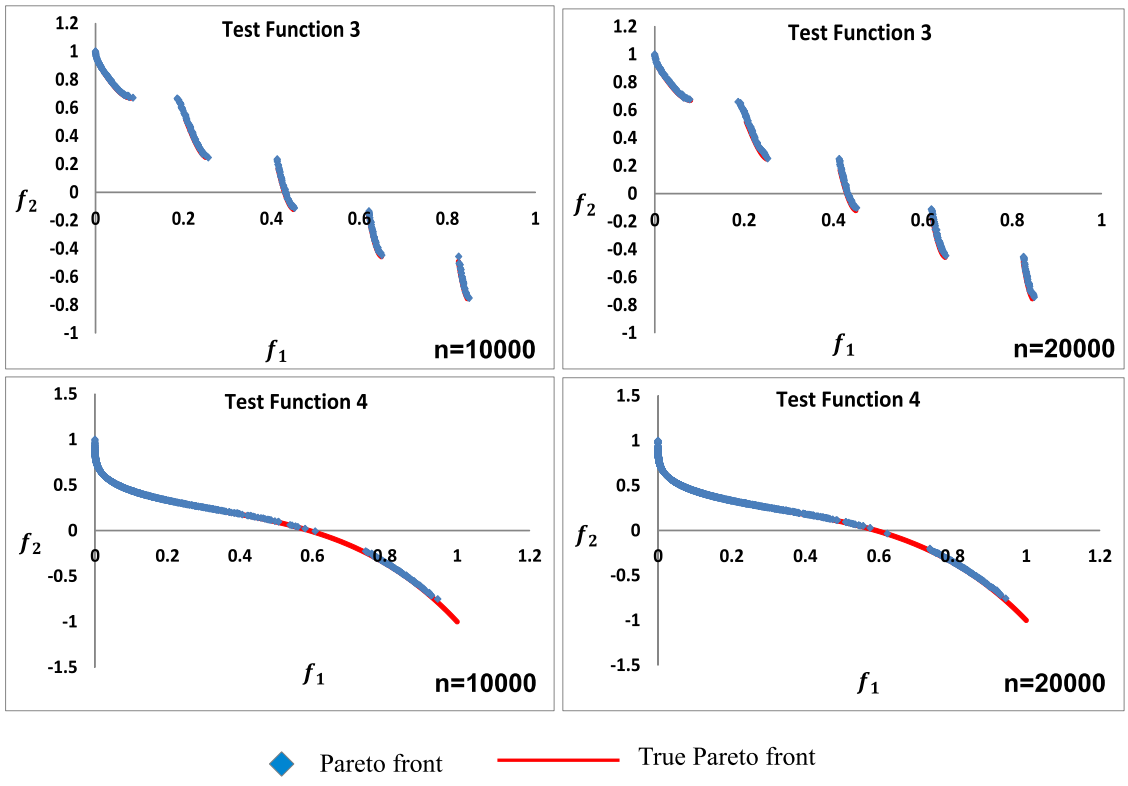

Fig. 5.15: Pareto fronts constructed on a CPU (n=10000 or 20000, d =512, number of iterations=2500) for Test function 3 and Test function 4.

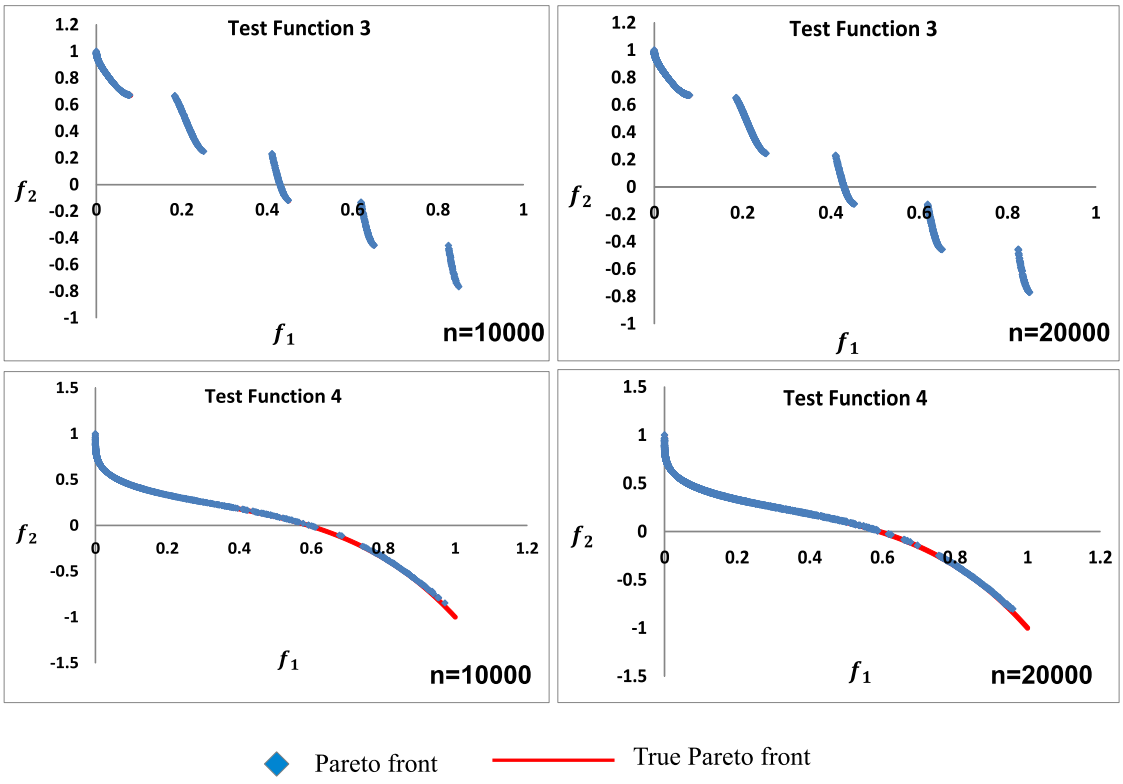

Fig. 5.16: Pareto fronts constructed on a GPU (n=10000 or 20000, d =512, number of iterations=2500) for Test function 3 and Test function 4.

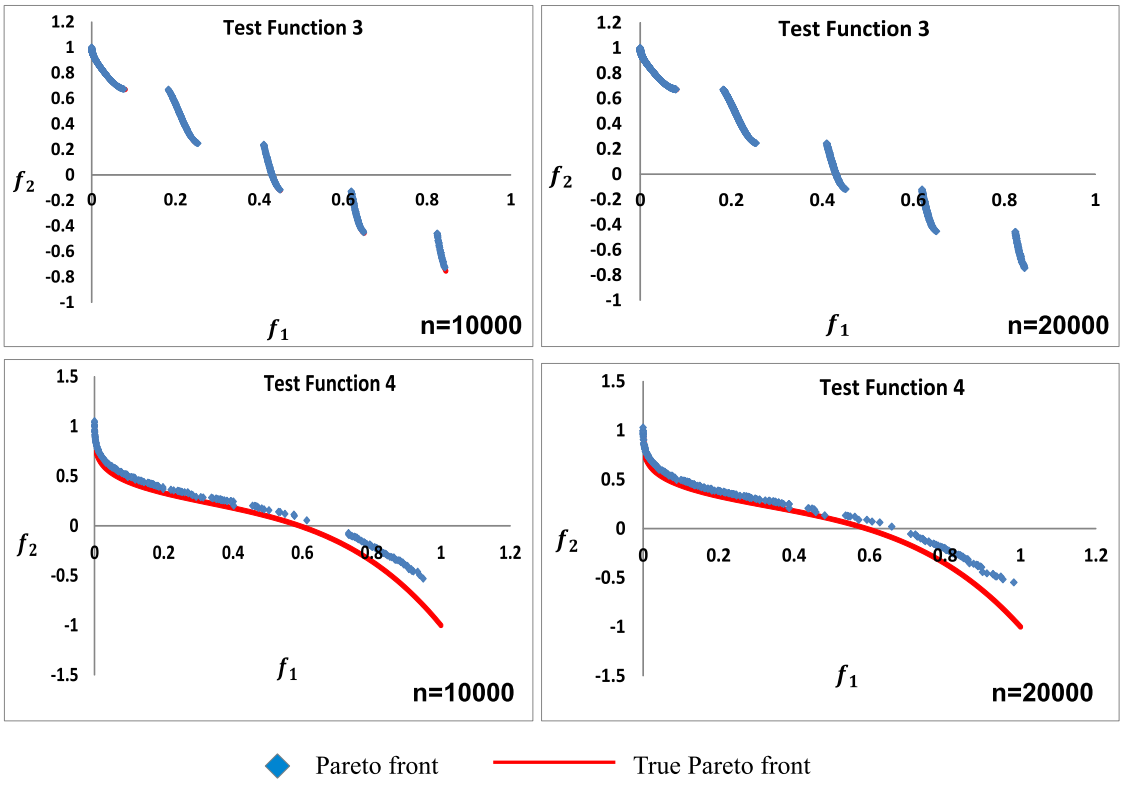

Fig. 5.17: Pareto fronts constructed on a CPU (n=10000 or 20000, d =1024, number of iterations=2500) for Test function 3 and Test function 4.

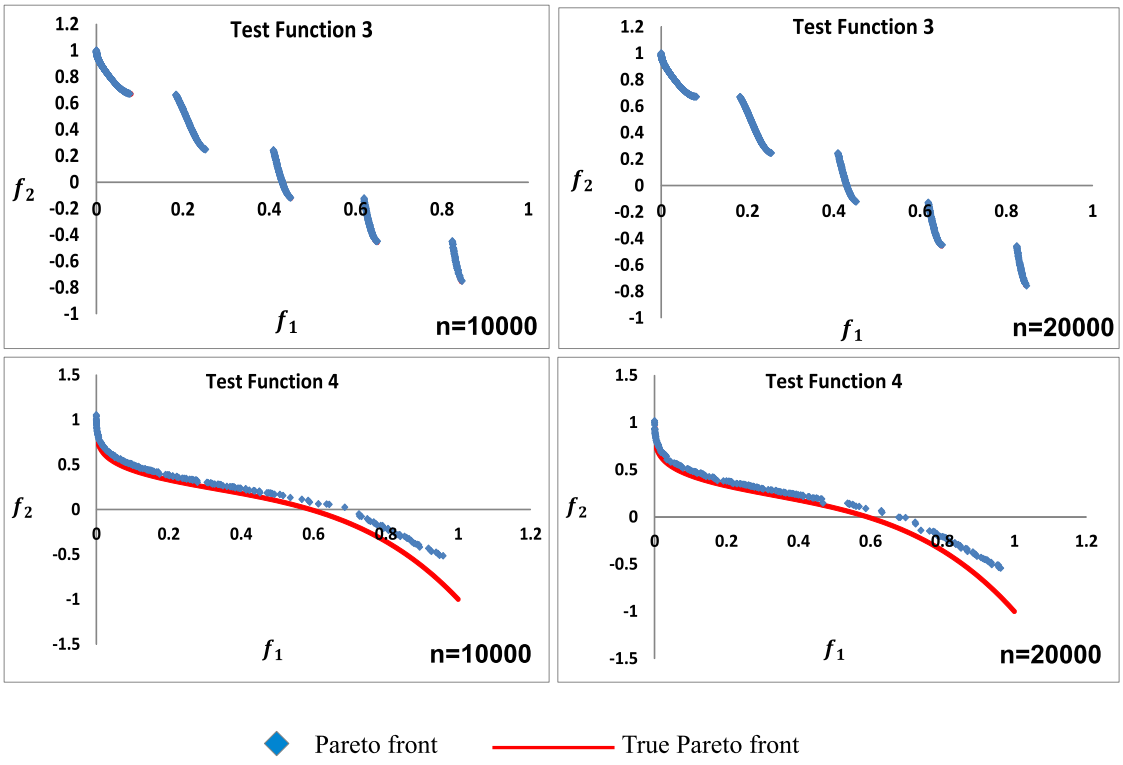

Fig. 5.18: Pareto fronts constructed on a GPU (n=10000 or 20000, d =1024, number of iterations=2500) for Test function 3 and Test function 4.

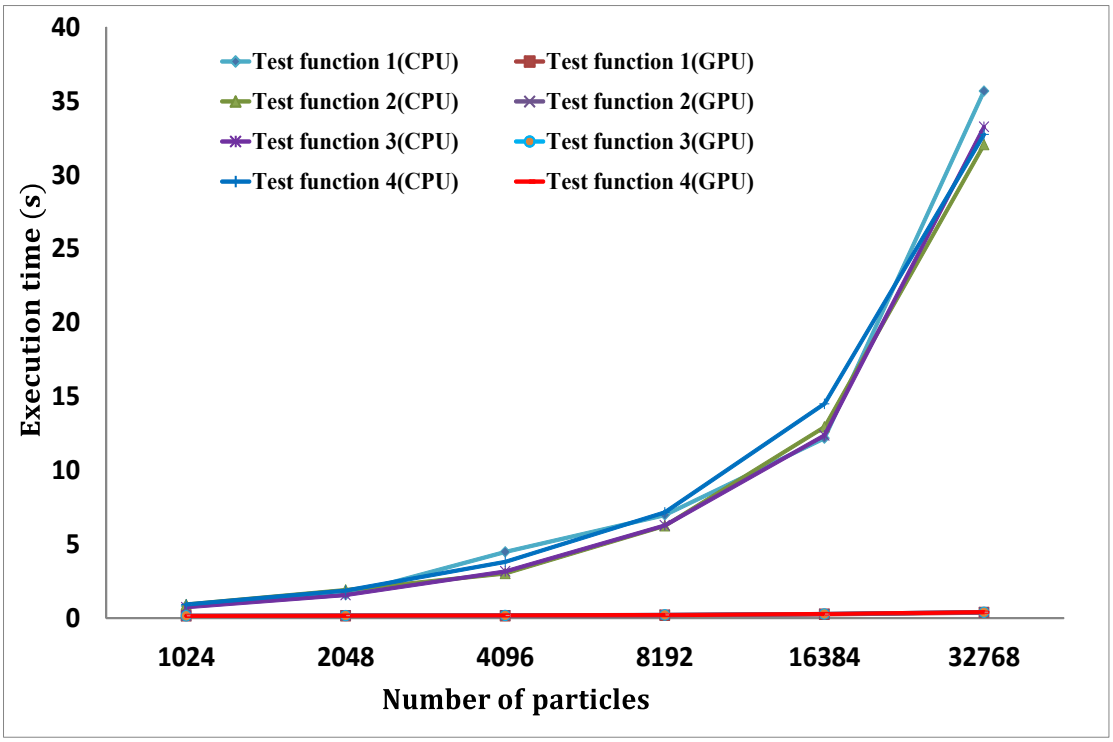

Fig. 5.19: Overlap of execution time on a GPU (number of dimensions = 30, number of iterations = 2500).

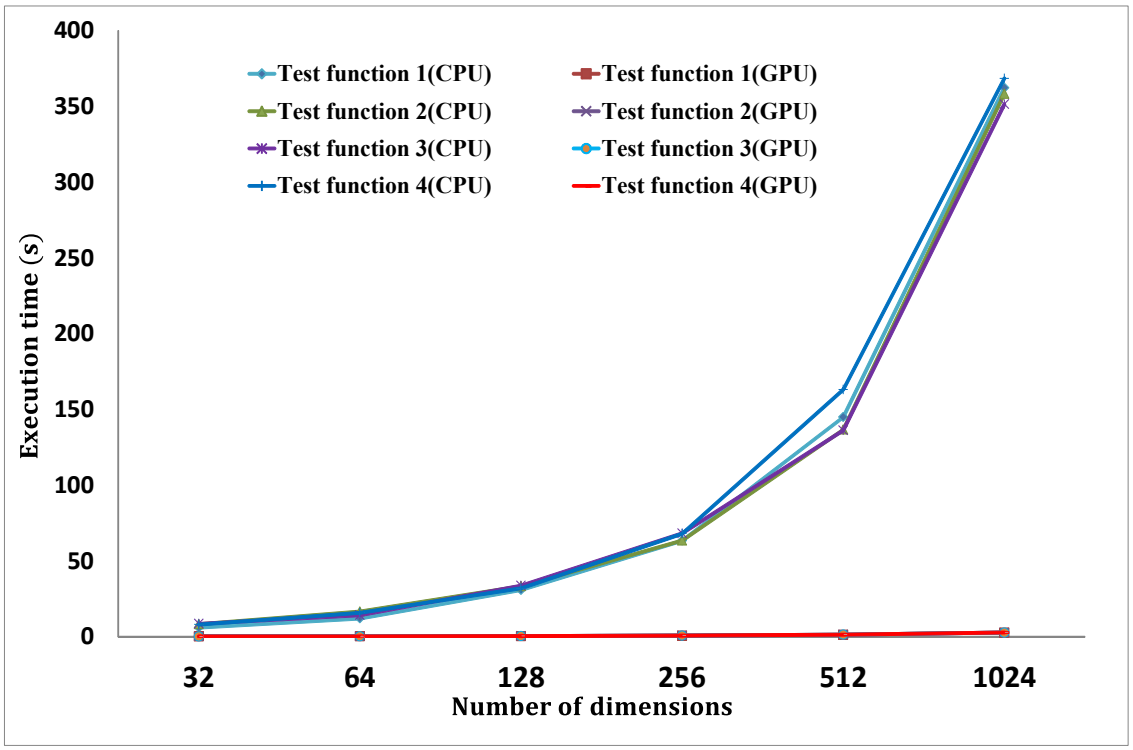

Fig. 5.20: Overlap of execution time on a GPU (number of particles = 8192, number of iterations = 2500).

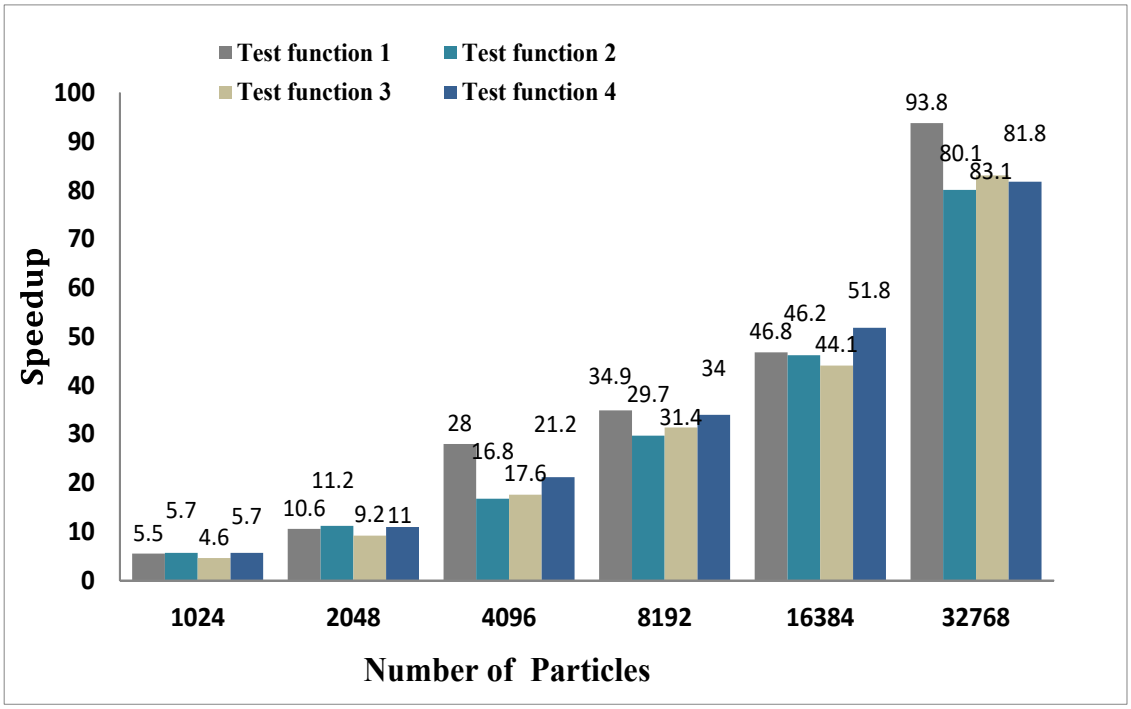

Fig. 5.21: Speedup (number of dimensions = 30, number of iterations = 2500).

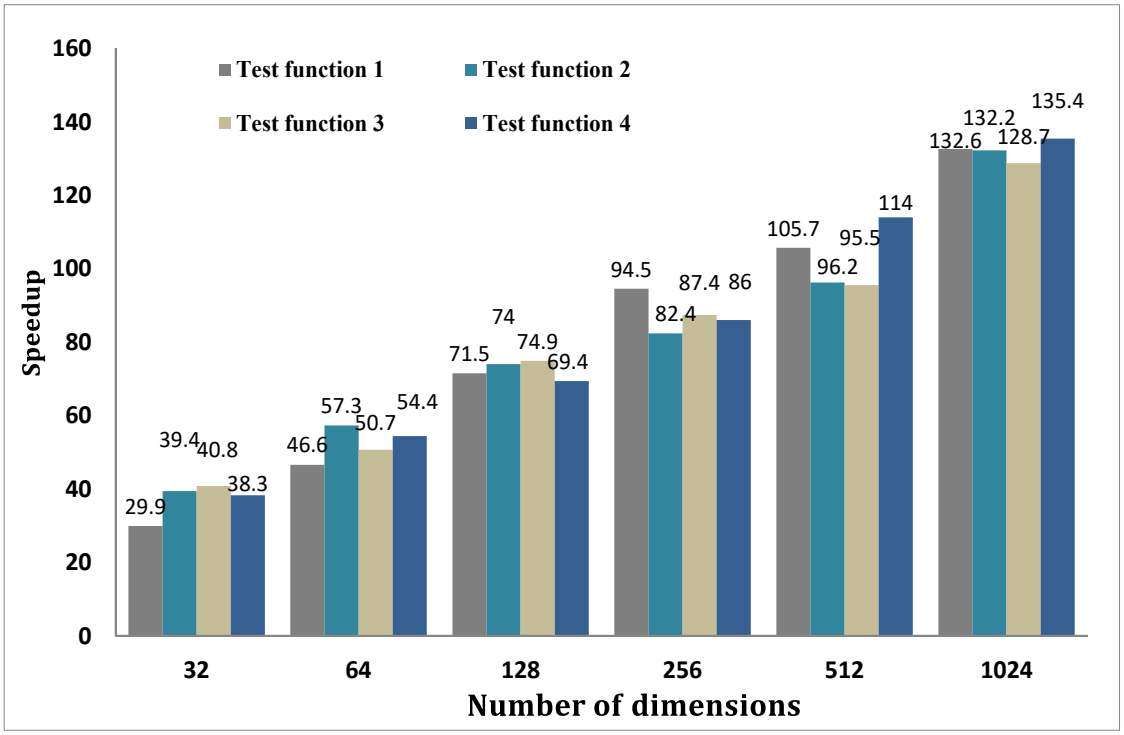

Fig. 5.22: Speedup (number of particles = 8192, number of iterations = 2500).

Table 5.13: The bottleneck on a GPU (number d of dimensions = 30, number n of particles = 2000, number of iterations = 2500) for Test function 4.

Time (%)	Time (s)	# of calls	Average (μs)	Minimum (μs)	Maximum (μs)	Name of the kernel
84.6776	2.0627	2500	0.8250	0.8181	0.9260	`void cub::DeviceScanKernel`
9.1045	0.2217	2500	0.0887	0.0479	0.1079	`__global__void evaluate_particles`
1.9281	0.0469	2500	0.0187	0.0176	0.0253	`__global__void move_particles`
1.6646	0.0405	2500	0.0162	0.0158	0.0189	`__global__void make_archive`
1.1505	0.0280	1	28.0284	28.0284	28.0284	`__global__void calculate_ParetoOptimalSet`
0.4954	0.0120	2500	0.0048	0.0045	0.0108	`__global__void select_pBests`
0.4441	0.0108	2500	0.0043	0.0042	0.0109	`void cub::DeviceScanInitKernel`
0.3166	0.0077	2500	0.0030	0.0029	0.0102	`__global__void calculate_archivesrc`
0.1266	0.0030	2500	0.0012	0.0011	0.0096	`[CUDA memset]`
0.0442	0.0010	1	1.0783	1.0783	1.0783	`void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__merge_sort::BlockSortAgent>`
0.0234	0.0005	2	0.2852	0.2493	0.3210	`void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__merge_sort::MergeAgent>`
0.0210	0.0005	2	0.2559	0.0915	0.4202	`void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__merge_sort::PartitionAgent>`
0.0009	0.0000	1	0.0221	0.0221	0.0221	`__global__void initialize`
0.0008	0.0000	1	0.0207	0.0207	0.0207	`__global__void allocate_memory`
0.0006	0.0000	1	0.0159	0.0159	0.0159	`__global__void free_memory`

5.4 Related Works

Y. Zhou and Y. Tan [9] presented a parallel approach to running MOPSO on a GPU for two objective problems based on hierarchical models. In their paper, they modified a parallel version of the Vector Evaluated Particle Swarm Optimization (VEPSO) method [53] for multi-objective problems. The optimization performance was evaluated by only one of the objectives instead of both, thus the execution time was reduced. We believe that this approach will not satisfy the constraints of MOPSO. In this approach, running speed of their GPU MOPSO was 10 times faster than that of their CPU MOPSO. In this case, no Parent front was shown.

M. L. Wong [54] implemented a parallel MOEA on a GPU based on the CUDA architecture. In his implementation, all procedures were performed on a GPU except the nondominated selection procedure, and the implementation was quite complex. The speedups of the parallel MOEA ranged from 5.62 to 10.75 times.

J. P. Arun et al.[8] presented a faster parallel MOPSO based on a master-slave model. They compared the performance of CUDA and OpenCL implementations with a sequential implementation of MOPSO through simulations. Results showed that the performance was improved by 90 percent using parallel implementations. They considered population sizes from 1000 to 2000 and low dimensions. Master archives were developed inside the CPU. Therefore, the performance decreases for large swarms and high dimensional problems.
In most of the serial and parallel implementations, they fixed the number of dimensions and a swarm at small size because the time complexity of archive handling is proportional to the sizes. If they consider a large number of iterations for best quality Pareto fronts, in the case of high dimensionality and a large swarm the computational speed will decrease significantly. We have demonstrated that our proposed implementations can handle both of cases with good Pareto fronts which are very close to the true Pareto fronts.

5.5 Summary

With the evolution of big data era and the fast development of distributed parallel computing technologies, many complicated problems can be transformed into multi-objective large scale and high dimensional problems. In this circumstance, we have developed the PSO algorithm into MOPSO for large swarms and high dimensional problems. This is the first approach to implement MOPSO for large swarms and high dimensional problems. Our GPU MOPSO and CPU MOPSO found good Pareto fronts of the four test functions which are very close to the true Pareto fronts. In our GPU MOPSO, by using the master-slave model, the optimization tasks can be allocated from the master to large amounts of slaves, which can effectively improve our implementation. We have achieved shorter execution

time. The proposed implementation is 157 times faster than our CPU MOPSO. The proposed GPU MOPSO significantly reduces execution time compared to the previous implementation. In our future work, we want to investigate the parallel implementation of the multi-objective particle swarm optimization with more than two or many objectives. We also want to improve the present MOPSO performance and Pareto fronts on a GPU.

Chapter 6

Conclusion

During my PhD, I worked in the realm of nature inspired optimization techniques using particle swarm optimization (PSO) for single and multi-objective optimization (MOPSO) problems. I studied the most efficient way to implement PSO and MOPSO using both CPU and GPU. This kind of optimization has a huge potential to solve many kinds of problems where a large number of data is analyzed to drive the best solution. The effective implementation of the PSO algorithm as a single objective optimizer is enticing to extend its uses in other areas. One of such areas is multi-objective optimization. Multi-objective problems are very common in real-world optimization fields, of which the objectives to be optimized are normally in conflict with each other. In my research, I tried to develop a new parallelized implementation of the multi-objective particle swarm optimization (GPU MOPSO) for large swarm and high dimensional optimization problems. Performance of MOPSO is dependent upon an archiving technique and leader selection. I implemented an effective and quality archiving technique. This technique starts with the first particle selection and finishes with the Pareto optimal set construction from the final archive. After generating the final archive, we randomly pick out a leader from the final archive for new velocity and positions. A master slave model is used in MOPSO Implementation where CPU handles a master and a GPU handles slaves. In my GPU MOPSO, by using master slave model, the optimization tasks can be allocated from the master to large amounts of slaves, which can effectively improve the running competency. We have achieved short execution time. The proposed implementation is 157 times faster than our CPU MOPSO.

Moving forward, I am foreseeing that my prior knowledge and hands on experiences in GPU programming and swarm intelligence can be directly applied for surgical robotics and swarm robotics-both are used for medical treatment and rescue operation. My vision of such applications, e.g. in diagnosis of cancer and similar endogenous diseases are supported by the community. Apart from swarm robotics, many more unmet clinical needs are left to be improved and I find that the health sector is one of the most challenging areas to adopt information technology

due to the complexity of our body and the concern of risking lives. Current medical system relies on a human doctor (MD) to provide the best diagnosis and treatment for the patients. However, workload, stress and other human factors can lead to an imperfect decision towards diagnosis. Some efforts have been made to replace MDs which are far from perfect. However, as Barabara Engelhardt, one of the leaders in AI and Genetics, says, "Build Methods to focus clinicians and screen patients, not MDs. Doctors are not stupid and AI is not magic", instead of replacing human, an additional intelligent system with archive of patients history can help doctors further to reach quicker and more accurate decision and thus saving both time and risk of life. Archiving patient history, handling big data in a secured manner, building an intelligent system utilizing patient history-everywhere I can see myself to be able to contribute. I enjoy outgrowing my limit, shedding old shell to grow further in more challenging field. My knack for taking new challenges with open arms and diverse experiences always helped me to think out of the box and I strongly believe, I will be a valuable member for any Innovation program.

Bibliography

[1] J. Kennedy and R. C. Eberhart, Particle Swarm Optimization, IEEE International Conference on Neural Networks 4 (1995) 1942-1948.

[2] A. P. Engelbrecht, Fundamentals of Computational Swarm Intelligence, Wiley (2005).

[3] J. Kołodziejczyk Survey on Particle Swarm Optimization accelerated on GPGPU, International Journal of Scientific Engineering and Research 5(12) (2014) 2229-5518.

[4] M. M. Hussain, H. Hattori, and N. Fujimoto, A CUDA Implementation of the Standard Particle Swarm Optimization, 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (2016) 219-226.

[5] M. M. Hussain and N. Fujimoto, Effect of the Pseudorandom Number Generators on the Standard Particle Swarm Optimization on a GPU, International Conference on Computational Science and Computational Intelligence (CSCI) (2018).

[6] J. Moore and R. Chapman, Application of Particle Swarm to Multiobjective Optimization, department of Computer Science and Software Engineering, Auburn University (1999).

[7] B. Cao, J. Zhao, Z. Lv, X. Liu, S. Yang, X. Kang, and K. Kang, Distributed Parallel Particle Swarm Optimization for Multi-Objective and Many-Objective Large-Scale Optimization, IEEE Access 5 (2017) 8214-8221.

[8] J. P. Arun, M. Mishra, and S. V. Subramaniam, Parallel Implementation of MOPSO on GPU Using OpenCL and CUDA, 18th International Conference on High Performance Computing (HiPC) (2011) 1-10.

[9] Y. Zhou and Y. Tan, GPU-Based Parallel Multiobjective Particle Swarm Optimization, International Journal of Artificial Intelligence 7, A11 (2011).

[10] X. Hu, Particle Swarm Optimization, www.swarmintelligence.org (2006).

[11] D. Bratton and J. Kennedy, Defining a Standard for Particle Swarm Optimization, IEEE Swarm Intelligence Symposium (2007) 120-127.

[12] J. C. Bansal, P. K. Singh, and M. Saraswat, Inertia Weight Strategies in Particle Swarm Optimization, Third World Congress on Nature and Biologically Inspired Computing (2011) 633-640.

[13] V. Kumar and S. Minz, Multi-Objective Particle Swarm Optimization: An Introduction, Smart Computing Review 4(5) (2014) 335-353.

[14] C. A. C. Coello, G. T. Pulido, and M. S. Lechuga, Handling Multiple Objectives With Particle Swarm Optimization, IEEE Transactions On Evolutionary Computation 8 (3) (2004) 256-279.

[15] K. Deb, Multi-Objective Optimization Using Evolutionary Algorithms, Wiley (2009).

[16] E. Zitzler, K. Deb and L.Thiele, Comparison of Multiobjective Evolutionary Algorithms: Empirical Results, Evolutionary Computation 8 (2) (2000) 173-195.

[17] NVIDIA, CUDA Toolkit Documentation 10.1.168, http://docs.nvidia.com/cuda (2019).

[18] K. E. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha, Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware, 26th Annual Conference on Computer Graphics and Interactive Techniques (1999) 277-286.

[19] Z. W. Luo, H. Liu, and X. Wu, Artificial Neural Network Computation on Graphic Process Unit, IEEE International Joint Conference on Neural Networks 1 (2005) 622-626.

[20] W. Liu and B. Vinter, An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data, 28th IEEE International on Parallel and Distributed Processing Symposium (2014) 370-381.

[21] G. Dafeng and W. Xiaojun, Real-time Visual Hull Computation Based on GPU, IEEE International Conference on Robotics and Biomimetics (ROBIO) (2015) 1792-1797.

[22] A. P. Yazdanpanah, A. K. Mandava, E. E. Regentova, V. Muthukumar, and G. Bebis, A CUDA Based Implementation of Locally-and Feature-Adaptive Diffusion Based Image Denoising Algorithm, 11th International Conference on Information Technology: New Generations (ITNG) (2014) 388-393.

[23] NVIDIA, CUDA C Best Practices Guide 10.1.168, https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html (2019).

[24] L. Howes and D. Thomas, Efficient Random Number Generation and Application Using CUDA, GPU Gems 3, Chapter 37, http://developer.nvidia.com/gpugems/GPUGems3/gpugems3_pref01.html (2016).

[25] P. L'ecuyer, Tables of Maximally Equidistributed Combined LFSR Generators, Mathematics of Computation 68 (225) (1999) 261-269.

[26] D. E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms (2nd Edition) (1969).

[27] G. Marsaglia, Xorshift RNGs, Journal of Statistical Software 8 (2003) 1-6.

[28] F. Panneton and P. L'Ecuye, Particle Swarm Optimization, IEEE International Conference on Neural Networks 4 (1995) 1942-1948.

[29] J. Kaur, S. Singh and S. Singh, Parallel Implementation of PSO Algorithm Using GPGPU, Second International Conference on Computational Intelligence and Communication Technology (CICT) (2016).

[30] NVIDIA, cuRAND, https://docs.nvidia.com/cuda (2019).

[31] NVIDIA, Thrust, http://docs.nvidia.com/cuda/thrust/index .html (2019).

[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, The MIT Press (2009).

[33] NVIDIA, CUB Documentation ,https://nvlabs.github.io /cub/index.html (2013).

[34] NVIDIA, CUDA Programming Guide 10.1.168, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (2019).

[35] D. A. VanVeldhuizenr, J. B. Zydallis and G. B. Lamont, Considerations in Engineering Parallel Multiobjective Evolutionary Algorithms, IEEE Transactions on Evolutionary Computation 7 (2) (2003) 144-173.

[36] Y. Zhou and Y. Tan, GPU-based Parallel Particle Swarm Optimization, 11th IEEE Congress on Evolutionary Computation (2009) 1493-1500.

[37] R. M. Calazan, N. Nedjah, and L. D. M. Mourelle, Parallel GPU-based Implementation of High Dimension Particle Swarm Optimizations, IEEE Fourth Latin American Symposium on Circuits and Systems (LASCAS) (2013) 1-4.

[38] V. K. Reddy and L. S. S. Reddy, Performance Evaluation of Particle Swarm Optimization Algorithms on GPU Using CUDA, International Journal of Computer Science and Information Technologies 5(1) (2012) 65-81.

[39] L. Mussia, F. Daoliob, and S. Cagnoni, Evaluation of Parallel Particle Swarm Optimization Algorithms within the CUDA™ Architecture, Information Sciences on Interpretable Fuzzy Systems 181 (2011) 4642-4657.

[40] W. Li and Z. Zhang, A CUDA-based Multichannel Particle Swarm Algorithm, International Conference on Control, Automation and Systems Engineering (CASE) (2011) 1-4.

[41] R. M. Calazan, N. Nedjah, and L. D. M. Mourelle, A Cooperative Parallel Particle Swarm Optimization for High-Dimension Problems on GPUs, BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence (2013) 356-361.

[42] H. Zhu, Y. Guo, J. Wu, and J. Gu, Paralleling Euclidean Particle Swarm Optimization in CUDA, 4th International Conference on Intelligent Networks and Intelligent Systems (ICINIS) (2011) 93-96.

[43] E. H. M. Silva and C. J. A. B. Filho, PSO Efficient Implementation on GPUs Using Low Latency Memory, IEEE Latin America Transactions 13(5) (2015) 1619-1624.

[44] O. Bali, W. Elloumi, P. Krömer, and A. M. Alimi, GPU Particle Swarm Optimization Applied to Travelling Salesman Problem, IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC) (2015) 112-119.

[45] X. Yang, Test Problems in Optimization, Cornell University Library (2010).

[46] NVIDIA, GeForce GTX 980 for Desktop, http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980.

[47] K. Masuda and K. Kurihara, A Constrained Global Optimization Method Based on Multi-Objective Particle Swarm Optimization, Electronics and Communications in Japan 95 (1) (2012) 43-54.

[48] K. E. Parsopoulos and M. N. Vrahatis, Recent Approaches to Global Optimization Problems through Particle Swarm Optimization, Natural Computing (1) (2002) 235-306.

[49] K. E. Parsopoulos and M. N. Vrahatis, Particle Swarm Optimization Method in Multiobjective Problems, ACM Symposium on Applied Computing (2002) 603-607.

[50] NVIDIA, NVIDIA TITAN V, https://www.nvidia.com/en-us/titan/titan-v.

[51] Intel, Intel Xeon E3-1220 v5, https://ark.intel.com/content /www/us/en/ark/products/88172/ intel-xeon-processor-e3-1220-v5-8m-cache-3-00-ghz.html.

[52] NVIDIA, GeForce 9800 GT, https://www.geforce.com/hardware /desktop-gpus/geforce-9800gt.

[53] K. E. Parsopoulos, D.K. Tasoulis, and M. N. Vrahatis, Multiobjective Optimization Using Parallel Vector Evaluated Particle Swarm Optimization, IASTED International Conference on Artificial Intelligence and Applications 2 (2004) 823-828.

[54] M. L. Wong , Parallel Multi-objective Evolutionary Algorithms on Graphics Processing Units, 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference (2009) 2515-2522.

Publications of the Author

Refereed Journal Paper

1. Md. Maruf Hussain and Noriyuki Fujimoto : GPU based Parallel Multi-objective Particle Swarm Optimization for Large Swarm and High Dimensional Problems, Parallel Computing, Vol.92, 19 pages, Elsevier, Impact Factor 1.281, accepted, https://doi.org/10.1016/j.parco.2019.102589 (2020)

Refereed International Conference Papers

1. Md. Maruf Hussain and Noriyuki Fujimoto : Effect of the Pseudorandom Number Generators on the Standard Particle Swarm Optimization on a GPU, Proc. of the 2018 International Conference on Computational Science and Computational Intelligence (CSCI), pp.1–6, IEEE Computer Society Conference Publishing Services, to appear (2020)

2. Md. Maruf Hussain and Noriyuki Fujimoto : Parallel Multi-objective Particle Swarm Optimization for Large Swarm and High Dimensional Problems, Proc. of the 2018 IEEE Congress on Evolutionary Computation (CEC), pp.1–10, DOI: 10.1109/CEC.2018.8477848 (2018)

3. Md. Maruf Hussain, Hiroshi Hattori, and Noriyuki Fujimoto : A CUDA Implementation of the Standard Particle Swarm Optimization, Proc. of 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp.219–226, IEEE Computer Society Conference Publishing Services, DOI: 10.1109/SYNASC.2016.043 (2016)