A GPU-based Parallel Fireworks Algorithm for Optimization

Ke Ding
Key Laboratory of Machine Perception (MOE), Peking University
Department of Machine Intelligence, School of Electronics Engineering and Computer Science, Peking University
Beijing, China
keding@pku.edu.cn

Shaoqiu Zheng
Key Laboratory of Machine Perception (MOE), Peking University
Department of Machine Intelligence, School of Electronics Engineering and Computer Science, Peking University
Beijing, China
zhengshaoqiu@pku.edu.cn

Ying Tan
Key Laboratory of Machine Perception (MOE), Peking University
Department of Machine Intelligence, School of Electronics Engineering and Computer Science, Peking University
Beijing, China
ytan@pku.edu.cn

ABSTRACT

Swarm intelligence algorithms have been widely used to solve difficult real world problems in both academic and engineering domains. Thanks to the inherent parallelism, various parallelized swarm intelligence algorithms have been proposed to speed up the optimization process, especially on the massively parallel processing architecture GPUs. However, conventional swarm intelligence algorithms are usually not designed specifically for the GPU architecture. They either can not fully exploit the tremendous computational power of GPUs or can not extend effectively as the problem scales go large. To address this shortcoming, a novel GPU-based Fireworks Algorithm (GPU-FWA) is proposed in this paper. In order to fully leverage GPUs' high performance, GPU-FWA modified the original FWA so that it is more suitable for the GPU architecture. An implementation of GPU-FWA on the CUDA platform is presented and tested on a suite of well-known benchmark optimization problems. We extensively evaluated and compared GPU-FWA with FWA and PSO, in respect with both running time and solution quality, on a state-of-the-art commodity Fermi GPU. Experimental results demonstrate that GPU-FWA generally outperforms both FWA and PSO, and enjoys a significant speedup as high as 200x, compared to the sequential version of FWA and PSO running on an up-to-date CPU. GPU-FWA also enjoys the advantage of being easy to implement and scalable.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

GECCO'13, July 6-10, 2013, Amsterdam, The Netherlands.

General Terms

Algorithms, Performance, Experimentation

Keywords

Swarm Intelligence, Fireworks Algorithm, Parallel Computing, GPU computing, CUDA

1. INTRODUCTION

Optimization problems are ubiquitous throughout the scientific community and arise in a variety of engineering applications. For many optimization problems, the objective is complicatedly non-linear and non-differentiable, often with no explicit expression. To deal with these problems, various nature-inspired methods are proposed, one of which is the family of swarm intelligence algorithms.

Swarm intelligence is a cluster of population-based metaheuristic stochastic algorithms. It is based on the study of collective behavior in decentralized, self-organized systems without external guidance or central coordination [4]. A swarm is typically made up of a population of simple agents. The collective system is capable of performing complex tasks in a dynamic environment without external guidance and central coordination.

In the swarm algorithm, a swarm constitutes a number of individuals. Each individual owns a position in the search space and can evaluate the goodness or fitness of the current position. Individuals can exchange knowledge about the search space with one another through specific mechanism.

Due to its simplicity and effectiveness, there has been a growing interest in applying swarm intelligence algorithms to engineering optimization problems. Many swarm intelligence algorithms have been proposed in the last several decades. New algorithms are also continually proposed. Those swarm algorithms make use of different heuristic mechanisms inspired by collective behavior and succuss in diverse problems.

With its low price and easy access, the GPU has gained much popularity in general purpose computing [12]. Accelerating swarm intelligence algorithms for solving complex real problems on GPU platform has attracted the attention of many researchers due to their applicability to many engineering and scientific problems. With improvements in its programmability and the emergence of more handy develop-

ment toolkits, more and more swarm intelligence algorithms are implanted to the GPU hardware to leverage the rapidly increasing performance of GPU. Various GPU-based implementations from low-level rendering languages [16] to lately high-level general languages [17, 3, 6, 14] have been reported.

Although GPUs have been successful in accelerating swarm intelligence algorithms, conventional swarm intelligence algorithms are usually designed to be used under small swarm size. In the meantime due to the very different architectures, the same operation may have very different efficiency on CPU and GPU. All these factors prevent the fully exploitation of GPU's computing power.

To tackle this, a new GPU-based swarm algorithm called GPU-FWA is proposed in this paper.

Fireworks Algorithm (FWA) [15] is a novel swarm intelligence algorithm for optimization. FWA stimulates the phenomenon of firework explosion. It is reported that FWA shows better performance than standard PSO and Clonal PSO [15]. GPU-FWA is based on the FWA, and can extend along with the problem scale in a natural and easy way. GPU-FWA modifies the original FWA to suit the particular GPU architecture. It do not need special complicated restructure data, thus making it easy to implement, while, in the meantime, can fully exploit the great computing power of GPU. A mutation mechanism called attract-repulse mutation is introduced to guide the search process.

We implemented the proposed algorithm within the CUDA platform. The proposed algorithm is extensively evaluated and compared on both speedup and solution quality on a state-of-the-art Fermi GPU architecture. Experiments show that it outperforms standard PSO and FWA, and obtains a significant speedup up to 140 and 200 compared to CPU-based FWA and PSO respectively on an up-to-date CPU.

The remainder of this paper is organized as follows. Section 2 briefly reviews the previous work on FWA as well as general purpose GPU computing and its application in swarm intelligence, especially the NVIDIA's Computing Unified Device Architecture (CUDA). We present the proposed algorithm GPU-FWA and its implementation within CUDA platform in section 3. The experiments and results goes in Section 4. Finally, we conclude this paper in the last section.

2. RELATED WORK

2.1 Fireworks Algorithm (FWA)

A thorough and detailed review on swarm intelligence is beyond the scope of this paper. In this section, we brief reviewed FWA base on which the GPU-FWA is proposed. For a comprehensive survey on swarm intelligence, readers can refer to [4] and [1].

Inspired by the explosion process of fireworks, a novel Fireworks Algorithms (FWA) for optimization was proposed by Tan and Zhu [15]. It has been successfully applied in Nonnegative matrix factorization[5]. The framework of FWA is as depicted by Fig. 1.

In FWA, the optimization process is driven by the "explosions" of a swarm of artificial fireworks. The amplitude of a certain explosion is calculated as follows:

$$A_i = \hat{A} \cdot \left(\frac{f(\mathbf{x}_i) - y_{min} + \xi}{\sum_{i=1}^{n}(f(\mathbf{x}_i) - y_{min} + \xi)} + \delta \right), \qquad (1)$$

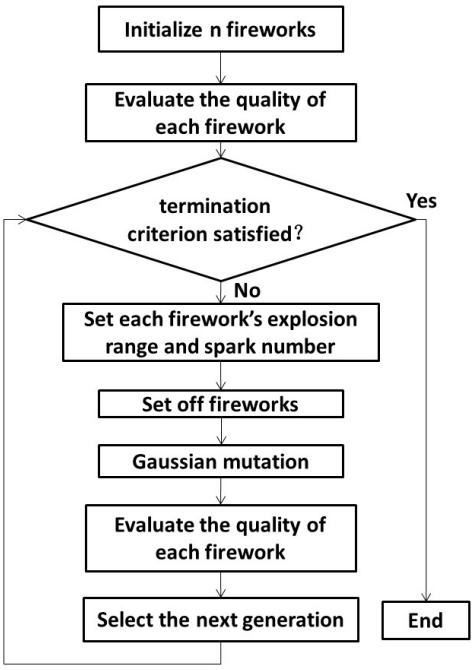

Figure 1: Framework of fireworks algorithm

where the predefined $\hat{A}$ denotes the maximum explosion amplitude, and $y_{min} = min(f(\mathbf{x}_i))\,(i = 1, 2, \ldots, n)$ is the minimum (best) value of the objective function among the n fireworks, and ξ , which denotes the machine precision, is utilized to avoid zero division error, and δ is a small float constant to avoid A_i too small for the ith firework.

The number of sparks generated by each fireworks x_i is defined as follows:

$$s_i = m \cdot \frac{y_{max} - f(\mathbf{x}_i) + \xi}{\sum_{i=1}^{n}(y_{max} - f(\mathbf{x}_i)) + \xi}, \qquad (2)$$

where m is a parameter controlling the total number of sparks generated by the n fireworks, $y_{max} = max(f(\mathrm{x}_i))\,(i = 1, 2, \ldots, n)$ is the maximum (worst) value of the objective function among the n fireworks, and ξ denotes the machine precision. s_i is rounded to the nearest integer.

The philosophy lying behind Eq. 1 and Eq. 2 is that fireworks with better fitness generate more sparks within smaller range (observe Fig. 2). Via this mechanism, more computing resource can be assigned to better space to enhance exploitation, and for the worse space, the search trends to exploration.

To increase the diversity of the swarm, a gaussian mutation is introduced to generate sparks. This mechanism is critical for the GPU-FWA, we will discuss it in detail in section 3.

2.2 General Purpose GPU Computing

Driven by the insatiable demand for realtime, high-definition 3D graphics, the GPU has evolved into a massively parallel, many-core processor in the last few years. With its tremendous computational horsepower, the GPU has become a significant part of modern mainstream, general-purpose computing systems [11]. The GPU is especially well-suited to address problems with high arithmetic intensity (the ratio

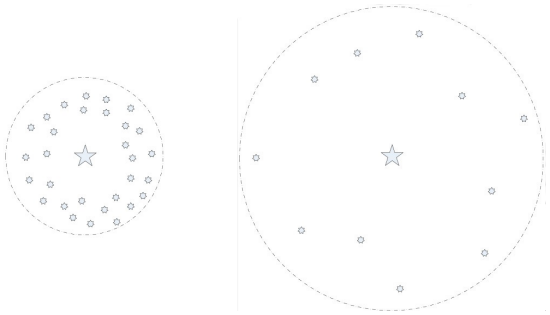

Figure 2: Two Types of Firework Explosions:Good Explosion (left) and Bad Explosion (right)

of arithmetic operations to memory operations) where the same program is executed on many data elements in parallel.

NVIDIA's Computing Unified Device Architecture (CUDA) is a high level general purpose parallel computing platform and programming model. CUDA comes with a software environment that allows developers to use C as a high-level programming language, thus, makes it easier for programmers to fully exploit the parallel feature of GPUs without an explicit familiarity with the GPU architecture.

In CUDA programming, GPU computing is conducted by kernels. A kernel is a function that explicitly specifies data parallel computations to be executed on GPUs. When a kernel is launched on the GPU, it is executed by a batch of threads. Threads are organized into independent blocks, and several blocks, in turn, constitute a grid.

Threads run in a unique mechanism called Single Instruction, Multiple Thread (SIMT). When the GPU is given one or more thread blocks to execute, it partitions them into warps. As a warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths. As can be seen in section 3, the concept of warp plays an import role in designing and implementing GPU-FWA.

Closely related to CUDA's tread hierarchy is its memory model. CUDA threads may access data from multiple memory spaces during their execution as illustrated by Fig. 3. Each thread has private registers and local memory. Each thread block has shared memory visible to all threads of the block. All threads have access to the same global memory. Register and shared memory are very fast on-chip memory while global memory is off-chip and has very long access latency. So shared memory should be used whenever possible.

3. GPU-FWA: ALGORITHM AND IMPLEMENTATION

GPUs provide great computing power at an affordable cost, but it is not clear whether a conventional paradigm is suitable for expressing parallelism in a way that is efficiently implementable on GPU architectures. In this section, we present a GPU-based approach which is dedicated to the GPU massively parallel architecture.

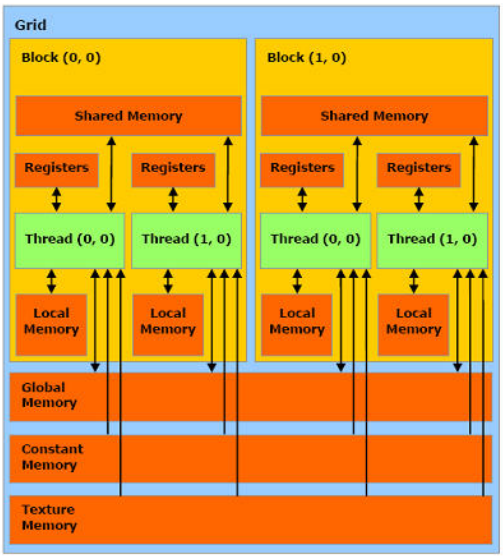

Figure 3: Memory Model of CUDA

Algorithm 1 GPU-FWA

```
Initialize n fireworks
calculate the fitness value of each fireworks
calculate A_i according to Eq.1
while termination condition unsatisfied do
    for i = 1 to n do
        Search according to Al. 2
    end for
    Mutate according to Al. 3
    calculate the fitness values of the new fireworks
    update A_i according to Eq.1
end while
```

The goals of the proposed algorithm on the CUDA platform are achieving:

- Good quality of solutions. The algorithm can find good solutions, compared to the state-of-the-art algorithms.
- Good scalability. As the problem gets complex, the algorithm can scale in a natural and decent way.
- And ease of implementation and usability, i.e. few control variables to steer the optimization. These variables should also be robust and easy to choose.

To meet these goals, several critical modifications to the original FWA are adopted to take benefit of this particular architecture. The pseudo-code of the proposed algorithm is depicted by Al. 1.

As other swarm intelligence algorithms, GPU-FWA is an iterative algorithm. In each iteration, every firework does a local search independently. Then, an information-exchange mechanism is triggered to utilize the heuristic information to guide the search process. The mechanism should make a balance between exploration and exploitation.

Algorithm 2 FWA Search

1: **for** $i = 1$ to L **do**
2: generate m sparks according to Al. 2
3: evaluate the fitnesses of each sparks
4: find the best spark with best fitness value, replace it with the current firework if better.
5: **end for**

As the algorithm is self-descriptive, what's left to be made clear is Al. 2 and Al. 3. In the following subsections, we explain these two algorithms in detail, respectively.

3.1 FWA Search

Mimicking the explosion procedure of a firework in sky, FWA generate certain number of sparks to exploit the neighbor solution space. Fireworks with better fitness values generate more sparks with a smaller amplitude. This strategy aims to put more computational resources to the more potential position, thus making a balance between exploration and exploitation.

In Al. 2, we adopt this strategy, but in a 'greedy' way, i.e., instead of a global selection procedure in FWA, each firework is updated by its current best sparks. The mechanism exhibits an enhanced hill-climbing behavior search.

Each firework generates a fixed number of sparks. The exact number (m) of sparks is determined in accordance with the specific GPU hardware architecture. This fixed encoding of firework explosion is more suitable for parallel implementation on the GPUs.

As aforementioned in section 2.2, within CUDA-enabled GPU, threads are scheduled by warp. Nowadays, the warp size is 32 for all the CUDA-enable GPUs. Each warp is assigned certain number of Stream Processors (SPs). All threads in the same warp execute a common instruction at a time on these SPs. For the older generation Tesla architecture [8], the number is 8, and for Fermi architecture [9] is 16.

As with our experimental setting (GeForce 560Ti, see Sec. 4.1), the warp size is 32, and is assigned to 16 SPs. To avoid waste of hardware resource, m should be 16 or multiple of 16. But, it is unnecessary to pick $m > 16$, as greater m is apt to over-exploit a certain position, while a better refined search can be achieved via running more explosions.

So as a rule of thumb, m should be 16 and 32 on GPUs of the Fermi architecture, and 8 or 16 on previous generation Tesla architecture. Thus the sparks of each firework can be generated by treads in a single warp, which, as mentioned a section 2.2, need not any extra synchronization overhead.

Also, as can seen from Al. 2, unlike FWA, in GPU-FWA fireworks don't exchange information in each explosion procedure, and the number of sparks for each firework generation is fixed.

It takes the following advantages.

Firstly, global communications among fireworks need explicit synchronization, which implies a considerable overhead. By letting the algorithm perform a given number of iterations without exchanging information, the time can be reduced greatly.

Secondly, the number of each firework to generate is dynamically determined, the computation task must be assigned dynamically through the optimization procedure. As GPUs are inefficient at control operations ,the dynamic computation assignment is apt to harm the overall performance of GPUs. By fixing the sparks number, we can assign each firework to a warp, this way, all sparks are synchronized implicitly without extra overhead.

The last but not the least, implemented the explosion in one block of threads, it can fully utilize the shared memory, thus, once the firework position and fitness is loaded from the global memory, no visit to the global memory is needed. The latency of visiting global memory can be reduced greatly.

Algorithm 3 Attract-Repulse Mutation

1: Initialize the new location: $\hat{\mathbf{x}}_i = \mathbf{x}_i$;
2: $s = U(1-\delta, 1+\delta)$;
3: **for** $d = 1$ to D **do**
4: $r = rand(0, 1)$;
5: **if** $r < \frac{1}{2}$ **then**
6: $\hat{\mathbf{x}}_{i,d} = \hat{\mathbf{x}}_{i,d} + (\hat{\mathbf{x}}_{i,d} - \mathbf{x}_{best,d}) \cdot s$;
7: **end if**
8: **if** $\hat{\mathbf{x}}_{j,d} > \mathbf{ub}_d$ or $\hat{\mathbf{x}}_{j,d} < \mathbf{lb}_d$ **then**
9: $\hat{\mathbf{x}}_{j,d} = \mathbf{lb}_d + |\hat{\mathbf{x}}_{j,d} - \mathbf{lb}_d| \bmod (\mathbf{ub}_d - \mathbf{lb}_d)$;
10: **end if**
11: **end for**

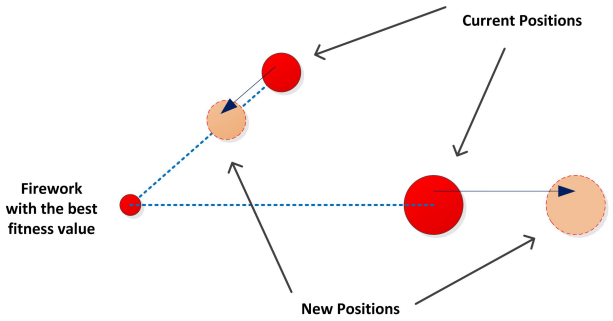

Figure 4: Attract-Repulse Mutation

3.2 Attract-Repulse Mutation

While the heuristic information is used to guide local search, other strategies should be taken to keep the diversity of the firework swarm. Keeping a diversity of the swarm is crucial for the success of optimization procedure.

In FWA, a gaussian mutation is introduced to increase the diversity of the firework swarm. In this mutation procedure, m extra sparks are generated. To generate such a spark, first, a scaling factor g is generate from $G(1, 1)$ distribution. Randomly selecting a firework, the distance between each corresponding dimension of the firework and the best current firework is multiplied by g. Thus, the new sparks can be closer to the best firework or further away from it.

Similar to gaussian mutation, in GPU-FWA, a mechanism called attract-repulse mutation (ar-mutation) is proposed to achieve this aim in an explicit way, as illustrated by Al. 3, where $\mathbf{x}_i$ depicts the i-th firework, while $\mathbf{x}_{best}$ depicts the firework with the best fitness.

The philosophy behind ar-mutation, as illustrated by Fig. 4, is that, for non-best fireworks, they either attracted by the best firework to 'help' exploit the current best location or repulsed by the best firework to explore more space. The choice between 'attract' and 'repulse' reflects the balance between exploitation and exploration.

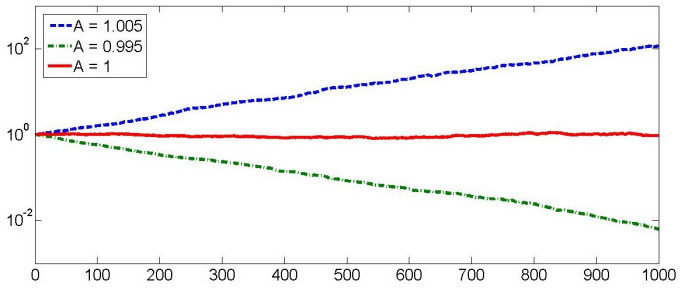

Figure 5: $E[x]$ under different values of A

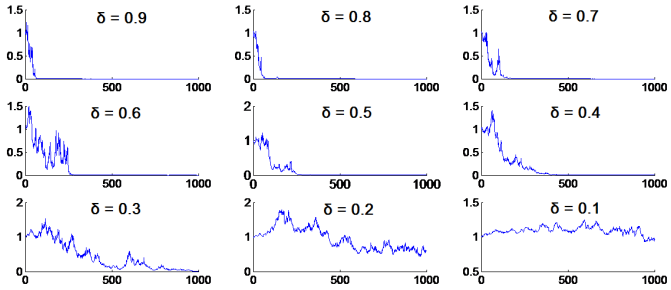

Figure 6: Simulation Results with different uniform distribution

In [15] gaussian mutation is used. But various distribution can be taken. As uniform distribution is most straightforward and easiest to utilized, we takes this strategy in the proposed algorithm.

To theoretically analyze the ar-mutation mechanism, the procedure can be simplified to a 1-order Markorv chain. Given, $x_0 = 1$, the next state is generated by the Eq. 3

$$x_{t+1} = \alpha_t * x_t \tag{3}$$

where, α_t subjects to uniform distribution between a and b, $0 < a < 1$ and $b > 1$.

Then the t-th state can be expressed by the following equation:

$$x_t = \prod_{i=1}^{t} \alpha_i \cdot x_0,$$

We can calculate the expected position,

$$E[x_t] = E\left[\prod_{i=1}^{t} \alpha_i\right] \cdot x_0 = \prod_{i=1}^{t} E[\alpha_i] \cdot x_0 \tag{4}$$
$$= \prod_{i=1}^{t} E[\alpha] \cdot x_0 = A^t \cdot x_0$$

As can be seen from Eq. 4, if the expectation of α, i.e. A, is greater than 1, then x is expected to increase exponentially; otherwise, if A less than 1, x is expected to decay exponentially. Fig. 5 demonstrates a simulation result, where tree process subject to $U(0.9, 1.11)$ $(A = 1.005)$, $U(0.9, 1.1)$ $(A = 1)$, and $U(0.9, 1.09)$ $(A = 0.995)$. As the simulation showed, even a small disturbance on $A = 1$, the results tend to diverge to infinite or converge to 0, exponentially.

As for ar-mutation, it means that fireworks are either 'repulsed' to the bounds of feasible range or 'attracted' to the current best position. Both conditions lead to prematurity and the loss of diversity.

To make sure that fireworks can 'linger' around the search space more steadily, A should take 1. The distribution should be the form of $s = U(1 - \delta, 1 + \delta)$, where $\delta \in (0, 1)$.

However, as the search range is limited , so δ should be taken with care, though A is set to 1.

As depicted by Fig. 6, from left to right, from top to bottom, δ takes 0.9 to 0.1, respectively. In the simulation, when $x > 100$, x is truncated to 10. x converges to 0 with diverse speeds. As a tendency, greater δ corresponds to faster convergency, and vice versa. But what exact convergency speed is most suitable, is task-dependent. It relies on the landscape of the objective function and how many iteration the algorithm will run.

3.3 Implementation

The flowchart of the GPU-FWA implementation on CUDA is as Fig. 7.

3.3.1 Thread Assignment

In the FWA search kernel, each firework is assigned to a single warp (i.e. 32 continual threads). But, not all the threads in the warp are necessary to be used to execute computation. If the number of sparks is set to 16, then we use the former half-warp threads, or if the number is 32, all threads in the warp are used. In our implementation, the number is set to 16.

Such an implementation brings several advantages. First, since threads in the same warp are synchronized inherently, there will cut down the overhead of inter-spark communication. Second, by keeping each firework and their sparks in the same warp, the explosion process takes place in a single block, thus the shared memory can be utilized. As accessing to the shared memory is with much lower latency than global memory, the overall running time can be greatly reduced. As GPUs automatically dispatch block according to the computing and memory resources, it is easy for the proposed algorithm to extend with the problem scale.

3.3.2 Data Organization

In our implementation, the position and fitness value of each firework are stored in the global memory, while the the data of sparks are stored in the fast shared memory. For the purpose of coalescing global memory access [8], data is usually organized in an interleaving configuration [18][13], as in Fig. 8. In this paper, we take the conventional way ,i.e. the data of the fireworks and sparks in both global and shared memory are stored in a continual manner (see Fig. 9). For in our implementation, each firework occupies a single SM. The threads running on the same SM are up to load the data of a particular firework from global memory, thus data of the same firework should be stored continually. This organization is also simpler and easier to extend with problem scale than the interleaving pattern.

3.3.3 Random Number Generation

Random number plays an important role in swarm intelligence algorithms. It can be very time-consuming to generating tremendous, high-quality random numbers. The performance of the optimization relies on the quality of random

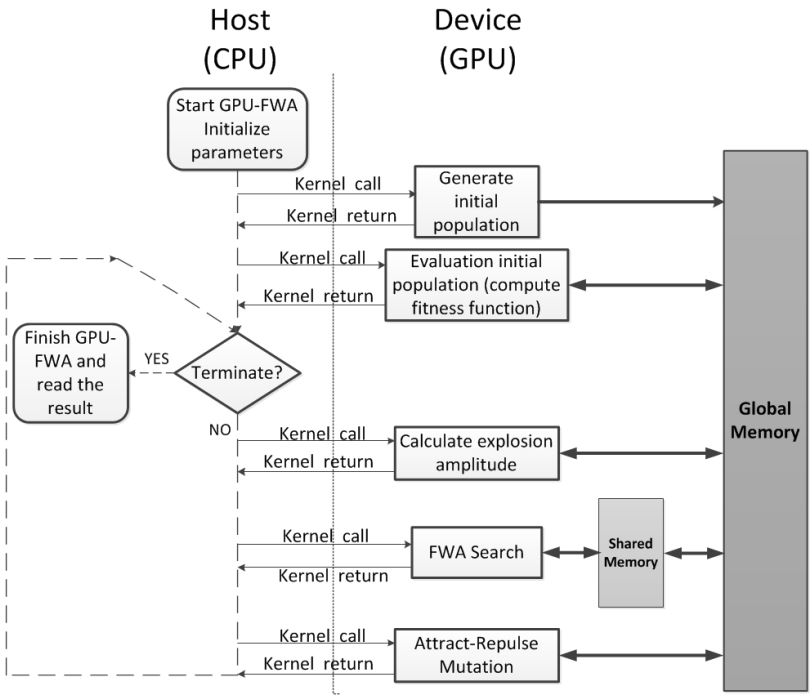

Figure 7: The Flowchart of the GPU-FWA implementation on CUDA

Memory	X_{11}	X_{21}	X_{31}	X_{41}	X_{12}	X_{22}	X_{32}	X_{42}	X_{13}	X_{23}	X_{33}	X_{43}	X_{14}	X_{24}	X_{34}	X_{44}

Figure 8: Interleaving Storage

Memory	X_{11}	X_{12}	X_{13}	X_{14}	X_{21}	X_{22}	X_{23}	X_{24}	X_{31}	X_{32}	X_{33}	X_{34}	X_{41}	X_{42}	X_{43}	X_{44}

Figure 9: Continuous Storage

numbers. For our implementation, the efficient CURAND library [10] is used for generating high-quality random numbers on the GPU.

4. EXPERIMENTS AND ANALYSIS

To empirically study the performance of GPU-FWA, extensive experiments are conducted and thoroughly analyzed.

4.1 Experimental Environment

To compare the performance, both in solution precision and runtime, we implemented two swarm algorithm, Particle Swarm Optimization (PSO) [2] and Fireworks Algorithm (FWA) [15]. We conducted our experiments on Windows 7 Professional x64 with 4G DDR3 Memory (1333 MHz) and Intel core I5-2310 (2.9 GHz, 3.1 GHz). The GPU used in the experiments is NVIDIA GeForce GTX 560 Ti with 384 CUDA cores. The CUDA runtime version is 5.0.

The benchmark functions are listed in Tab. 1 [7], of which f_1 to f_3 are unimodal functions, while f_4 to f_8 are multimodal functions. D depicts the dimension of the test functions, in our work D is set to 30.

We implemented PSO according to [2] with a ring-topology and FWA according to [15] with minor modification as mentioned in section 2.

In all simulations, we performed 20 trials for each function. For GPU-FWA, in each running, 1000 iterations were executed. FWA and PSO executed the same number of function evaluations as GPU-FWA.

For GPU-FWA, the parameters are set as follows: $n = 48$, $L = 30$, $\delta = 0.5$. As in our experimental environment, the GeForce 560 Ti GPU has 12 CUDA cores, the number of fireworks should be the multiplication of 12 and big enough to avoid waste of computational power. 48 is adopted for the comparison of precision; when comparing the speedup, 72, 96 and 144 are also used (see section 4.2).

So far, there is no theoretical rules on the criterion of the selection of L and δ. Some experiments are conducted to pre-determine them. $L = 30$ and $\delta = 0.5$ performed quit well compared to various parameter settings ($L = 10, 20, 30, 40, 50$ and $\delta = 0.1 \cdots 0.9$, as the limit of space, the results are omitted here). The total function evaluation time was $48 * 16 * 1000 = 768000$.

For a fair comparison, all of the three algorithms were tested under the same scale. Here, by saying scale ,we mean that the number of function evaluations that can be executed in parallel. For GPU-FWA, the scale in this experiment is 768, so PSO's swarm size is set as the same number. As with FWA, as the firework number takes 64, and total spark number is 640 and number of gaussian sparks is 64.

4.2 Quality of Solutions

All benchmark functions in Tab. 1 were optimized in 20 independent trails, and the average results and corresponding standard deviations are as Tab. 2.

Under the significance level of 0.01 (observe Tab 3), it can be seen that GPU-FWA outperforms FWA on $f1 \sim f6$ and f8, it only lost to FWA on f6. PSO outperforms GPU-FWA on unimodal function f2, but fail to GPU-FWA on another

Table 1: Benchmark Functions

ID	Function	Expression	Feasible bounds	Dimension	optima
f1	Sphere	$f_1 = \sum_{i=1}^{D} \mathbf{x}_i^2$	$[-5.12, 5.12]^D$	30	0
f2	Hyper-ellipsoid	$f_2 = \sum_{i=1}^{D} i \cdot \mathbf{x}_i^2$	$[-5.12, 5.12]^D$	30	0
f3	Schwefel 1.2	$f_3 = \sum_{i=1}^{D} \left(\sum_{j=1}^{i} \mathbf{x}_j\right)^2$	$[-65.536, 65.536]^D$	30	0
f4	Rosenbrock	$f_4 = \sum_{i=1}^{D-1} \left[100 \cdot \left(\mathbf{x}_{i+1} - \mathbf{x}_i^2\right)^2 + (1 - \mathbf{x}_i)^2\right]$	$[-2.048, 2.048]^D$	30	0
f5	Rastrigin	$f_5 = 10 \cdot D + \sum_{i=1}^{D} \left[\mathbf{x}_i^2 - 10\cos(2\pi\mathbf{x}_i)\right]$	$[-5.12, 5.12]^D$	30	0
f6	Schwefel	$f_6 = \sum_{i=1}^{D} \left[-\mathbf{x}_i \sin\left(\sqrt{\lvert\mathbf{x}_i\rvert}\right)\right]$	$[-500, 500]^D$	30	-1.3e+04
f7	Griewangk	$f_7 = \frac{1}{4000}\sum_{i=1}^{D} \mathbf{x}_i^2 - \prod_{i=1}^{D} \cos\left(\frac{\mathbf{x}_i}{\sqrt{i}}\right) + 1$	$[-600, 600]^D$	30	0
f8	Ackley	$f_8 = -a \cdot \exp\left(-b \cdot \sqrt{\frac{1}{D}\sum_{i=1}^{D} \mathbf{x}_i^2}\right) - \exp\left(\frac{1}{D}\sum_{i=1}^{D} \cos(c\mathbf{x}_i)\right) + a + \exp(1)$	$[-32.768, 32.768]^D$	30	0

Table 2: Precision Comparison

Fun	GPU-FWA		FWA		PSO	
	Avg.	Std.	Avg.	Std.	Avg.	Std.
f1	**1.31E-09**	1.85E-09	7.41E+00	1.98E+01	3.81E-08	7.42E-07
f2	1.49E-07	6.04E-07	9.91E+01	2.01E+02	**3.52E-11**	1.15E-10
f3	**3.46E+00**	6.75E+01	3.63E+02	7.98E+02	2.34E+04	1.84E+04
f4	**1.92E+01**	3.03E+00	4.01E+02	5.80E+02	1.31E+02	8.68E+02
f5	**7.02E+00**	1.36E+01	2.93E+01	2.92E+00	3.16E+02	1.11E+02
f6	-8.09E+03	2.89E+03	**-1.03E+04**	3.77E+03	-6.49E+03	9.96E+03
f7	1.33E+00	1.78E+01	**7.29E-01**	1.24E+00	1.10E+00	1.18E+00
f8	**3.63E-02**	7.06E-01	7.48E+00	7.12E+00	1.83E+00	1.26E+01

Table 4: Running Time and Speedup of Rosenbrock

n	FWA(s)	PSO(s)	GPU-FWA(s)	SU(FWA)	SU(PSO)
48	36.420	84.615	0.615	59.2	137.6
72	55.260	78.225	0.624	88.6	125.4
96	65.595	103.485	0.722	90.8	143.3
144	100.005	155.400	0.831	120.3	187.0

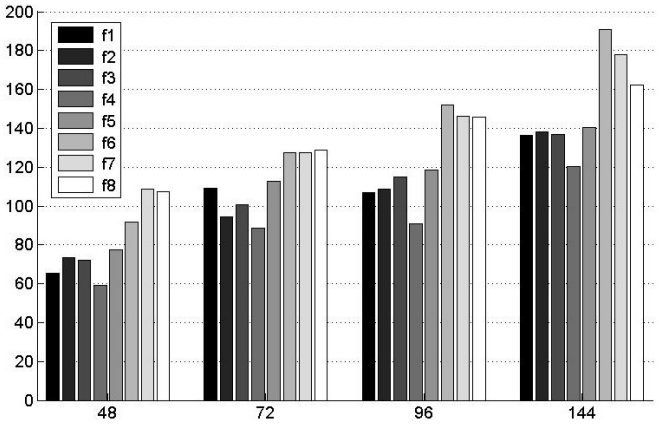

Figure 10: Speedup vs. FWA

unimodal function f3. GPU-FWA can get better results on multimodal functions f4, f5, f6, f8. In general, as far as the benchmark functions are concerned, we can see that GPU-FWA performs better than FWA and PSO.

4.3 Speedup VS. Swarm Size

Besides the precision of the optimization results, speedup efficiency is another critical factor we must consider.

To observe the speedup GPU-FWA can achieve in comparison with PSO and FWA, a series of experiments were conducted, where n is set respectively to $48, 72, 96, 144$ for GPU-FWA. 1000 iterations are run, and the same function evaluation time under the same scale for PSO and FWA.

The running time (in seconds) and speedup with respect to Rosenbrock function is illustrated by Tab. 4. Fig. 10 and Fig. 11 depict the speedup of all the 8 benchmark functions with respect to the swarm size.

GPU-FWA achieved a speedup as high as 180x with the scale of less than 200, in the meantime, the up-to-date GPU accelerated PSO achieve 200x fold speedup with the scale

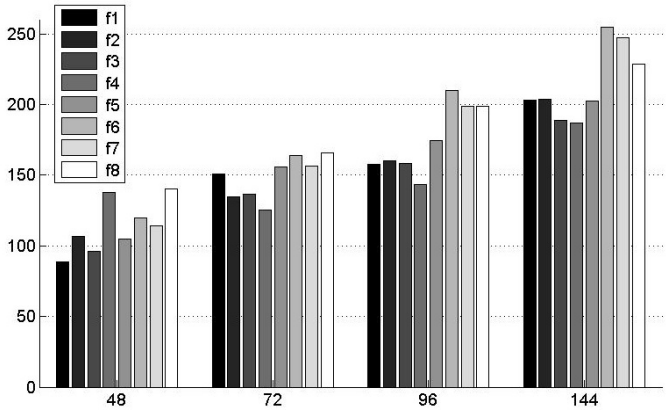

Figure 11: Speedup vs. PSO

Table 3: p-values of t-test

	f1	f2	f3	f4	f5	f6	f7	f8
GPU-FWA vs. FWA	**1.00E-06**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**	5.16E-01	**0.00E+00**
GPU-FWA vs. PSO	3.46E-01	**1.21E-04**	**0.00E+00**	**2.15E-02**	**0.00E+00**	**6.50E-03**	8.03E-01	**1.21E-02**

high up to 10000 [13]. Thus the proposed GPU-FWA are more scalable than the conventional GPU-based PSO.

5. CONCLUSION

Swarm intelligence is a kind of population-based meta-heuristic optimization algorithms. As GPU computing has come into the mainstream, it has attracted more and more interest from the field of swarm intelligence for algorithm acceleration purpose. Although GPUs provide great computing power at an affordable cost, it is not clear whether a conventional paradigm is suitable for expressing parallelism in a way that is efficiently implementable on GPU architectures. To take benefit of GPUs, in this paper, a novel swarm intelligence algorithm, GPU-FWA, for optimization is presented. The proposed algorithm can fully leverage the great computing power of the GPU architecture, lending itself very well to parallel computation. It do not need special complicated data structure, thus making it easy to implement. As the problem scale goes great, it can extend in an easy and natural way. The new method requires few control variables, thus is robust as well as easy to use.

Tested on suite of benchmark functions, it is demonstrated that the new method outperform FWA and the popular, well-researched PSO in the quality of solution. Experimental results obtained a speedup up to 160x and 200x compared to CPU-based FWA and PSO respectively on an up-to-date CPU .

We can conclude that GPU-FWA is a potential powerful tool for solving large-scale optimization problems on the massively parallel architecture.

6. ACKNOWLEDGMENTS

This work was supported by National Natural Science Foundation of China (NSFC), Grant No. 60875080 and No. 61170057.

7. REFERENCES

[1] P. E. Andries and P. Engelbrecht. *Fundamentals of Computational Swarm Intelligence*. Wyley, 2005.

[2] D. Bratton and J. Kennedy. Defining a standard for particle swarm optimization. In *Swarm Intelligence Symposium, 2007. SIS 2007. IEEE*, pages 120–127, April 2007.

[3] L. de P. Veronese and R. A. Krohling. Swarm's flight: Accelerating the particles using c-cuda. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pages 3264–3270, May 2009.

[4] R. C. Eberhart, Y. Shi, and J. Kennedy. *Swarm Intelligence*. Morgan Kaufmann, San Francisco, California, 2001.

[5] A. Janecek and Y. Tan. Swarm intelligence for non-negative matrix factorization. *International Journal of Swarm Intelligence Research*, 2(4):12–34, October-December 2011.

[6] P. Krömer, V. Snášel, J. Platoš, and A. Abraham. Many-threaded implementation of differential evolution for the cuda platform. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 1595–1602. ACM, 2011.

[7] M. Molga and C. Smutnicki. *Test functions for optimization needs*, 2005.

[8] NVIDIA. *NVIDIA CUDA C Best Practices Guide 5.0*, October 2012.

[9] NVIDIA. *NVIDIA's Next Generation CUDA^{TM} Compute Architecture: Fermi^{TM}*, 2012.

[10] NVIDIA. *Toolkit 5.0 CURAND Guide*, September 2012.

[11] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[12] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krĺźger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[13] V. Roberge and M. Tarbouchi. Parallel particle swarm optimization on graphical processing unit for pose estimation. *WSEAS TRANSACTIONS on COMPUTERS*, 11(6):170 –179, June 2012.

[14] S. Solomon, P. Thulasiraman, and R. Thulasiram. Collaborative multi-swarm pso for task matching using graphics processing units. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 1563–1570, New York, NY, USA, 2011. ACM.

[15] Y. Tan and Y. Zhu. Fireworks algorithm for optimization. In *Advances in Swarm Intelligence*, volume 6145 of *Lecture Notes in Computer Science*, pages 355–364. Springer Berlin Heidelberg, 2010.

[16] M.-L. Wong, T.-T. Wong, and K.-L. Fok. Parallel evolutionary algorithms on graphics processing unit. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 3, pages 2286–2293, September 2005.

[17] Y. Zhou and Y. Tan. Gpu-based parallel particle swarm optimization. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pages 1493–1500, May 2009.

[18] Y. Zhou and Y. Tan. Gpu-based parallel multi-objective particle swarm optimization. *International Journal of Artificial Intelligence*, 7(A11):125–141, October 2011.