國 立 交 通 大 學

資訊科學與工程研究所

碩 士 論 文

一 個 創 新 的 平 行 蜂 群 演 算 法

實 作 在 圖 形 處 理 器 架 構

A novel parallel Bees Algorithm for optimization problems on GPU

研 究 生：黃聖凱

指導教授：袁賢銘　教授

中 華 民 國 一百零一 年 六 月

一個基於圖形處理器的平行式蜂群演算法
用以解決最佳化問題

A novel parallel Bees Algorithm for optimization problems on GPU

研　究　生：黃聖凱　　　　Student：Sheng-Kai Huang

指導教授：袁賢銘　　　　Advisor：Shyan-Ming Yuan

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Department of Computer and Information Science

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer and Information Science

June 2012

Hsinchu, Taiwan, Republic of China

一個創新的平行蜂群演算法
實作在圖形處理器架構

研究生：黃聖凱　　　　　指導教授：袁賢銘

國立交通大學資訊科學與工程研究所碩士班

摘要

尋求最佳解在電腦科學或和其他工程領域裡面是很重要的一環，然而尋求最佳解屬於 NP 問題，取而代之去尋近似最佳解的演算法也越來越多。Swarm Intelligence 是透過觀察大自然中社會型動物的生活方式應用在人工智慧來解決問題，蜂群演算法就是其中一個。由於找尋近似最佳解的演算法也 CPU bound job，一些平行處理的演算法也因此相繼被人提出，到目前為止，鮮少人關心蜂群演算法的平行化。

圖形處理器(GPU)是專門針對圖形處理而設計的架構，因為影像處理具備高度平行化的特性，造就了 GPGPU 的發展，許多的研究也開始利用 GPU 進行大量的平行運算。近年，由 NVIDIA 提出 GPGPU 的解決方案 CUDA(Compute Unified Device Architecture)讓開發者利用熟悉的 C/C++就可以在上面開發自己的平行應用程式。

本論文針對需要大量運算的蜂群演算法提出許多方法使其平行化且適合在 CUDA 的平行架構運行，並針對幾個知名的最佳化問題做了效能上面的測試。根據我們的實驗結果，可以發現平行化的 CUDA Bees Algorithm (CUBA) 利用龐大數量的處理器，在各個最佳化問題比起傳統的 Bees Algorithm(BA)至少有 3X 倍效能的加速。

A novel parallel Bees Algorithm for optimization problems on GPU

Student: Sheng-Kai Huang Advisor: Shyan-Ming Yuan

Department of Computer Science and Engineering

National Chiao Tung University

Abstract

Searching the solution of optimization problems is a very important work in computer science and engineering field. The problems are belong to NP problem, so more and more algorithms are developed to find approximate solutions instead of the real solutions. Swarm Intelligence is the collective behavior of the system of social animals in nature. The concept is used in algorithm on artificial intelligence(AI). The Bees Algorithm is one of the works. Because these algorithms are computational bound jobs, some parallel swarm based algorithms are proposed in recent years. Even so, few works are developed base on The Bees Algorithm.

GPU is a special architecture of graphic processor. The highly parallel features of graphic processing made the rapid development of General Purpose GPU(GPGPU). A lot works for massive parallel computing on GPU are proposed. NVIDIA provide a general purpose parallel model, thus programmer could use their familiar language C/C++ to develop own parallel applications.

In this paper, we do many modifications for The Bees Algorithm and make it adapt to run in the parallel architecture of CUDA. We also test the performance accelerations for numerous famous optimization problems. Finally, the result shows the CUDA Bees Algorithm(CUBA) we proposed perform at least 3x times faster than traditional BA in numerous different optimization problems.

Acknowledgement

碩士班生涯是我最重要的人生過程之一，很慶幸加入分散式系統實驗室。非常感謝我的指導教授**袁賢銘**老師這兩年來對我不厭其煩的教導，老師開明且自由的指導方式，讓我能夠在碩士班期間學到不少專業領域的相關知識，老師充分給予學生想法揮灑的空間，也會耐心的給予各種寶貴的建議。在這兩年，無論是大大小小的程式競賽，或是研究計畫，甚至在找工作方面，老師給我非常多的幫助，讓我能夠順利完成這些任務。實驗室方面非常感謝**國亨**學長在這期間，在我研究遇到瓶頸挫折的時候，給予我一些專業的方向，刺激我的思考，讓我能夠走過低潮，也感謝**永威**學長、**家峰**學長以及江川彥不吝在各個方面提供我幫助。感謝我實驗室的同學們**紘維**、**冠穎**、珮瑜在這兩年陪伴，不論在課業上，研究上，生活上讓我能夠學習成長。**孟傑**、**俊凱**在這一年的加入，讓實驗室更加的歡樂、溫馨。學弟們**先博**、**柏志**、**振庭**、**丞訓**讓實驗室充滿著朝氣，每個禮拜的球聚，以及不定期的餐聚，凝結著我們DCSLAB團結的心。感謝實驗室的大家帶給我歡笑與成長。在最後我要感謝我的家人，沒有你們，就沒有今天的我，有了你們的支持，我的人生過得很充實。

Table of Contents

List of Table

List of Figures

Chapter 1 Introduction

1.1 Preface

Optimization problems have been the subject of much research in recent years. It’s a NP-problem, so many different alternative techniques have been developed. The swarm intelligence is one of those methods which is used to solve the near optimal solution. Many researchers have introduced various algorithms by modeling the behaviors of the swarm of animals in nature [1-5]. Self-organization is the feature of the system which gets global-level response by means of many different low-level interactions.

The Bees Algorithm was proposed by DT Pham [1] in 2005 for optimizations problems, and the improved performance of the algorithm have been proposed several years latter [6]. Researchers have come up with several real-word applications such as data mining [7], robot controlling [8], electronic engineering [9], job scheduling for the Bees Algorithm [10].

1.2 Motivation

Because of the optimization problems are computational issues, we want to find a parallel way that can speed up the Bees Algorithm. Nowadays modern Graphic Processing Units (GPU), which can be seen as highly fast parallel general-purpose systems. Developers have designed many algorithms and applications on GPU for better performance. Moreover, several general purpose languages for GPUs have become popular such as CUDA [11, 12]. It supports many graphic programming APIs, so developers do not have to consider more complexity of low-level problems while programming with CUDA. Although much work has be done on developing parallel swam intelligence algorithm on GPU such as Ant Colony Optimization[13-15], Genetic Algorithm[16, 17] and so on, no attention has been paid to develop the parallel bees algorithm on GPU. The purpose of this paper is to develop a novel parallel Bees Algorithm which is adapt running on GPU.

1.3 Research Objective

For the reasons we mentioned above, we implemented a parallel Bees Algorithm to test the speedup for several common functions of optimizations problems. We evaluated and compared both of the execution time with CUDA on GPU and the execution time with C++ on CPU to verify efficiency of the algorithm.

1.4 Research Contribution

The following are our research contributions:

1. A novel parallel Bees Algorithm on GPUs
2. Tens times speedup than traditional version on CPU

Chapter 2 Background and Related Work

2.1 Optimization problems

The *standard form* of a (continuous) optimization problem is

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0, \quad i = 1, \ldots, m \\ & h_i(x) = 0, \quad i = 1, \ldots, p \end{aligned}$$

where

- $f(x) : \mathbb{R}^n \to \mathbb{R}$ is the **objective function** to be minimized over the variable x,
- $g_i(x) \leq 0$ are called **inequality constraints**, and
- $h_i(x) = 0$ are called **equality constraints**.

By convention, the standard form defines a **minimization problem**. A **maximization problem** can be treated by negating the objective function.[18-20]

2.2 Intelligent swarm-based optimization Algorithm

Many complex multi-valuable optimization problems can't be solved within polynomial computation times. For the reason, many researchers interested in search algorithms which finding approximate optimal solutions in reasonable running time. Swarm intelligence is the field of optimization and researchers have developed various algorithms by modeling the behaviors of different swarm animal with social organization such as ants, bees, birds…and so on. In 1990s, Those algorithm inspired by ants like Ant Colony Optimization had been proposed by Marco Dorigo [1]. Kennedy also developed the Particle swarm optimization (PSO) [3]. Those algorithm inspired by honey bees such as The Bees Algorithm by DT Pham [2] in 200, and the Artificial Bee Colony Algorithm (ABC) by D. Karaboga [5] in 2007.

2.3 The Bees Algorithm

The Bees Algorithm is population-based method to search optimization of the problems which is inspired by the behavior of honey bees in nature[2, 6]. It requires several parameters to be set as following: n (number of scout bees), m (number of sites to be selected from n visited sites), nep(number of bees recruited for top e sites from the m visited sites), nsp(number of bees recruited for the other (m-e) selected sites), ngh initial size of patches which includes site and its neighbourhood and stopping criterion. The algorithm begins with the n scout bees which randomly being placed in the searching domain. The basic Bees Algorithm is shown in following and the corresponded flowchart is in Figure 2-1

```
1. Initialize populations with random solutions.
2. Evaluate Fitness of the population.(see the fig.)
3. While(the stopping criterion is not met)
   //Forming new population.
4. Select sites for neighbourhood search.
5. Recruit bees for selected sites( nep bees for top e sites and nsp bees for remain (m-e) sites)
   and evaluate fitness.
6. Select the fittest bee from each patch.
7. Assign remaining bees (n-m) to search randomly and evaluate fitness.
8. End While
```

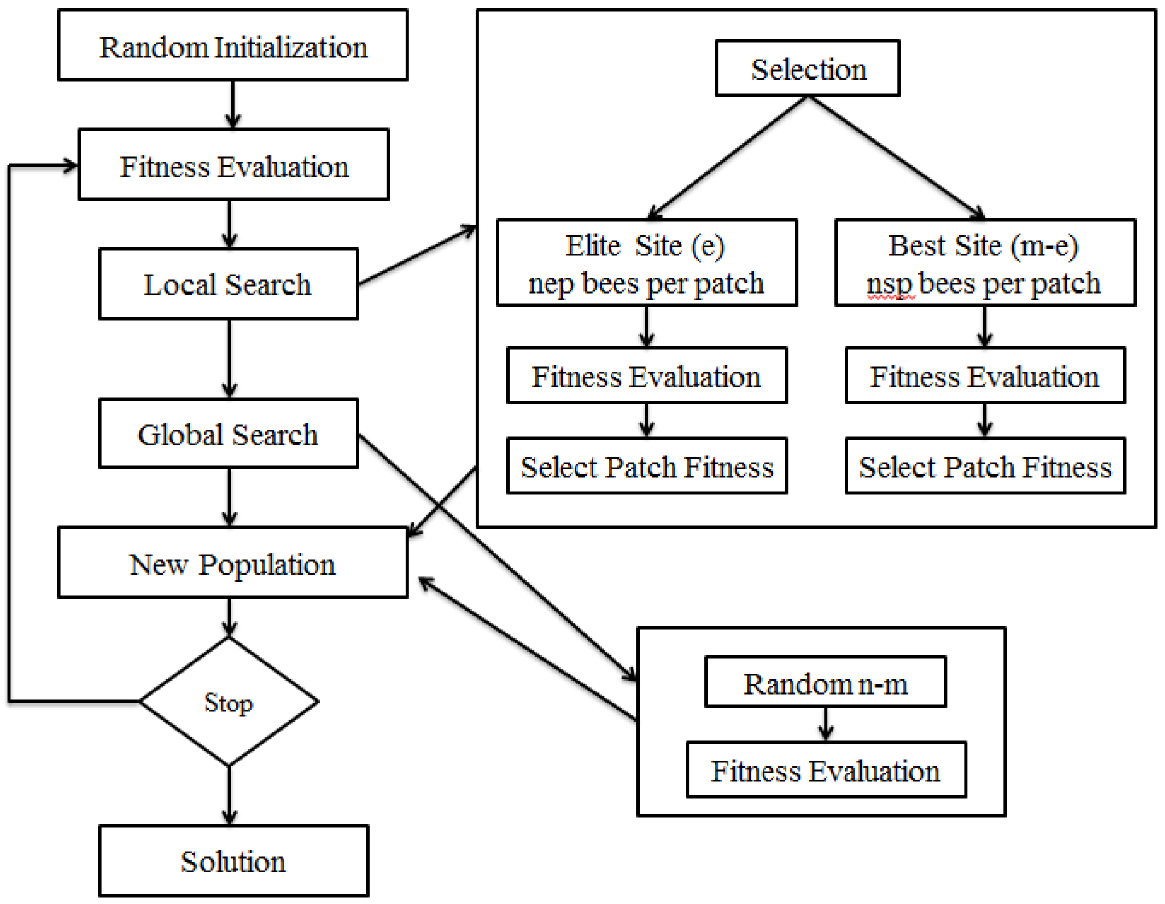


Figure 2 - 1 Flowchart of the basic Bees Algorithm [6]

The Bees Algorithm above is the most basic version. Pham DT proposed an improved version for the Bees Algorithm that increases the search accuracy and avoid superfluous computations in 2009. Two new procedures were introduced as follows:

1. neighbourhood shrinking

The size of ngh is initially set to a large value as following:

ngh(0) = (maxV - minV)

where the maxV, minV means the max and min searching site in the global area. The local search is initially defined over a large neighbourhood (equal to the range of the global search), and has largely explorative feature. The local search procedure finds any better site with higher fitness, it keeps the size of ngh unchanged. If no improvement during the step, then the size of ngh be decreased. The updating formula is shown by following:

ngh(i+1) = 0.8 *ngh(i) if no improvement

ngh(i+1) = ngh(i) else

2. **Site abandonment**

   When no fitness improvement after a number of times (stlim) local search even by neighbourhood shrinking method, it means the local search procedure perhaps to reach the top of the local fitness peak, in other words, no further progress will be made. For efficiency, the exploration of the patch is stopped. If no better fitness of other site is generated during the remaining random search procedure then abandons this site.

Although there are several researchers come up with new models based on honeybees our work is based on this model proposed by Pham DT.

2.4 General Purpose Computation on GPU

GPGPU is the use of a GPU (graphic processing unit) as a co-processor to accelerate GPUs for general purpose scientific and engineering computing .

The GPU accelerates computations and applications running on the CPU by loading part of the code with high compute-loading. The rest of the code is runs on CPU. To accelerating application by using the massively parallel processing power of the GPU to get high performance is also call "hybrid" way of computation.

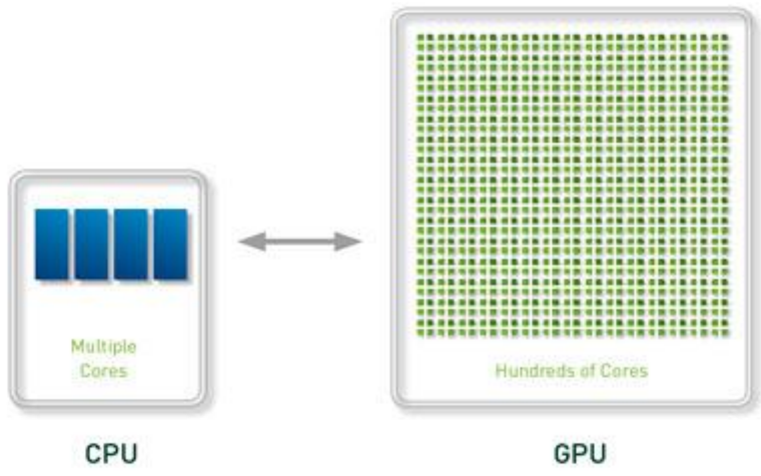


Figure 2 - 2 The comparison of computation power between CPU and GPU .

As illustrated by Figure 2-2, nowadays a CPU consists of 4 to 8 cores while the GPU consist of hundreds of cores. They cooperate with each other in the application. The massively parallel computing architecture gives the application higher performance.

GPUs now offer much faster floating-point calculation than CPU as illustrated by Figure 2-3, moreover, several high-level languages for GPGPU such as CUDA and OPENCL have developed for programmers. The main difference between CPU and GPU is that the GPU is specialized for compute-intensive, highly parallel computations, GPU devotes more transistors to process data rather than to cache data and flow control as illustrated by Figure 2-4.

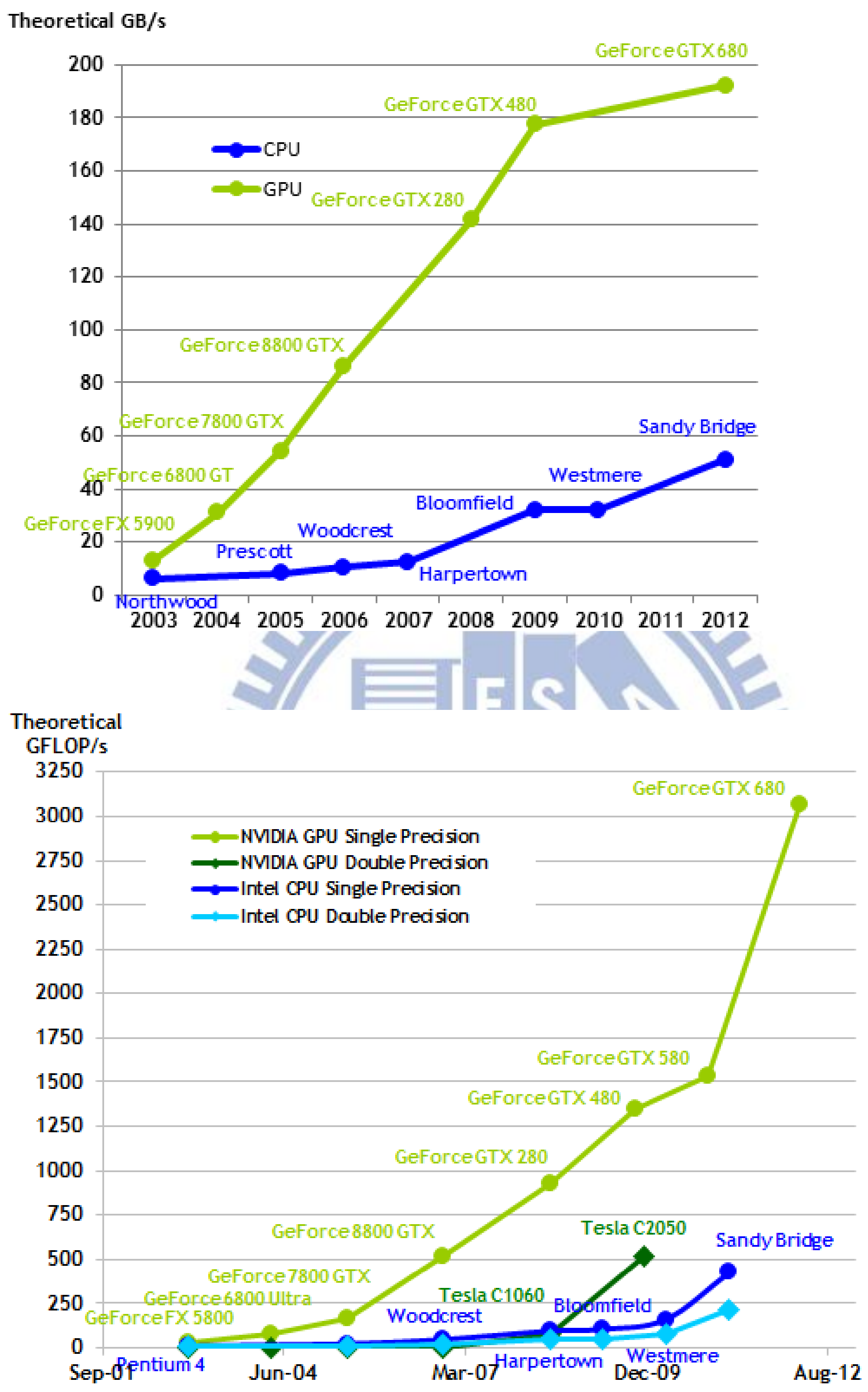


Figure 2 - 3 Floating-Point Operations per second and memory bandwidth for the CPU and GPU [11]

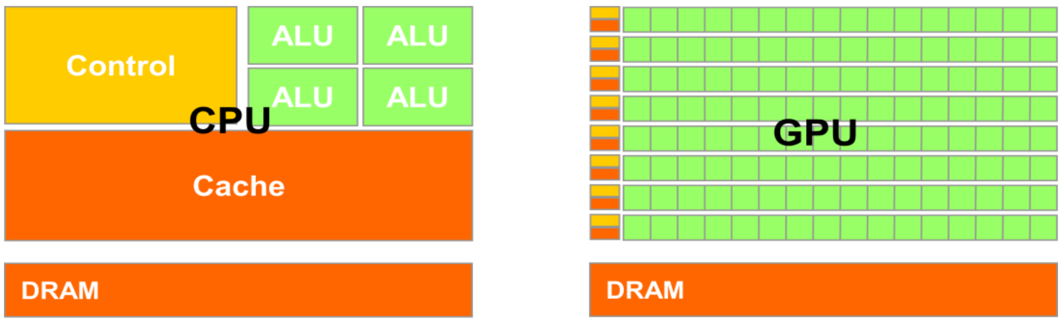


Figure 2 - 4 The GPU Devotes More Transistors to Data Processing [12]

2.5 Compute Unified Device Architecture (CUDA)

NVIDIA provides CUDA that is a general purpose parallel programming model, thus the programmers don't need to consider the complex low-level issues of GPU. What they have to concern is how to design an parallel algorithm for their applications. The CUDA programming model provides various languages including C/C++ for developers.

From the perspective of hardware, A GPU consists of a number of multiprocessors, there are many stream processors in a multiprocessor and each stream processor is a smallest computational unit. There is shared memory in a multiprocessor among numerous stream processors, they could communicate by using shared memory. In addition to shared memory, other types of memory like constant memory, texture memory that could be used in different situations. Figure 2-5 is a diagram of the hardware viewpoint of CUDA architecture. From the perspective of software, the code running sequentially on CPU is called "Host" and the code running paralleled on GPU is called "Kernel". A kernel is launched as a grid of thread blocks, the thread blocks are executed on multiprocessors. The software viewpoint of CUDA architecture and memory model is shown in Figure 2-6

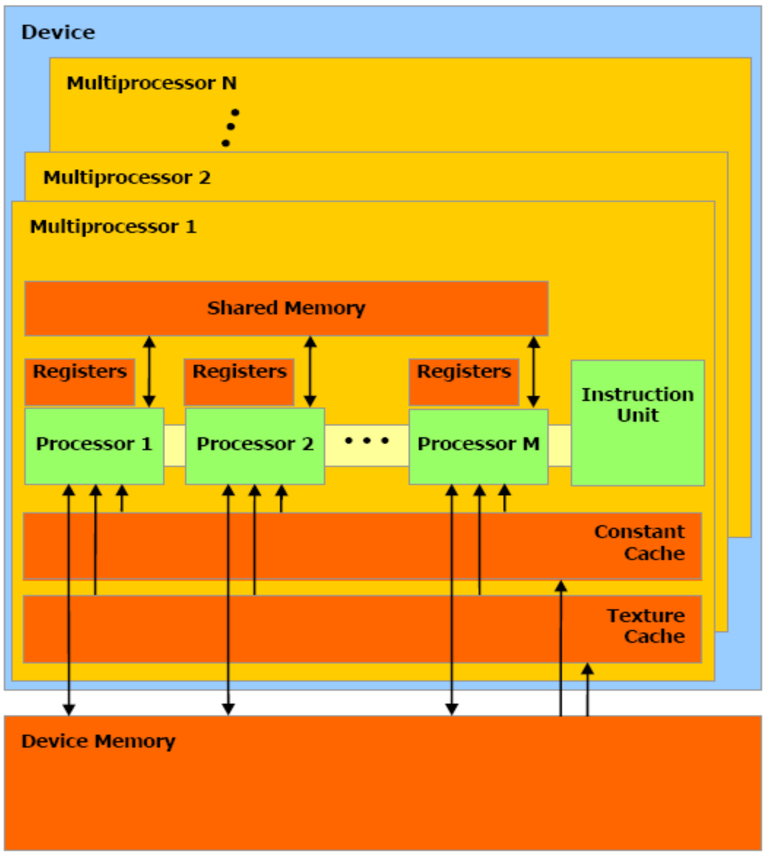


Figure 2 - 5 hardware viewpoint of CUDA architecture

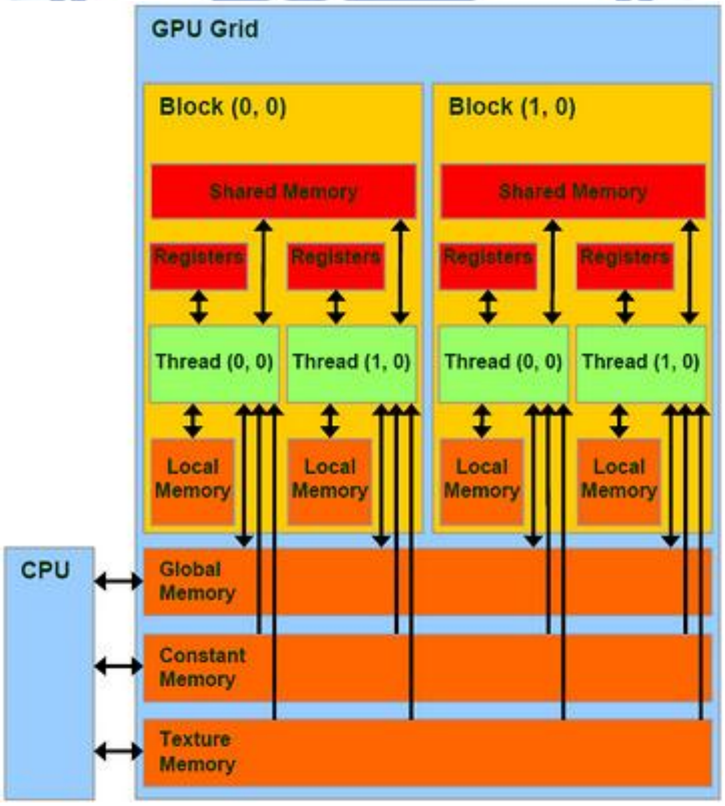


Figure 2 - 6 Software viewpoint of CUDA architecture

2.6 Related Work

2.6.1 Parallel Bees Algorithm for ATC Enhancement in Modern Electrical Network

The paper has proposed a parallel Bees Algorithm for determining the optimal allocation of FACTS devices[21]. The PAB(parallel Bees Algorithm) simultaneously for nearby searches. In PAB computations are distributed among the CPUs by matlab workers, thus it's faster to search a solution and getting better accuracy of solution compared to other technique. It's the first application of parallel Bees Algorithm in application of FACT devices. The method to parallel the algorithm is to distribute the computations of evaluating fitness, and the main difference between serial and parallel approach is shown in Figure 2-7. When they compared the elapsed time for the application on Intel Quad Core Q6600 running at 2.4GHz system in matlab 7 environment, the result is illustrated by the Figure 2-8 and getting 2~4 times better.

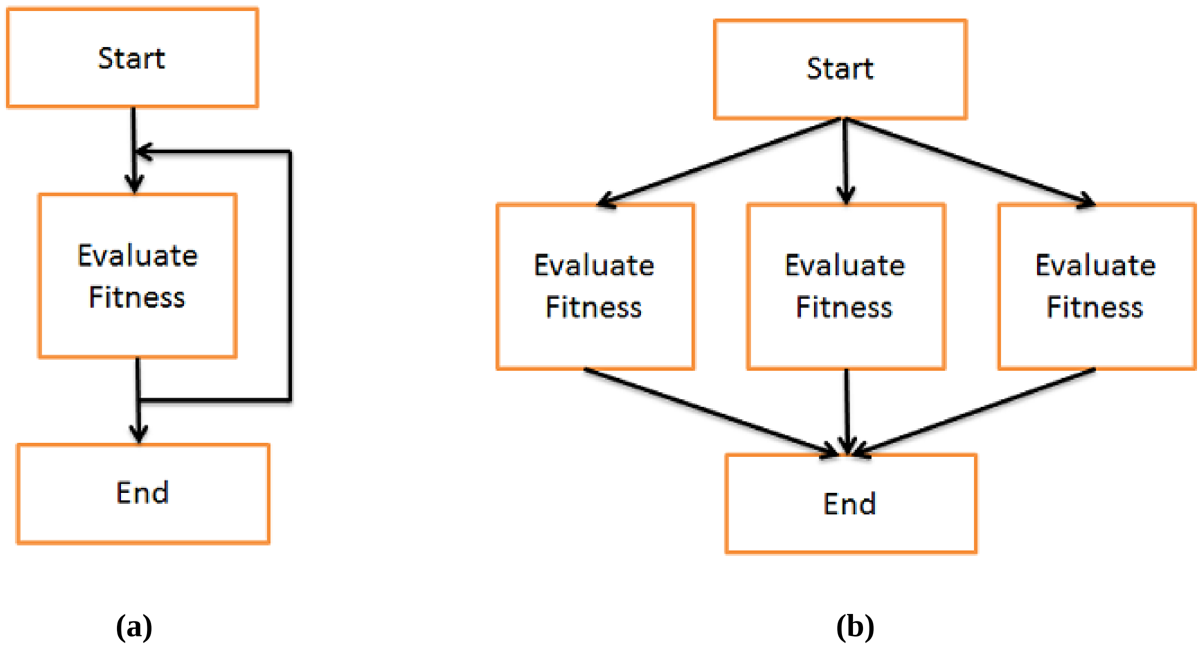


Figure 2 - 7 (a)Serial Approach (b)Parallel Approach [21]

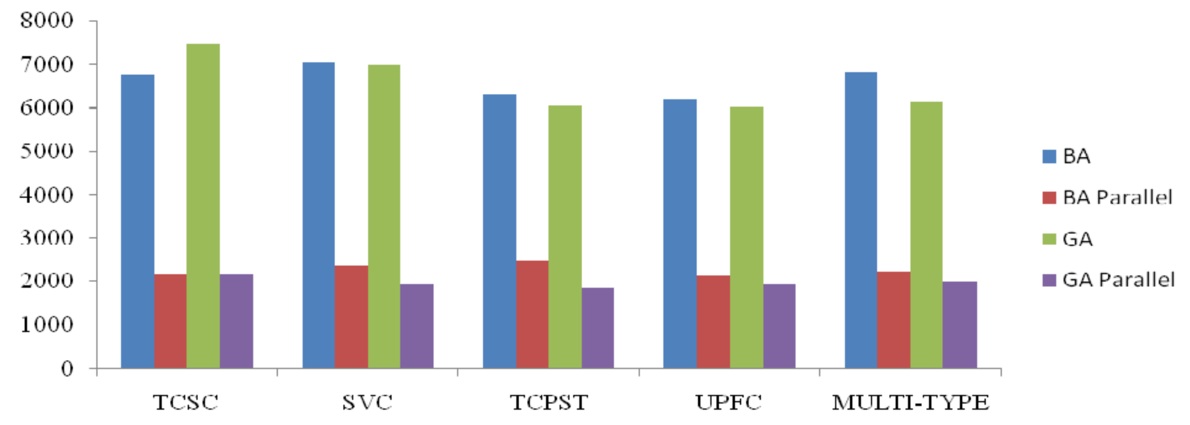


Figure 2 - 8 Comparison of Elapsed Time for Different Algorithms in seconds [21]

Chapter 3 Parallel Bees Algorithm on GPU

The key that decides the accelerated effect is the level of parallelization. In the traditional Bees Algorithm, the most computational loading is in neighbourhood search procedure. A naïve method is to take the neighbourhood search procedure as a kernel to distribute the computations in loop of the procedure. In fact, the optimal number of the neighbourhood size is fluctuant according to different features of functions. However, if the size of the neighbourhood is not larger than number of the total threads within the GPU then the accelerated effect would not be obvious. Another common solution is "multi-colonies" that means we should run many Bees Algorithms independently in each threads. There are two major disadvantages. The first disadvantage is each thread contains many conditional branch, we could not avoid the divergent branch, so the overhead would be too expensive. The second one is that the communication among the threads after a round would be more complex to do. For these reasons, we design a new Bees Algorithm of parallel multi-colonies that bring good efficiency.

3.1 System Overview

We choose CUDA framework to implement our multi-colonies Bees Algorithm on GPU called CUBA. In our algorithm, we group the threads within a block to several colonies. To explain clearly, each thread is assigned to a honey bee to search the solution for its colony. We divide a block into different colonies by thread ID, and running Bees Algorithm independently. When one iteration finished, we will change the information between colonies in the same block by using shared memory. The colonies will not communicate with each other if they are in different blocks because the shared memory is shared by threads in the same block, and it's not efficient if we shared the information by using global memory. The communication step is critical for converge time. The Figure 3-1 shows an overview of our system, and the detail will be described latter.

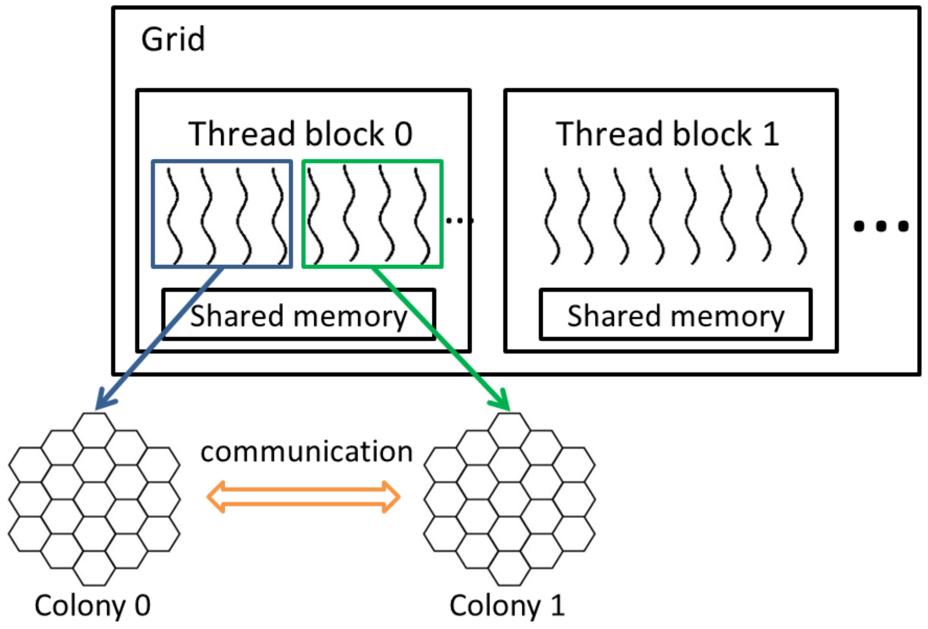


Figure 3 - 1 Framework of the CUBA

3.2 Parallel Approaches

To parallel The Bees Algorithm, we have overcome many issues. Our CUDA bees algorithm shows in Figure 3-2, and the detail will be explained in this section.

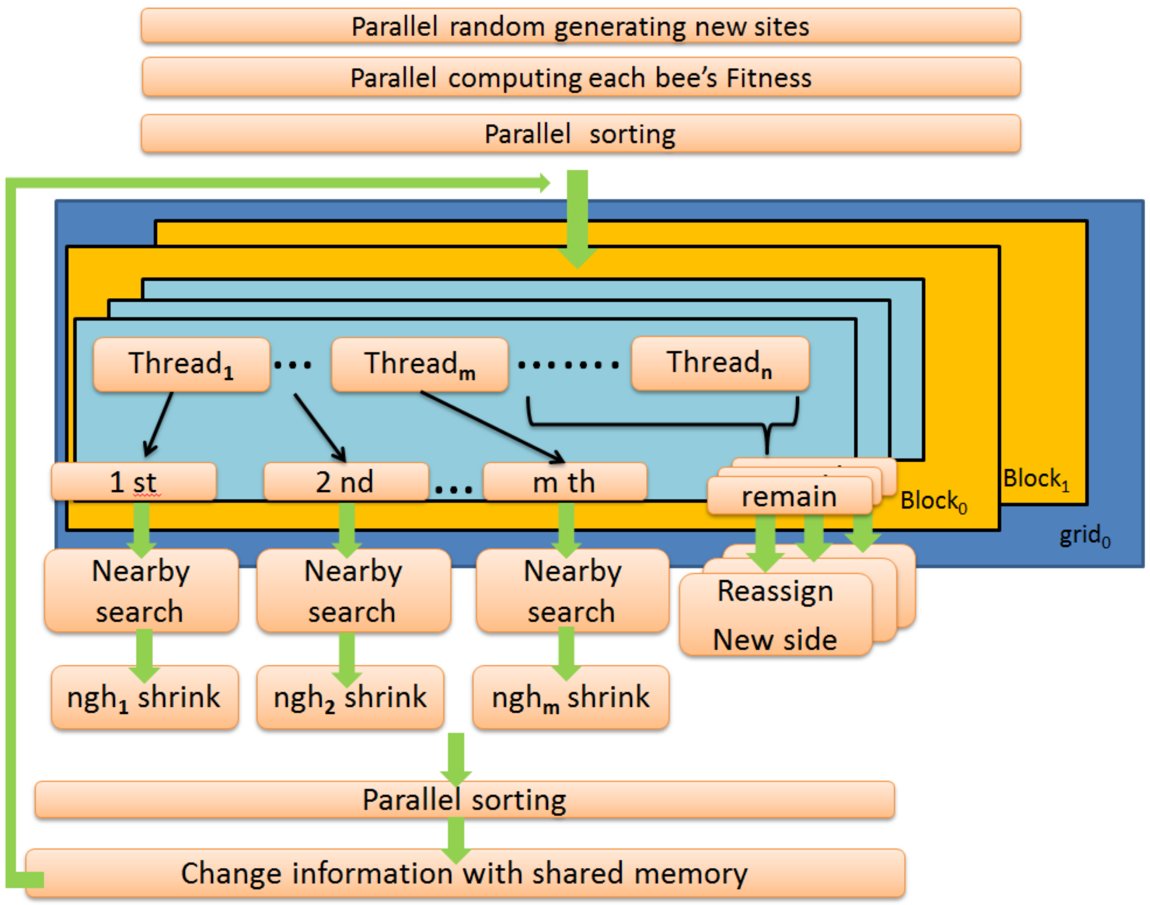


Figure 3 - 2 Algorithm of the CUBA

3.2.1 Parallel Initialization

In BA approach, the initialization of population and the evaluation of the fitness of the population achieve one after one whereas CUBA distributes and computes them among the threads of GPU. Ideally, it accelerates times of the numbers of threads for this procedure.

3.2.2 Odd–Even Sort

It's necessary to sort the fitness of all populations to get the best m sites. Because the size of the sorting data in this application is small respect to others, we sort the colonies in the same block individually by using Odd–Even Sorting algorithm [22, 23] that is based on the Bubble Sort technique of comparing two elements and switching them by the comparing rule.

This method only requires n/2 iterations of the two phase sort. The procedure diagram of Odd-Even Sorting in Figure 3-3 and the algorithm in Table 3-1

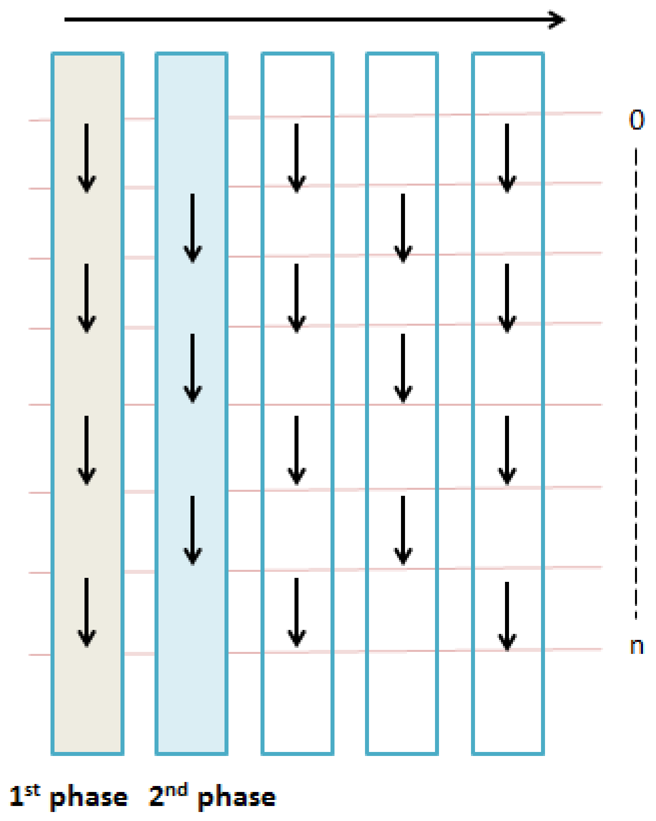


Figure 3 - 3 Procedure of Odd-Even Sort

Algorithm **Odd-Even Sort**

```
Input:    array A

Declare max = sizeof(A)
Run max/2 times:
For i is odd and i < max do in parallel:
    If A[i] > A[i+1] then swap(A[i], A[i+1])
For i is even and i < max do in parallel:
    If A[i] > A[i+1] then swap(A[i], A[i+1])
Output:    sorted array A
```

Table 3 - 1 Odd-Even Sort Algorithm

3.2.2 Group Bees into Different Colonies

We divide threads in the blocks to different colonies according to their thread ID, each thread is assigned to a honey bee and searching the solution for its colony, so there are a number of colonies run Bees Algorithm parallel. The number of bees and colonies in the algorithm is depending on what the number of blocks per grid and number of threads per block we set.

The number of colonies in a block = number of threads per block / number of bees per colony.

3.2.3 Modified Bees Algorithm

3.2.3.1 Modification of local search

The local search in traditional BA approach, more bees (nep) recruit for elite sites and fewer bees (nsp) recruits for the rest of sites from e sites. It's reasonable because the mechanism is based on probability. But in our system, we just assign nep bees to recruit m sites for balancing the loading among the threads, to be more precisely, it's not make sense in parallel architecture if some threads would do nothing after finishing their jobs and waiting for the others.

3.2.3.2 Random Seeds

We have different threads in GPU with different random seeds, so we get more random effect.

3.2.3.3 Neighbourhood Shrinking

According to the new procedure "neighbourhood shrinking" in BA, ngh constantly change values, In our approach, we have numerous local searching in different sites simultaneously for parallelism, and we let the recruited bees in different sites with different ngh. Another adjust is that we don't need to set a such large number of the recruited bees like in BA, because we have so numerous colonies to search simultaneously that the risk which may cause wrong shrink we accept is much lower. In the meanwhile, the rapid decreasing of ngh could bring a faster converge time. What shrinking equation we use is the same with the equation in the Bees Algorithm. Initially, the size of ngh is set to a large value.

3.2.3.4 Communication with shared memory

In general parallel architectures may use shared memory or message passing method to communicate between the multiple processing units. There is a shared memory in the same block in CUDA architecture, so we use it to implement the communication in the end of the each iteration. In this strategy, there are three issues we have to concern. The first is what information to share, the second one is who to share with, and the last is how long to communicate once.

We had tried and compared several mechanisms for communication. For example, we sort the best results which are gained from individual colonies in the same block after neighbourhood search, and sharing the best to others. To explain in detail, the site with lowest fitness in each colony is replaced by best one with highest fitness in the block. The result shows that converge rate is quite good. However, a sorting procedure often impact on the execution time, finally, we develop the two-phase communication that avoiding sorting and with good converge rate, too. The paired exchange take few time to share, and the second phase improve the global convergence over time. The method is shown as follow:

Adjacent exchange (first phase):

If colony ID is odd, then exchange with colony (ID+1) % number of colonies per block

If colony ID is even, then exchange with colony (ID-1) % number of colonies per block

Skip one exchange phase(second phase)

If colony ID is odd, then exchange with colony (ID+2) % number of colonies per block

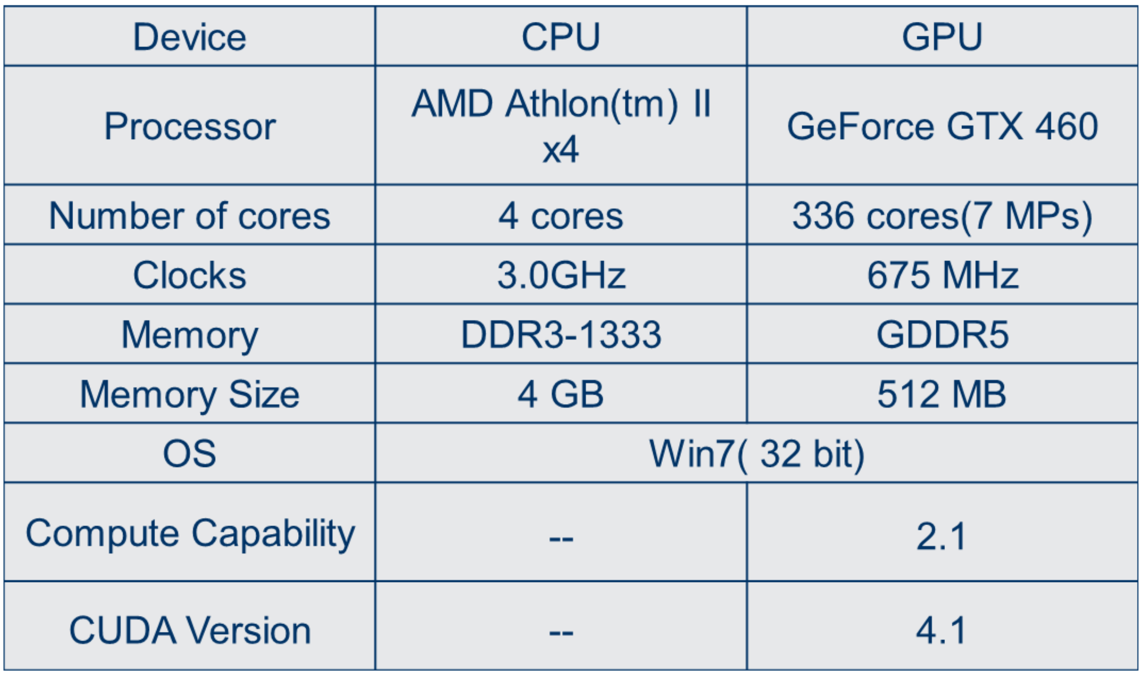
If colony ID is even, then exchange with colony (ID-2) % number of colonies per block

The two kinds of communication are executed alternately one after one iteration.

Chapter 4 Experimental Results and Analysis

4.1 Evaluation Environment

We adopt AMD Athlon (tm) II and GeForce GTX 460 for our computation platform. The configuration information is described as following. The host is AMD Athlon(tm) II which has 4 cores, and each core has clock rate with 3.0GHz. The device is GeForce GTX 460 which has 7 multiprocessors (MPs) and each MP has 48 CUDA cores. Totally, there are 336 CUDA cores in the device. Table 4-1 shows the experiment environment.



Device	CPU	GPU
Processor	AMD Athlon(tm) II x4	GeForce GTX 460
Number of cores	4 cores	336 cores(7 MPs)
Clocks	3.0GHz	675 MHz
Memory	DDR3-1333	GDDR5
Memory Size	4 GB	512 MB
OS	Win7(32 bit)	
Compute Capability	--	2.1
CUDA Version	--	4.1

Table 4 - 1 Hardware configurations

4.2 Benchmark Functions

Table 4-2 shows the equations of 9 continuous function minimization benchmarks. The equations are given together with the range of the variables and global minimum. These functions are widely used multi-modal test functions. The definitions and surface graphs of the 9 functions are shown as following :

Ackley function:

$$f(x_1, x_2) = 20 - 20e^{-0.2\sqrt{\frac{1}{2}({x_1}^2+{x_2}^2)}} - e^{\frac{1}{2}[\cos(2\pi x_1)+\cos(2\pi x_2)]} + e\ , -32 < x_i < 32$$

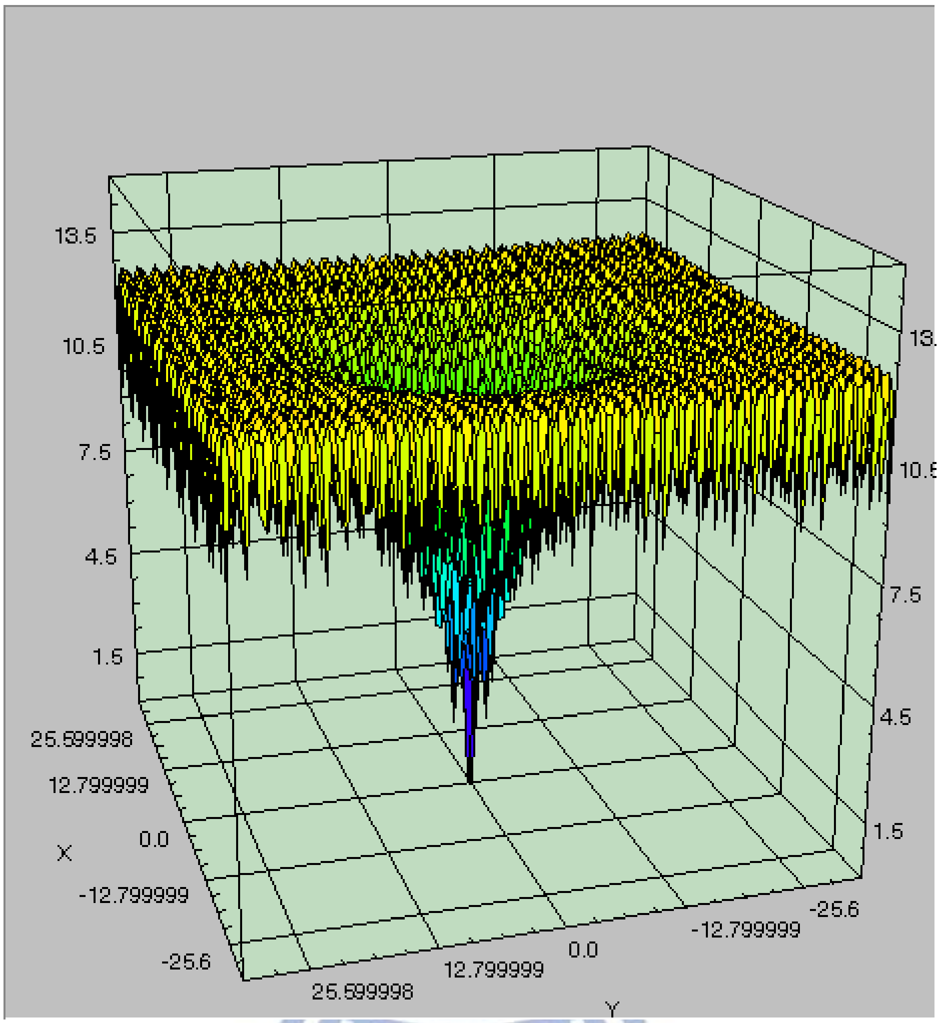


Figure 4 - 1 The surface plot of Ackley' function

Easom function:

$$f(x_1, x_2) = -\cos(x_1) * \cos(x_2)\, e^{-(x_1-\pi)^2 - (x_2-\pi)^2}, -100 < x_i < 100$$

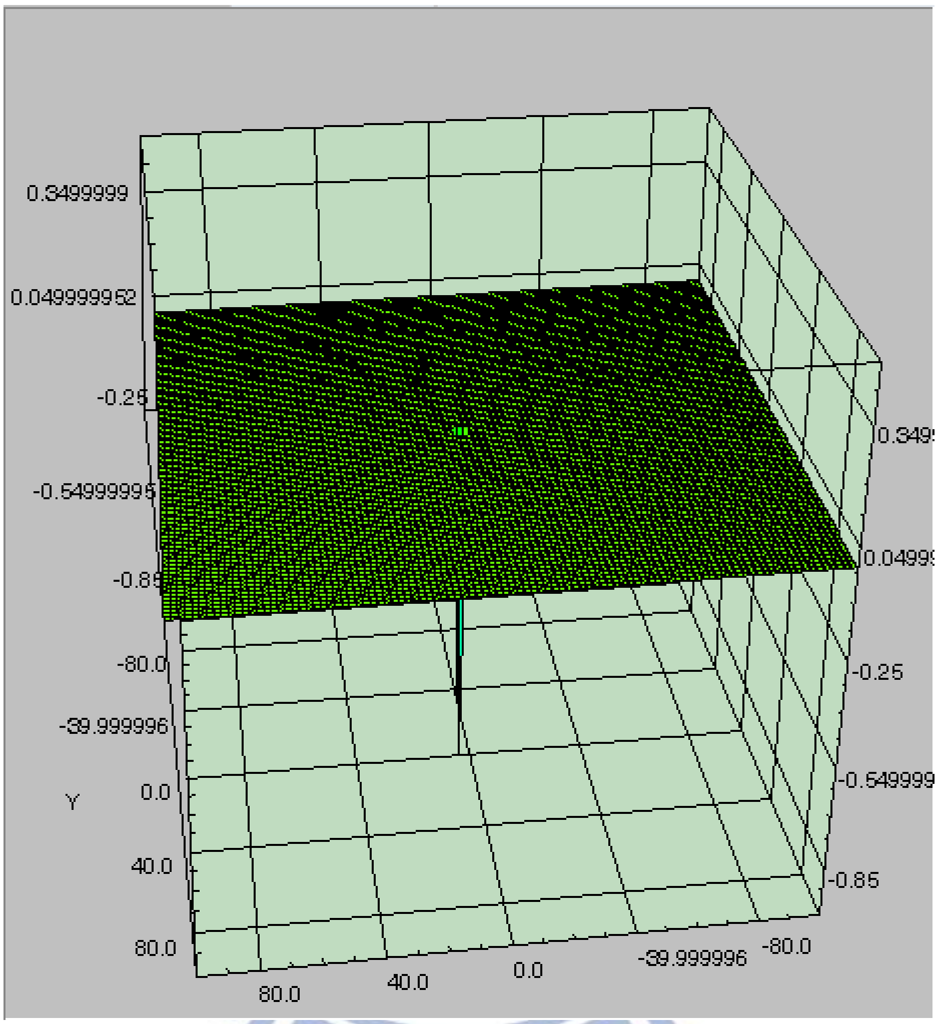


Figure 4 - 2 The surface Easom function

Goldstein and Price function

$\mathrm{A}(x_1, x_2) = 1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3{x_1}^2 - 14x_2 + 6x_1x_2 + 3{x_2}^2)$

$\mathrm{B}(x_1, x_2) = 30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12{x_1}^2 + 48x_2 - 36x_1x_2 + 27{x_2}^2)$

$f(x_1, x_2) = AB, -2 < x_i < 2$

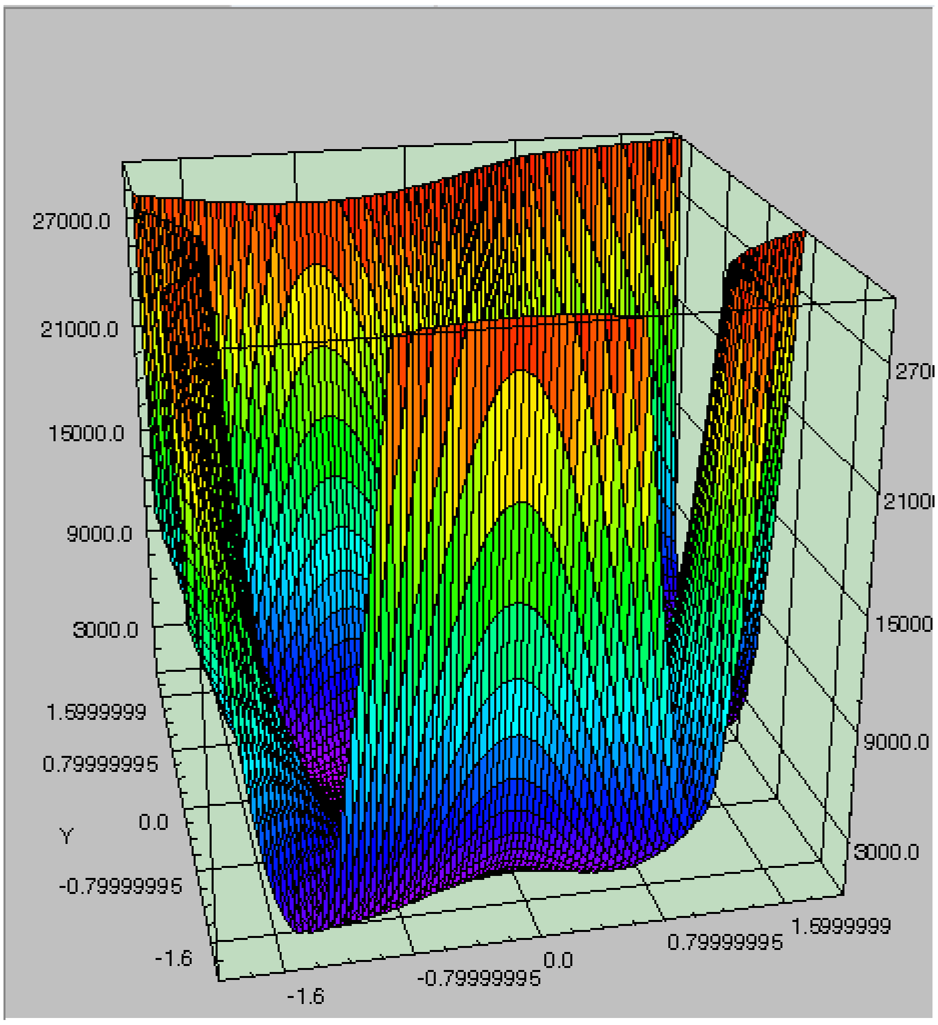


Figure 4 - 3 The surface Gold and Price function

Martin and Gaddy function

$$f(x_1, x_2) = (x_1 - x_2)^2 + [\frac{(x_1 + x_2 - 10)}{3}]^2, -20 < x_i < 20$$

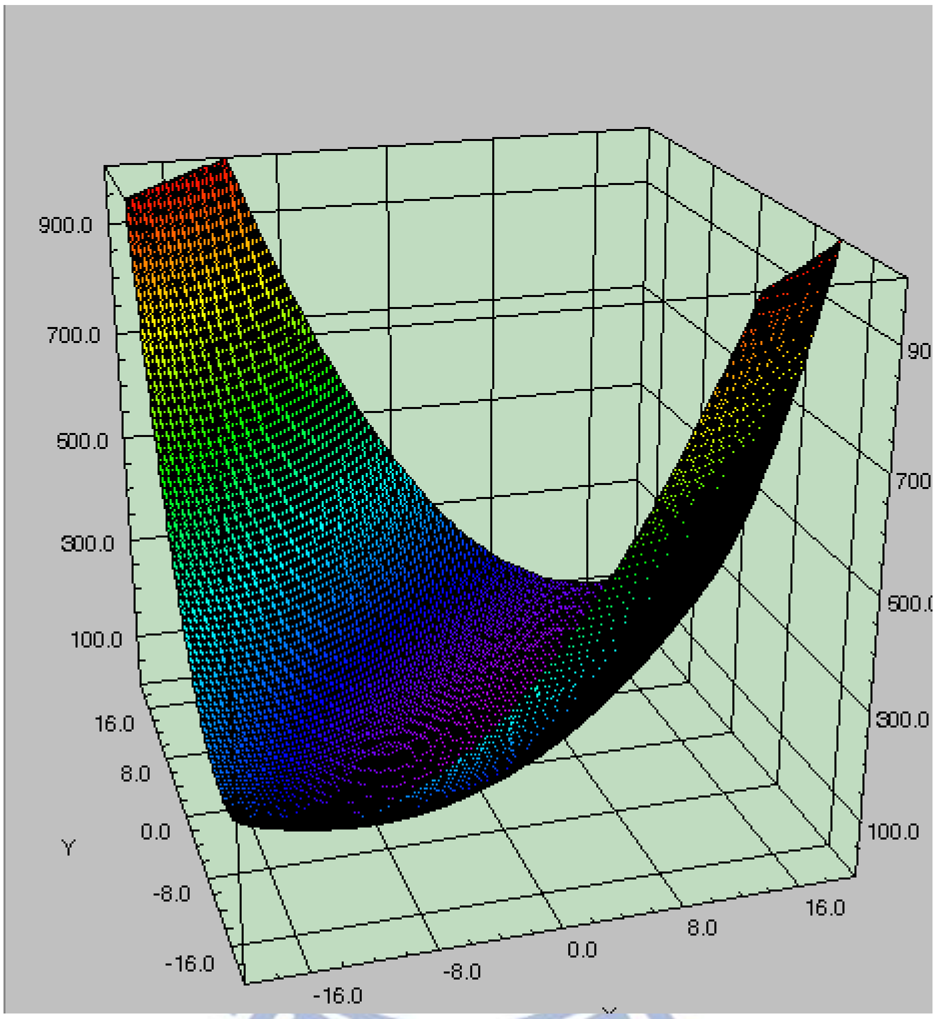


Figure 4 - 4 The surface Martin and Gaddy function

Schaffer function

$$f(x_1, x_2) = 0.5 + \frac{[\sin\left(\sqrt{{x_1}^2+{x_2}^2}\right)]^2 - 0.5}{[1.0 + 0.001({x_1}^2+{x_1}^2)]^2}, -100 < x_i < 100$$

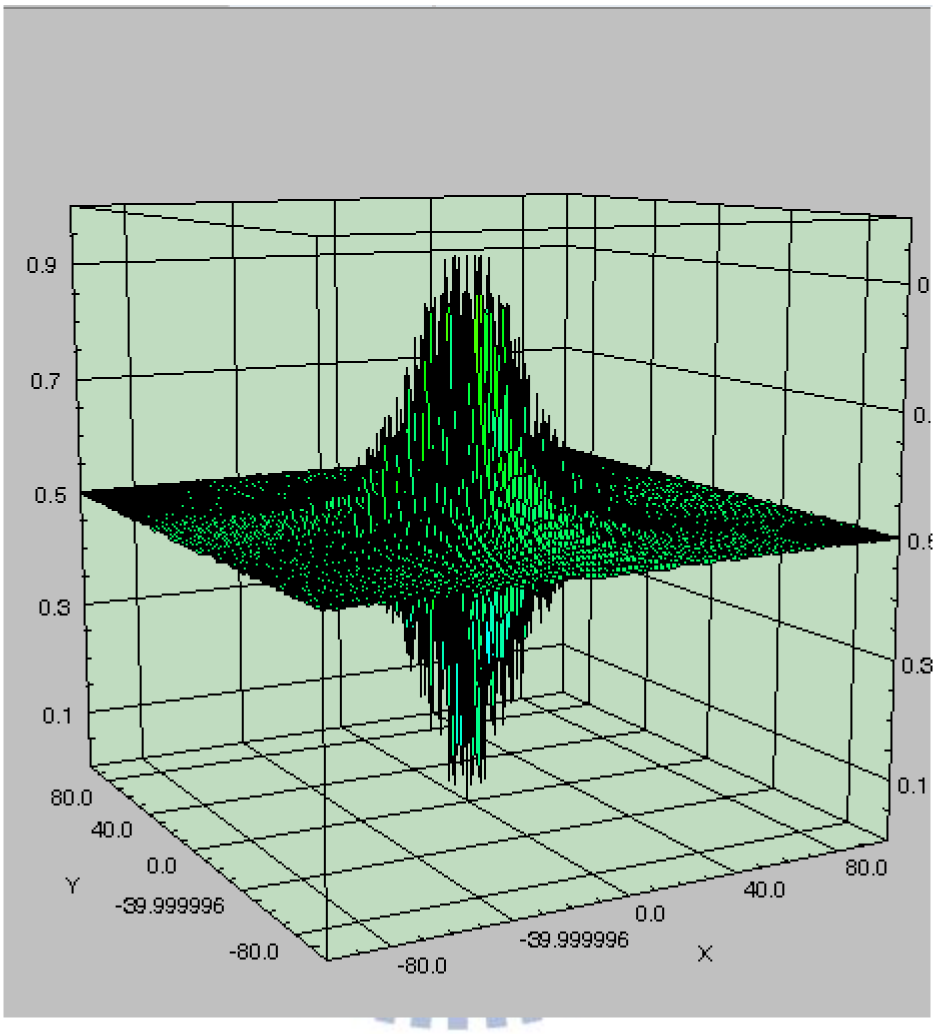


Figure 4 - 5 The surface Schaffer function

Schwefel function

$f(x_1, x_2) = -x_1 \sin(\sqrt{|x_1|} - -x_2 \sin\left(\sqrt{|x_2|}\right), -500 < x_i < 500$

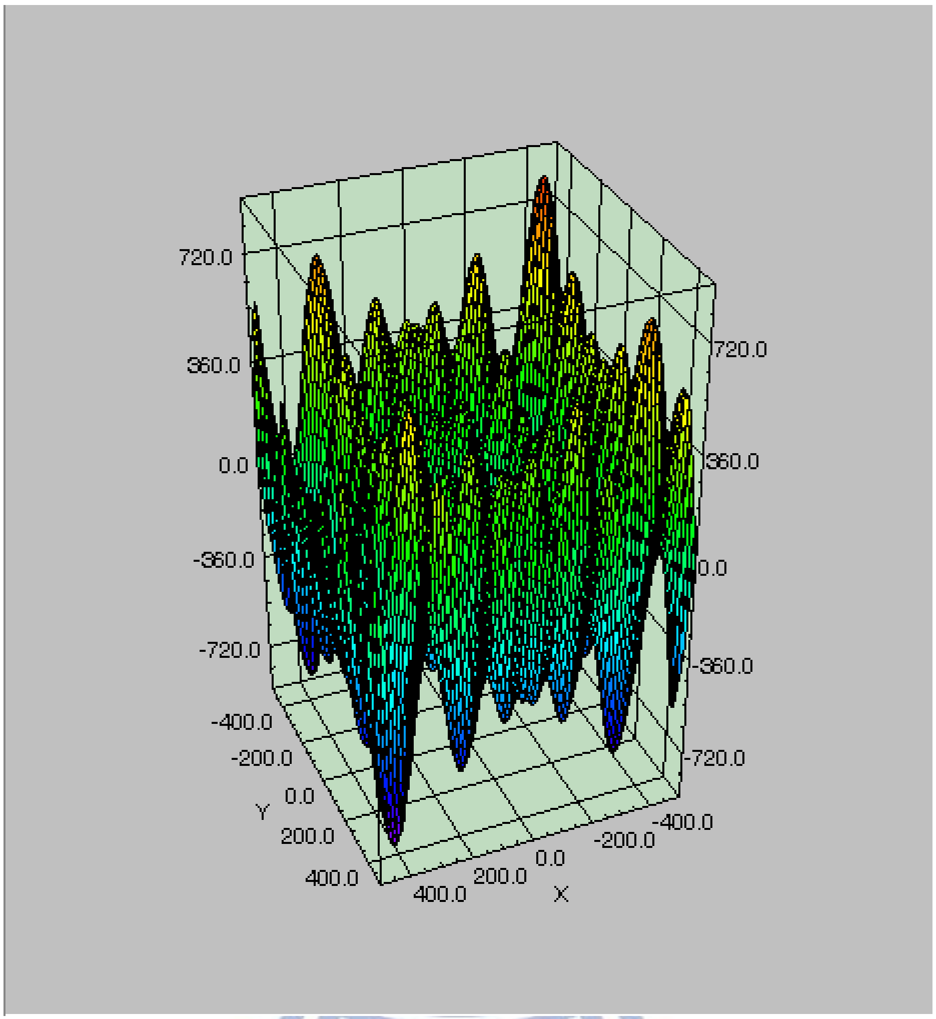


Figure 4 - 6 Schewefel function

Hyper Sphere

$$f(\vec{x}) = \sum_{i=1}^{10} {x_i}^2, -100 < x_i < 100$$

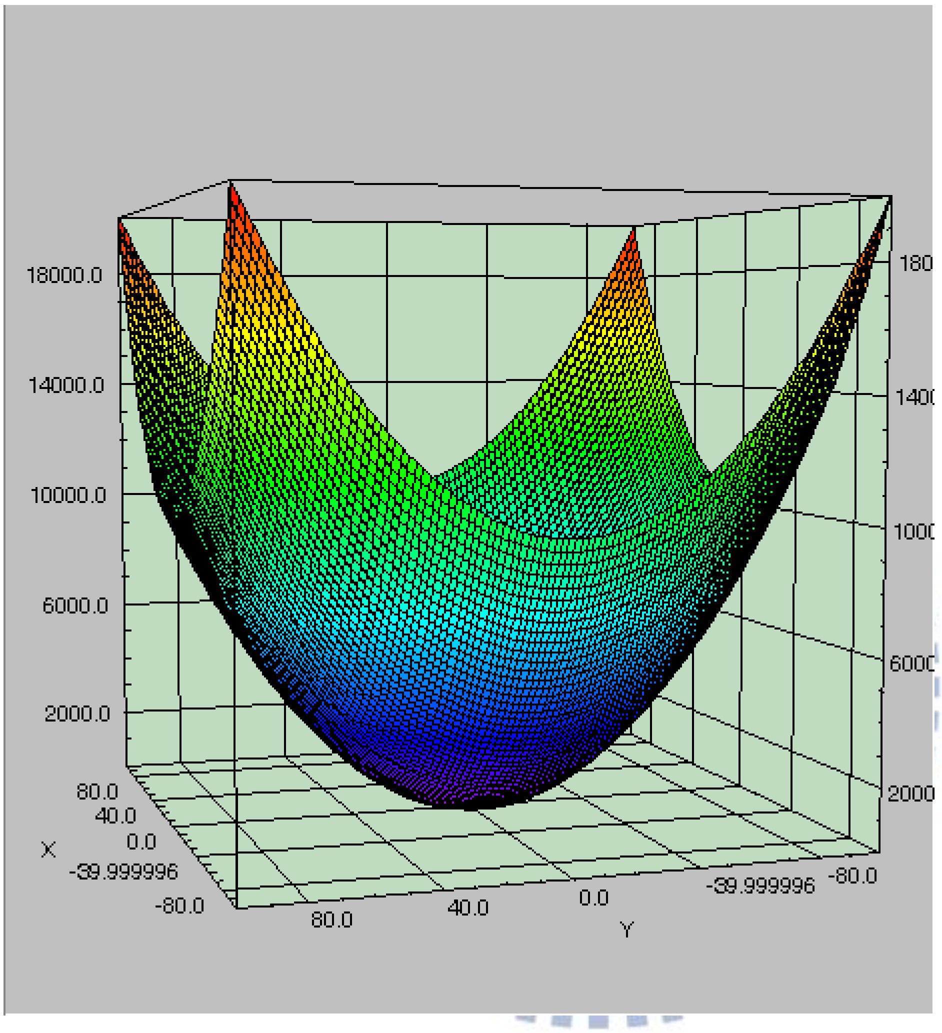


Figure 4 - 7 The surface Hyper Sphere function

Griewank

$$f(\vec{x}) = \frac{1}{4000}\sum_{i=1}^{10}(x_i - 100)^2 - \prod_{i=1}^{10}\cos\left(\frac{x_i - 100}{\sqrt{i+1}}\right) + 1, -600 < x_i < 600$$

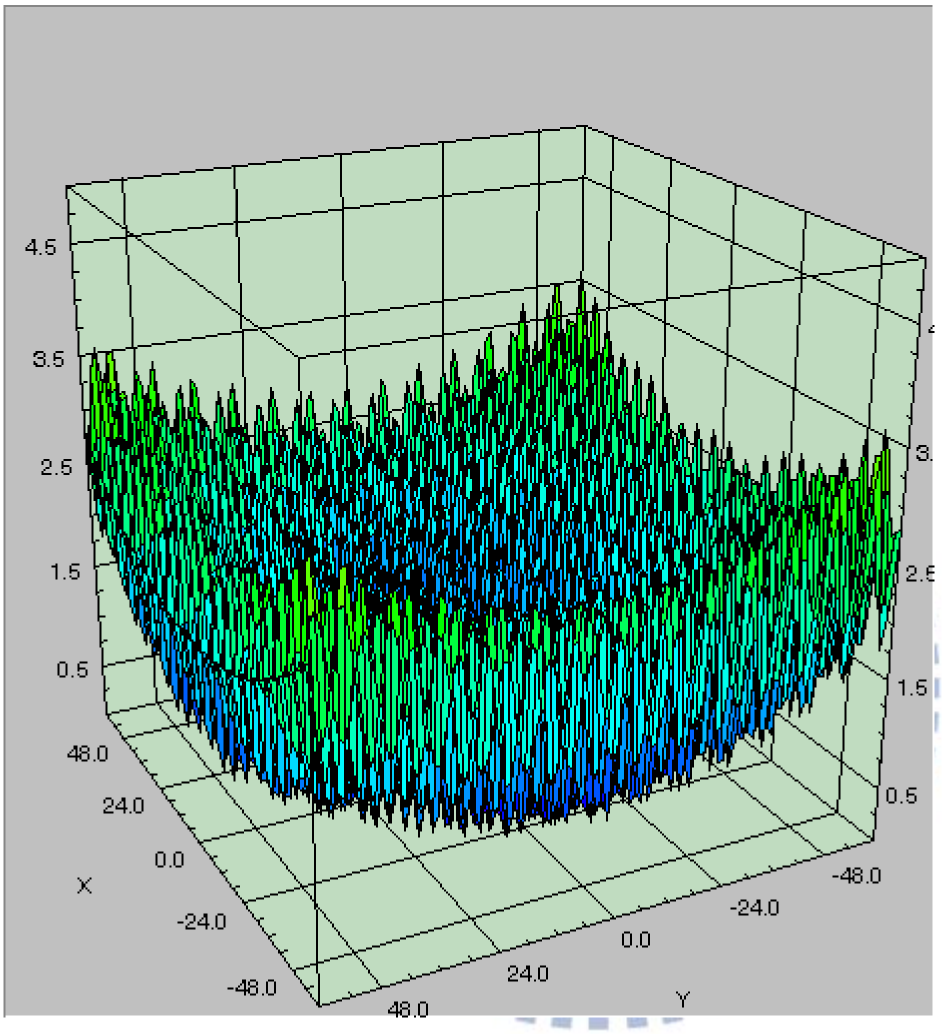


Figure 4 - 8 Griewank function

Rosenbrock function

$$f(\vec{x}) = \sum_{i=1}^{9} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2, \ -50 < x_i < 50$$

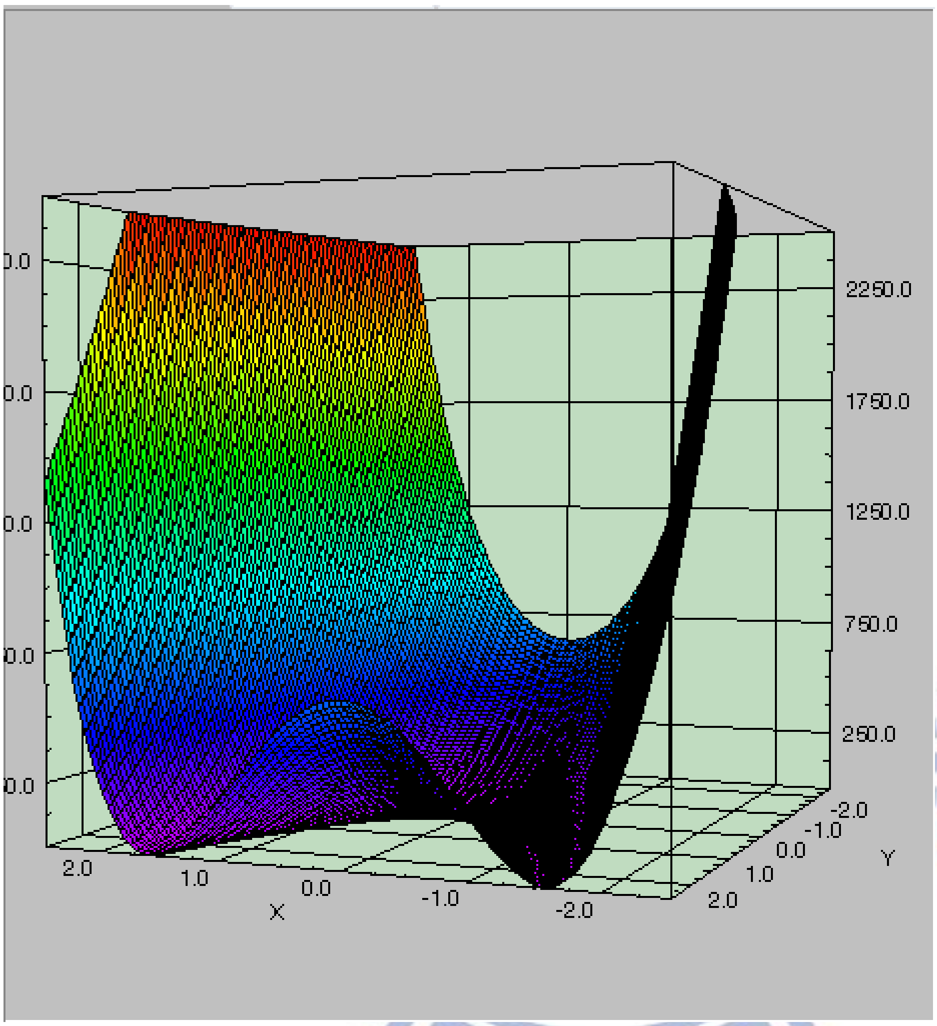


Figure 4 - 9 Rosenbrock function

Function	Equation	Minimum
Ackley(2D)	$f(x_1,x_2) = 20 - 20e^{-0.2\sqrt{\frac{1}{2}(x_1{}^2+x_2{}^2)}} - e^{\frac{1}{2}[\cos(2\pi x_1)+\cos(2\pi x_2)]} + e\,, -32 < x_i < 32$	$\vec{x}= (\vec{0})$ $\mathrm{f}(\vec{x}) = 0$
Easom(2D)	$f(x_1,x_2) = -\cos(x_1) * \cos(x_2)\, e^{-(x_1-\pi)^2 - (x_2-\pi)^2}, -100 < x_i < 100$	$\vec{x}= (\pi,\pi)$ $\mathrm{f}(\vec{x}) = -1$
Goldstein and Price(2D)	$\mathrm{A}(x_1,x_2) = 1 + (x_1+x_2+1)^2(19 - 14x_1 + 3x_1{}^2 - 14x_2 + 6x_1x_2 + 3x_2{}^2)$ $\mathrm{B}(x_1,x_2) = 30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12x_1{}^2 + 48x_2 - 36x_1x_2 + 27x_2{}^2)$ $f(x_1,x_2) = AB, -2 < x_i < 2$	$\vec{x}= (0,-1)$ $\mathrm{f}(\vec{x}) = 3$
Martin and Gaddy(2D)	$f(x_1,x_2) = (x_1 - x_2)^2 + [\frac{(x_1 + x_2 - 10\,)}{3}]^2, -20 < x_i < 20$	$\vec{x}= (5,5)$ $\mathrm{f}(\vec{x}) = 0$
Schaffer(2D)	$f(x_1,x_2) = 0.5 + \frac{[\sin\left(\sqrt{x_1{}^2+x_2{}^2}\right)]^2 - 0.5}{[1.0 + 0.001(x_1{}^2+x_1{}^2)]^2}, -100 < x_i < 100$	$\vec{x}= (\vec{0})$ $\mathrm{f}(\vec{x}) = 0$
Schwefel(2D)	$f(x_1,x_2) = -x_1\sin(\sqrt{\lvert x_1\rvert} - -x_2\sin\left(\sqrt{\lvert x_2\rvert}\right), -500 < x_i < 500$	$\vec{x}= (\overrightarrow{420.97})$ $\mathrm{f}(\vec{x}) = -837.97$
Hyper Sphere(10D)	$f(\vec{x}) = \sum_{i=1}^{10} x_i{}^2, -100 < x_i < 100$	$\vec{x}= (\vec{0})$ $\mathrm{f}(\vec{x}) = 0$
Griewank(10D)	$f(\vec{x}) = \frac{1}{4000}\sum_{i=1}^{10}(x_i - 100)^2 - \prod_{i=1}^{10}\cos\left(\frac{x_i - 100}{\sqrt{i+1}}\right) + 1, -600 < x_i < 600$	$\vec{x}= (\overrightarrow{100})$ $\mathrm{f}(\vec{x}) = 0$
Rosenbrock(10D)	$f(\vec{x}) = \sum_{i=1}^{9} 100(x_{i+1} - x_i{}^2)^2 + (1 - x_i)^2,\ -50 < x_i < 50$	$\vec{x}= (\vec{0})$ $\mathrm{f}(\vec{x}) = 0$

Table 4 - 2 Benchmark Functions

4.3 Analysis and Result

For CUBA, there are 5 parameters we have to set, GridDim, BlockDim, N, M and nep. In CUDA programming, the code running parallel is called “kernel”, the job size of kernel is so called “grid”. The programmer should set a dimensional number of grid. CUDA will divide the job to many smaller jobs and distribute them to different multiprocessors to execute. The size of each smaller job is called “block”. As setting dimensional number of grid, the programmer has to set the dimensional number of block, meaning how many threads in a block. In our algorithm, there are BlockDim / N colonies per block. For example if we set BlockDim = 256 and N = 8, then there are 32 colonies in a block. The parameters are set according to the results of the experiments in the next chapter, we will discuss in more detail later.

All the programs both of BA and CUBA in the following experiments were run until either the minimum of the function was approximated to better than 0.001, or reached a maximum number of cycles (here we set 5000).In BA, because there is only one colony foraging, if it make a wrong ngh shrinking, the global optimum solution will never be found. To overcome this, BA set a quite large nep and nsp to avoid as possible. Ideally, CUBA has more colonies foraged parallel in the same time, so we can afford more risks that we making a wrong ngh shrinking procedure. To prove this assumption, we test the 9 functions with various nep, 1, 2, 4, 8, 16 and 32. At first we set GridDim=4, BlockDim=256. It is a reasonable number of BlockDim. In most of GPUs, there are 32 or 64 stream processors (the smallest computation unit) in a multiprocessors, so we take the number as multiple of number of stream processors.

When we found adaptive nep for each function, we tried to decrease the number of BlockDim, that means we decreased the number of colonies. Another issue is what will happen if we increase the N and BlockDim with the same factor, in other words, we increase the bees for every colony and fix the number of colonies in a block. Finally, we will use the best parameters set we found from the three experiments, and take the result compare with The Bees Algorithm.

4.3.1 Analysis of nep

The result shows in Table 4-3 and Table 4-4. For low dimensional functions, we only need very small nep to get a good solution with less time. But for high dimensional functions, we need bigger nep, too small nep will lead the solution converge to the number with big error.

Benchmark Functions	nep=1	nep=2	nep=4	nep=8	nep=16	nep=32
Ackley	8.13ms	7.51ms	9.50ms	14.28ms	19.87ms	42.35ms
Easom	6.17ms	6.81ms	8.03ms	14.28ms	15.04ms	25.07ms
Goldstein and Price	6.01ms	5.56ms	7.57ms	9.14ms	12.92ms	18.49ms
Martin and Gaddy	4.58ms	4.72ms	4.92ms	5.67ms	5.52ms	5.74ms
Schaffer	7.11ms	7.27ms	9.24ms	11.81ms	16.20ms	33.28ms
Schwefel	5.52ms	6.35ms	6.48ms	8.33ms	11.18ms	16.20ms
HyperSphere	x	x	12ms	17.12ms	32.95ms	63.10ms
Griewank	x	52.51ms	70.95ms	125.23ms	219.76ms	439.90ms
Rosenbrock	x	x	x	1796.32ms	3140.61ms	4857.48ms

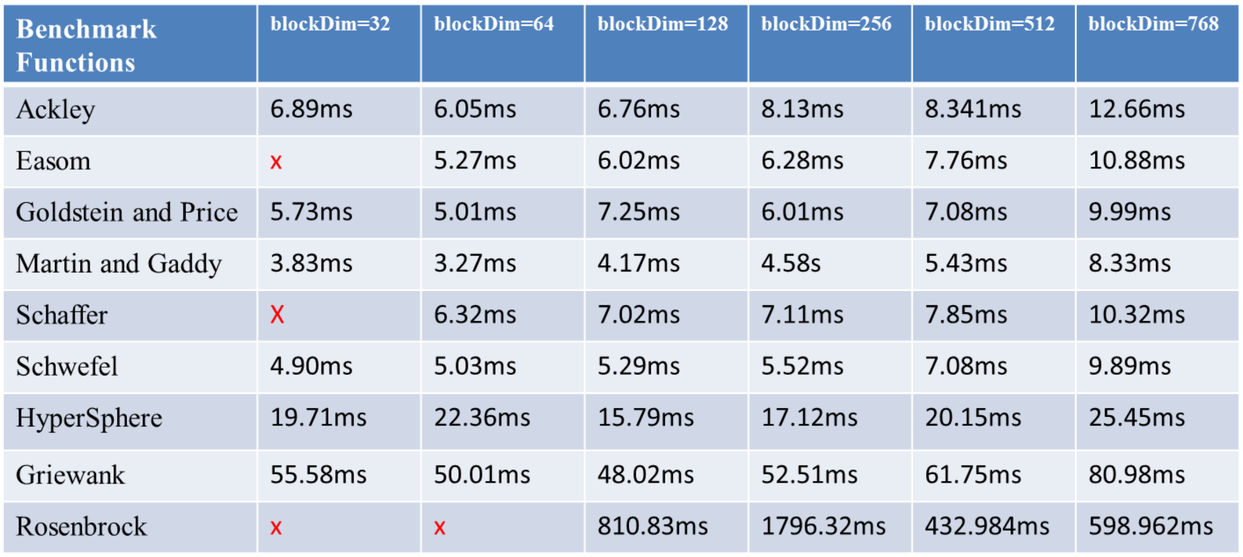
Table 4 - 3 nep increasing (time)

Benchmark Functions	nep=1	nep=2	nep=4	nep=8	nep=16	nep=32
Ackley	38	25	24	26	22	26
Easom	27	25	23	26	20	20
Goldstein and Price	22	12	17	14	13	11
Martin and Gaddy	17	16	14	15	8	5
Schaffer	29	22	22	19	16	20
Schwefel	27	30	22	23	21	19
HyperSphere	x	x	41	42	52	59
Griewank	x	51	43	44	42	44
Rosenbrock	x	x	x	439	410	329

Table 4 - 4 nep increasing (iterations)

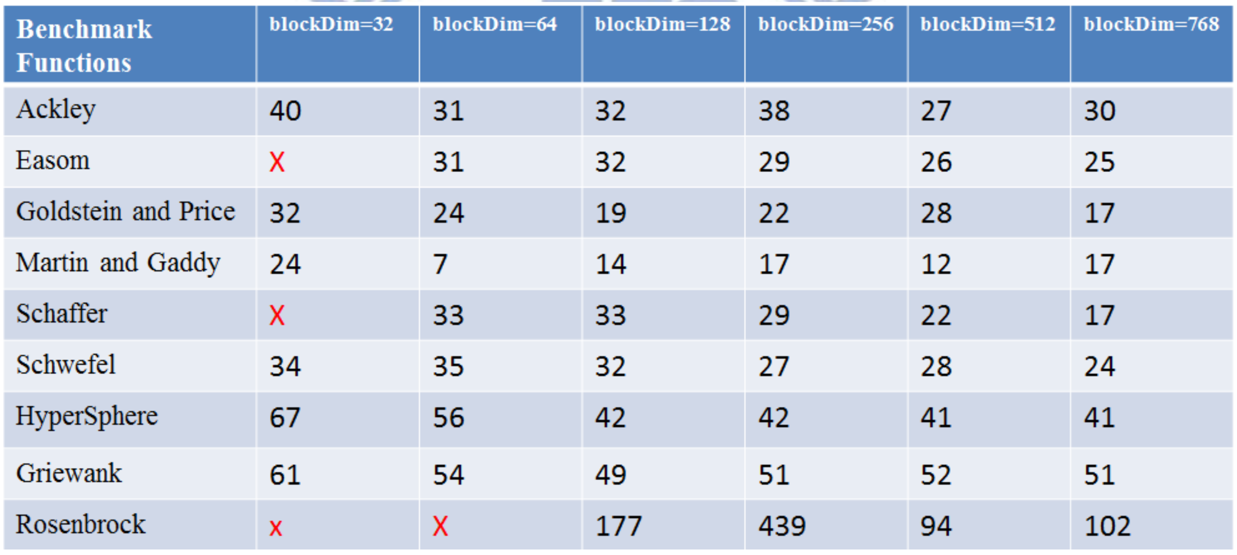
4.3.2 Analysis of the number of colonies

The result shows in Table 4-5 and Table 4-6, most of the functions get good performance and fewer execution time with small number of BlockDim, excluding the three high dimensional functions. For these high dimensional function, small number of BlockDim not always bring the benefit, the best number of BlockDim for HyperSphere and Griewank are 128, and 512 for Rosenbrock.



Benchmark Functions	blockDim=32	blockDim=64	blockDim=128	blockDim=256	blockDim=512	blockDim=768
Ackley	6.89ms	6.05ms	6.76ms	8.13ms	8.341ms	12.66ms
Easom	x	5.27ms	6.02ms	6.28ms	7.76ms	10.88ms
Goldstein and Price	5.73ms	5.01ms	7.25ms	6.01ms	7.08ms	9.99ms
Martin and Gaddy	3.83ms	3.27ms	4.17ms	4.58s	5.43ms	8.33ms
Schaffer	X	6.32ms	7.02ms	7.11ms	7.85ms	10.32ms
Schwefel	4.90ms	5.03ms	5.29ms	5.52ms	7.08ms	9.89ms
HyperSphere	19.71ms	22.36ms	15.79ms	17.12ms	20.15ms	25.45ms
Griewank	55.58ms	50.01ms	48.02ms	52.51ms	61.75ms	80.98ms
Rosenbrock	x	x	810.83ms	1796.32ms	432.984ms	598.962ms

Table 4 - 5 colonies increasing (time)



Benchmark Functions	blockDim=32	blockDim=64	blockDim=128	blockDim=256	blockDim=512	blockDim=768
Ackley	40	31	32	38	27	30
Easom	X	31	32	29	26	25
Goldstein and Price	32	24	19	22	28	17
Martin and Gaddy	24	7	14	17	12	17
Schaffer	X	33	33	29	22	17
Schwefel	34	35	32	27	28	24
HyperSphere	67	56	42	42	41	41
Griewank	61	54	49	51	52	51
Rosenbrock	x	X	177	439	94	102

Table 4 - 6 colonies increasing (iterations)

4.3.3 Analysis of the number of bees

Table 4-7 and Table 4-8 also show that we could set smaller number of BlockDim and N in low dimensional functions. In high dimensional function, for some functions, we could set a small bees number for shorter execution time like HyperSphere and Griewank. But sometimes the number of bees could not be too small, or the solution will converge with big error like Rosenbrock.

Benchmark Functions	blockDim=128 N=4	blockDim=256 N=8	blockDim=512 N=16
Ackley	6.76ms	8.13ms	9.68ms
Easom	5.72ms	6.28ms	7.87ms
Goldstein and Price	5.35ms	6.01ms	7.13ms
Martin and Gaddy	4.11ms	6.01ms	6.00ms
Schaffer	6.76ms	6.35ms	8.72ms
Schwefel	4.70ms	5.52ms	7.61ms
HyperSphere	15.88ms	17.12ms	25.12ms
Griewank	53.62ms	52.51ms	65.10ms
Rosenbrock	X	1796.32ms	2166.75ms

Table 4 - 7 bees increasing (time)

Benchmark Functions	blockDim=128 N=4	blockDim=256 N=8	blockDim=512 N=16
Ackley	36	38	33
Easom	31	29	26
Goldstein and Price	23	22	17
Martin and Gaddy	22	22	19
Schaffer	18	23	26
Schwefel	26	27	30
HyperSphere	46	42	49
Griewank	57	51	53
Rosenbrock	X	439	463

Table 4 - 8 bees increasing (iterations)

4.3.4 Robustness and Speedup

We calculated the execution times and the successful rates of fifty running times for the two algorithms. For estimating the execution time of BA, the parameters for all benchmark functions are given in table 4-9 according to the original set in the paper [6], and the table 4-10 shows the parameters set of CUBA by using the best parameters set we have found before.

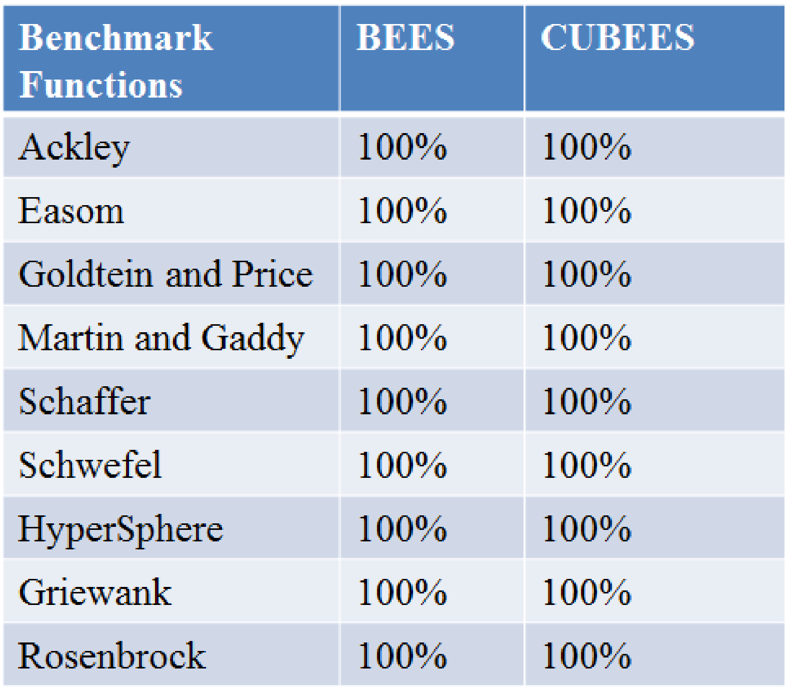
Benchmark	n	m	e	nep	nsp	stlim
Ackley	30	8	1	20	10	5
Easom	20	14	1	30	5	10
GoldsteinAndPrice	10	4	2	30	10	10
MartinAndGaddy	10	7	1	30	10	10
Schaffer	10	4	2	30	10	10
Schwefel	20	14	1	30	5	10
HyperSphere	10	4	2	30	10	10
Griewank	20	18	1	10	5	5
Rosenbrock	10	4	2	30	10	10

Table 4 - 9 Combinations of Bees Algorithm parameters

Benchmark	GridDim	BlockDim	n	m	nep
Ackley	4	64	8	6	1
Easom	4	64	8	6	1
GoldsteinAndPrice	4	64	8	6	1
MartinAndGaddy	4	64	8	6	1
Schaffer	4	64	8	6	1
Schwefel	4	32	8	6	1
HyperSphere	4	128	8	6	8
Griewank	4	128	8	6	2
Rosenbrock	4	512	8	6	8

Table 4 - 10 Combinations of CUDA Bees Algorithm parameters

As the result in Table 4-11 We found the solutions of those functions within the error, and getting 100% successful rates by using both of the algorithms in 50 times, The Bees Algorithm and CUBA. Finally, we compared the execution times of the functions and evaluated the speedup. The result shows that CUBA has 1x ~5x times faster than BA in different functions. CUDA supports fast math library and encourages programmers to use them as often as possible. The Griewank function with more trigonometric functions than others may bring the peak performance, because we could call more CUDA fast math libraries on it. The speed test results are shown in Table 4-12, and the time units are given in milliseconds. Not only the execution time of CUBA less than BA, but also less iterations to execute. As the result in Table 4-13, CUBA takes less iterations to find the solutions. It makes huger difference while running the high dimensional functions.



Benchmark Functions	BEES	CUBEES
Ackley	100%	100%
Easom	100%	100%
Goldtein and Price	100%	100%
Martin and Gaddy	100%	100%
Schaffer	100%	100%
Schwefel	100%	100%
HyperSphere	100%	100%
Griewank	100%	100%
Rosenbrock	100%	100%

Table 4 - 11 The successful rate in 50 runs

Benchmark Functions	BEES	CUBEES	SPEEDUP
Ackley	273ms	6.05ms	45.12
Easom	70ms	5.27ms	13.28
Goldstein and Price	92ms	5.01ms	18.36
Martin and Gaddy	76ms	3.27ms	27.24
Schaffer	231ms	6.32ms	36.55
Schwefel	279ms	4.90ms	56.93
HyperSphere	389ms	15.79ms	24.63
Griewank	2520ms	48.02ms	52.47
Rosenbrock	5595ms	432.98ms	12.92

Table 4 - 12 Speedups

Benchmark Functions	BEES	CUBEES
Ackley	90	31
Easom	44	31
Goldstein and Price	49	24
Martin and Gaddy	50	7
Schaffer	55	33
Schwefel	141	34
HyperSphere	158	42
Griewank	1487	49
Rosenbrock	4456	94

Table 4 - 13 Successive iterations

Chapter 5 Conclusion and Future work

Using GPU to solve problems with high density computation normally brings remarkable improving of performance. Of course, these problems should be able to parallel. Many applications have already been accelerated by GPGPU. In this paper, we proposed first parallel Bees Algorithm base on CUDA.

We modify the local search procedure. Running in SIMT (Single Instruction Multiple Thread) hardware architecture, we merge the two parts of the local searching sites avoiding wasting the computing powers of GPU. For the same reason, we have no site abandonment procedure. Another difference is that we let the bees recruiting in different sites maintain own ngh, meaning they shrink independently. We sort the colonies in the same block individually by using Odd–Even Sorting algorithm to get the benefit of parallel. The communication mechanism between colonies in the same block is also important point to decrease the convergence time, in our algorithm, we choose two-phase communication for better result.

To find the features of this new algorithm, we modify the parameters, and getting the result of the most of low dimensional functions could be run with good performance by using small nep. That’s one of the key points why CUBA runs with faster convergence time than The Bees Algorithm. We also try to decrease the number of colonies in a CUDA block and decrease the number of bees per colony to optimize the parameters set for each functions. The result shows in section 4.3.

Finally, we compare the convergent time (error < 0.001) of CUDA Bees Algorithm with The Bees Algorithm. The experiment result shows CUBA performs at least 1x times faster than BA from 9 different functions of optimization problems.

In the future, we will compare the CUBA to other parallel swarm based algorithm, and try more parallel sorting algorithm and communication mechanism to optimize. Not only for solving the optimization problem, we would also test the performance of the proposed algorithm on real world applications. Today, cloud computing becomes more and more important and popular. There are some platforms like Hoopoe which provides GPU based cloud computing service. We would improve the proposed algorithm and testing in GPUs clusters environment.

Reference

[1] M. Dorigo, "Optimization, Learning and Natural Algorithms," *Ph.D. thesis, Politecnico di Milano, Italie*, 1992.

[2] D.T. Pham, E. Koc, A. Ghanbarzadeh, S. Otri, S.Rahim, M. Zaidi "The Bees Algorithm–a novel tool for complex optimisation problems," *Proceedings of the Second International Virtual Conference on Intelligent Production Machines and Systems,* pp. 454-461, 2006.

[3] J. Kennedy, R. Eberhart, "Particle Swarm Optimization," *Proceedings of IEEE International Conference on Neural Networks,* vol. IV, pp. 1942-1948, 1995.

[4] E. Bonabeau, M. Dorigo, G. Theraulaz, "Swarm Intelligence: from Natural to Artificial Systems.," *Oxford University Press, New York,* 1999.

[5] D. Karaboga, B. Basturk "A powerful and Efficient Algorithm for Numerical Function Optimization: Artificial Bee Colony (ABC) Algorithm," *Global Optimization,* vol. 39, pp. 459-171, 2007.

[6] D.T. Pham, M. Castellani "The Bees Algorithm: modelling foraging behaviour to solve continuous optimization problems," *Proc Inst Mech Eng, C: J Mech Eng Sci,* vol. 223, pp. 2919-2938, 2009.

[7] D.T. Pham, S. Otri, A. Afify, M. Mahmuddin, H. Al-Jabbouli, "Data clustering using the Bees Algorithm," *Proceedings of the 40th CIRP International Manufacturing Systems Seminar,* 2007.

[8] D.T. Pham, A.H. Darwish, E.E. Eldukhri, S. Otri, "Using the Bees Algorithm to tune a fuzzy logic controller for a robot gymnast," *Proceedings of International Conference on Manufacturing Automation,* pp. 28-30, 2007.

[9] K. Guney, M. Onay "Bees Algorithm for design of dual-beam linear antenna arrays with digital attenuators and digital phase shifters," *Int J RF Microwave Comput Aided Eng,* vol. 18, pp. 337-347, 2008.

[10] D. T. Pham, E. Koc, J. Y. Lee, J. Phrueksanant "Using the Bees Algorithm to schedule jobs for a machine," *Proc Eighth International Conference on Laser Metrology, CMM and Machine Tool Performance, LAMDAMAP, Euspen,* pp. 430-439, 2007.

[11] *NVIDIA CUDA Programming Guide Version 4.2*: NVIDIA Corporation, 2012.

[12] *NVIDIA CUDA Best Practices Guild, 4.2 edition*: NVIDIA Corporation, 2012.

[13] Hongtao Baia, Dantong OuYang, Ximing Lia, Lili Hea, Haihong Yua, "MAX-MIN Ant System on GPU with CUDA," *Innovative Computing, Information and Control (ICICIC), 2009 Fourth International Conference,* pp. 801-804, 2009.

[14] J. C. Weihang Zhu, "Parallel Ant Colony for Nonlinear Function Optimization with Graphics Hardware Acceleration," *Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics*, pp. 1803-1808 2009.

[15] J.M. Cecilia, M. Ujaldon, A. Nisbet, M. Amos, *2011 IEEE International Parallel & Distributed Processing Symposium*, pp. 339-346 2011.

[16] Jian-Ming Li, Xiao-Jing Wang, Rong-Sheng He, Zhong-Xian Chi "An Efficient Fine-grained Parallel Genetic Algorithm Based on GPU-Accelerated," *2007 IFIP International Conference on Network and Parallel Computing Workshops*, pp. 855-862, 2007.

[17] Petr Pospichal, Jiri Jaros, Josef Schwarz, "Parallel Genetic Algorithm on the CUDA Architecture," *APPLICATIONS OF EVOLUTIONARY COMPUTATION*, vol. 6024, pp. 442-451, 2010.

[18] P. S. Boyd, "Convex Optimization," *Cambridge University Press*, p. 129, 2004.

[19] Ausiello, Giorgio, et al., *Complexity and Approximation (Corrected ed.)*: Springer, 2003.

[20] Available: http://en.wikipedia.org/wiki/Optimization_problem

[21] A. K. R. Mohamad Idris, M.W. Mustafa "A Parallel Bees Algorithm for ATC Enhancement in Modern Electrical Network," *2010 Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation*, pp. 450-455, 2010.

[22] S. Lakshmivarahan, S. K. Dhall,, L. L. Miller , L. Alt Franz and C. Marshall, Yovits, ed., "Parallel Sorting Algorithms," *Advances in computers (Academic Press)*, vol. 23, pp. 295-351, 1984.

[23] M. Phillips. (2011). Available: http://homepages.ihug.co.nz/~aurora76/Malc/Sorting_Array.htm#Exchanging

Appendex

source code: cuba.h

```
/***************************************************/
//               Normal math set
/***************************************************/
#define M_PI 3.14159265358979323846
#define M_E  2.71828182845904523536

/***************************************************/
//               CUDA parameter set
/***************************************************/
#define GRID_SIZE 4
#define BLOCK_SIZE 128

/***************************************************/
//               CUBA parameter set
/***************************************************/
#define EXETIMES 10
#define FUNCTION Griewank
#define FUNCTION_NAME "Griewank"
#define MAX_CYCLE 50
#define GLOBAL_MIN 0
#define DIM 10
#define N 8 //number of scout bees
#define M 6 //number of sites selected out of n visited site
#define NEP 8 //number of bees recruited for best e sites
#define LB -600 //lower bound of the parameters.
#define UB 600 //upper bound of the parameters. lb and ub can be defined as arrays
for the problems of which parameters have different bounds
#define NGH UB - (LB)//neighbourhood size
#define COLONY_PER_BLOCK BLOCK_SIZE/N
#define THREADS_PER_COLONY
#define TOTAL_BEES GRID_SIZE*BLOCK_SIZE
#define RAND_OFFSET 0
#define NGH_UP 1
#define NGH_DOWN 0.8
```

source code: cuba.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <conio.h>
#include <curand_kernel.h>
#include "device_launch_parameters.h"
#include "cuda_runtime.h"
#include <windows.h>
#include <ctime>
#include <algorithm>
#include <cstring>
#include <iostream>
#include <fstream>
#include "cuba.h"

using namespace std;

float params[TOTAL_BEES*DIM];
float fitness[TOTAL_BEES];
float values[TOTAL_BEES];

__device__ float sphere(float* sol) {

    int j;
    float top=0;
    for(j=0;j<DIM;j++)
    {
        top=top+sol[j]*sol[j];
    }
    return top;

}
__device__ float Rosenbrock(float* sol) {
    int j;
    float top=0;
    for(j=0;j<DIM-1;j++) {
```

```
    top=top+100*pow((sol[j+1]-pow((sol[j]),(float)2)),(float)2)+pow((sol[j]-1),
(float)2);
    }
    return top;
}
__device__ float Rastrigin(float* sol) {
    int j;
    float top=0;

    for(j=0;j<DIM;j++)
    {
        top=top+(pow(sol[j],(float)2)-10*__cosf(2*M_PI*sol[j])+10);
    }
    return top;
}
__device__ float Griewank(float* sol) {
     int j;
     float top1,top2,top;
     top=0;
     top1=0;
     top2=1;
     for(j=0;j<DIM;j++)
     {
         top1=top1+pow((sol[j]),(float)2);
         top2=top2*__cosf((((sol[j])/sqrt((float)(j+1)))*M_PI)/180);
     }
     top=(1/(float)4000)*top1-top2+1;
     return top;
 }

__device__ float MartinAndGaddy(float* sol) {
    float top = 0;
    top = pow((sol[0]-sol[1]),2) + pow((sol[0]+sol[1]-10)/3, 2);
    return top;
 }

__device__ float Easom(float sol[DIM]) {
    float top = 0;
    float top1 = (sol[0]-M_PI)*(sol[0]-M_PI);
    float top2 = (sol[1]-M_PI)*(sol[1]-M_PI);
    top = (-1)*__cosf(sol[0])*__cosf(sol[1])*pow(float(M_E), - top1 - top2);
    return top;
}

__device__  float Ackley(float sol[DIM]) {
    float top = 0;
    float top1 = 0;
    float top2 = 0;
    for(int i=0; i<DIM; i++) {
        top1 += sol[i]*sol[i];
        top2 += __cosf(2*M_PI*sol[i]);
    }
    top1 = 20.0*exp(-0.02*sqrt(top1/DIM));
    top2 = exp(top2/DIM);
    top = 20.0 - top1 - top2 + exp(1.0);

    return top;
}

__device__ float GoldsteinAndPrice(float sol[DIM]) {
    float top1 = 1 + pow((sol[0] + sol[1] + 1),
float(2))*(19-14*sol[0]+3*sol[0]*sol[0]-14*sol[1]+6*sol[0]*sol[1]+3*sol[1]*sol[
1]);
    float top2 = 30 + pow((2*sol[0] - 3*sol[1]),
float(2))*(18-32*sol[0]+12*sol[0]*sol[0] + 48*sol[1] - 36*sol[0]*sol[1] +
27*sol[1]*sol[1]);
    float top = top1*top2;
    return top;
}

__device__ float Schaffer(float sol[DIM]) {
    float top1 = pow(__sinf(sqrt(sol[0]*sol[0]+sol[1]*sol[1])), float(2));
    float top2 = pow(float(1.0+0.001*(sol[0]*sol[0] + sol[1]*sol[1])), float(2));
    float top = 0.5 + (top1 - 0.5)/top2;
    return top;
```

```
 }

__device__ float Schwefel(float sol[DIM]) {
    float top = (-1)*sol[0]*__sinf(sqrt(abs(sol[0]))) -
sol[1]*__sinf(sqrt(abs(sol[1])));
    return top;
 }

/***************************************************/
//          Calculate Fitness Function
/***************************************************/

__device__ float CalculateFitness(float fun) {
     float result=0;
     if(fun>=0)
     {
         result=1/(fun+1);
     }
     else
     {
         result=1+fabs(fun);
     }
     return result;
 }

/***************************************************/
//         Random initialize the point with given
//                index in global memory
/***************************************************/
__device__ void init(int idx, curandState_t* s, float* d_params, float* d_values,
float* d_fitness) {
    float r = curand_uniform(s);
    float solution[DIM];
    for(int j=0; j<DIM; j++) {
        r = curand_uniform(s);
        d_params[j*TOTAL_BEES+idx] = r * (UB - LB) + LB;
        solution[j] = d_params[j*TOTAL_BEES+idx];
    }
    d_values[idx] = FUNCTION(solution);
    d_fitness[idx] = CalculateFitness(d_values[idx]);
}

/**************************************************/
//          Random initialize the point with given
//                index in shared memory
/**************************************************/
__device__ void init2(int idx, curandState_t* s, float* s_params, float* s_values,
float* s_fitness) {
    float r = curand_uniform(s);
    float solution[DIM];
    for(int j=0; j<DIM; j++) {
        r = curand_uniform(s);
        s_params[j*BLOCK_SIZE+idx] = r * (UB - LB) + LB;
        solution[j] = s_params[j*BLOCK_SIZE+idx];
    }
    s_values[idx] = FUNCTION(solution);
    s_fitness[idx] = CalculateFitness(s_values[idx]);
}

/**************************************************/
//           Initial all bees positions
/**************************************************/
__global__ void initial(float* d_params, float* d_values, float* d_fitness) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int new_tb = TOTAL_BEES / (gridDim.x * blockDim.x); //the bees number of each
thread has
    int new_start = idx*new_tb;
    int new_end = (idx + 1)*new_tb;

    curandState_t state;//record the random sequence state
    unsigned long long seed = 0 + (unsigned long long) idx;
    curand_init(seed, RAND_OFFSET, 0, &state);

    for(int i=new_start; i<new_end; i++) {
        init(i, &state, d_params, d_values, d_fitness);
    }
```

```
}


__host__ __device__ void swap(float& a, float& b){
    float c=a;
    a=b;
    b=c;
}

/***************************************************/
//         Swap two positions in global memory
/***************************************************/
__host__ __device__ void mySwap(float* d_params, float* d_values, float* d_fitness,
int x, int y) {
     swap(d_values[x], d_values[y]);
     swap(d_fitness[x], d_fitness[y]);
     for(int j=0; j<DIM; j++) {
          swap(d_params[j*TOTAL_BEES+x], d_params[j*TOTAL_BEES+y]);
     }
}

/***************************************************/
//        Swap two positions in shared memory
/***************************************************/
__host__ __device__ void mySwap2(float* s_params, float* s_values, float* s_fitness,
int x, int y) {
     swap(s_values[x], s_values[y]);
     swap(s_fitness[x], s_fitness[y]);
     for(int j=0; j<DIM; j++) {
          swap(s_params[j*BLOCK_SIZE+x], s_params[j*BLOCK_SIZE+y]);
     }
}



__device__ void b_sort2(float* s_params, float* s_values, float* s_fitness, int
start, int end, int offset) {
     int len = end - start + 1;
    for(int i=0; i<len/offset; i++) {
        for(int j=start+1; j<end; j+=offset) {
            if(s_fitness[j] > s_fitness[j-1])
                mySwap2(s_params, s_values, s_fitness,  j-1, j);
        }
    }
}

/**************************************************/
//                    Nearbysearch
/**************************************************/
__device__ bool n_search(int idx, curandState* s, float ngh, float* s_params, float*
s_values, float* s_fitness) {
    float tmpParam[DIM];
    float tmpValue;
    float tmpFitness;
    for(int j=0; j<DIM; j++) {
        float r = curand_uniform(s);
        tmpParam[j] = 2*(r - 0.5)*ngh + s_params[j*BLOCK_SIZE+idx];
        if(tmpParam[j] > UB)
            tmpParam[j] = UB;
        if(tmpParam[j] < LB)
            tmpParam[j] = LB;
    }
    tmpValue = FUNCTION(tmpParam);
    tmpFitness = CalculateFitness(tmpValue);
    if(tmpFitness > s_fitness[idx]) {
        s_values[idx] = tmpValue;
        s_fitness[idx] = tmpFitness;
        for(int j=0; j<DIM; j++)
            s_params[j*BLOCK_SIZE+idx] = tmpParam[j];
        return true;
    }
    return false;
}
```

```
/**************************************************/
//               Replace data b from a
/**************************************************/
__device__ void replace(int a, int b, float* s_params, float* s_values, float*
s_fitness) {
    s_values[a] = s_values[b];
    s_fitness[a] = s_fitness[b];
    for(int j=0; j<DIM; j++)
        s_params[j*BLOCK_SIZE+a] = s_params[j*BLOCK_SIZE+b];
}


__host__ __device__ int findMax(float*values, int start, int end, int offset) {
    int maxIdx = start;
    for(int i=start+offset; i<end; i+=offset) {
        if(values[i] > values[maxIdx]){
                maxIdx = i;
        }
    }
    return maxIdx;
}

/**************************************************/
//               Ngh shrinking
/**************************************************/
__device__ float ngh_shrinking(float ngh, bool imp) {
    if(imp)
        return ngh*NGH_UP;
    else
        return ngh*NGH_DOWN;
}

/**************************************************/
//                  Run CUBA
/**************************************************/
__global__ void cubees(int m_cycle, int r_offset, float* d_params, float* d_values,
float* d_fitness) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int double_idx = threadIdx.x*2; //The index for odd-even sort
    int colonyNum = idx/N; //The colony which the bee belong to
    int colonyIdx = idx%N; //The colony ID which the bee with

    curandState_t state;//Record the random sequence state
    unsigned long long seed = (unsigned long long) idx;
    curand_init(seed, r_offset, seed, &state);

    bool improved = false;
    float ngh = NGH;

    //Claim shared memory for communication
    __shared__ float s_params[BLOCK_SIZE*DIM];
    __shared__ float s_values[BLOCK_SIZE];
    __shared__ float s_fitness[BLOCK_SIZE];

    //Initialize the shared memory
    s_values[threadIdx.x] = d_values[idx];
    s_fitness[threadIdx.x] = d_fitness[idx];
    for(int j=0; j<DIM; j++)
        s_params[j*BLOCK_SIZE+threadIdx.x] = d_params[j*BLOCK_SIZE+idx] ;

    //Odd-even sort
    __syncthreads;
    if(threadIdx.x < BLOCK_SIZE/2) {
        for(int l=0; l<=N/2; l++) {
            if(s_fitness[double_idx+1] > s_fitness[double_idx])
                mySwap2(s_params, s_values, s_fitness,  double_idx+1,
double_idx);
            if(colonyIdx < N-2) {
                if(s_fitness[double_idx+2] > s_fitness[double_idx+1])
                mySwap2(s_params, s_values, s_fitness,  double_idx+2,
double_idx+1);
            }
        }
    }
    __syncthreads;
    for(int c=0; c<m_cycle; c++) {
        improved = false;
        if(colonyIdx < M) {
            for(int i=0; i<NEP; i++) {
                if( n_search(threadIdx.x , &state, ngh, s_params, s_values,
s_fitness) ) {
                    improved = true;
                }
            }
        }
        else {
            init2(threadIdx.x, &state, s_params, s_values, s_fitness);
        }

        ngh = ngh_shrinking(ngh, improved);

        //Odd-even sort
        __syncthreads;
        if(threadIdx.x < BLOCK_SIZE/2) {
            for(int l=0; l<=N/2; l++) {
                if(s_fitness[double_idx+1] > s_fitness[double_idx])
                    mySwap2(s_params, s_values, s_fitness,  double_idx+1,
double_idx);
                if(colonyIdx < N-2) {
                    if(s_fitness[double_idx+2] > s_fitness[double_idx+1])
                    mySwap2(s_params, s_values, s_fitness,  double_idx+2,
double_idx+1);
                }
            }
        }

        __syncthreads;
        //Two phase communication
        if(colonyIdx == M-1){
            int target = (threadIdx.x + (c%2+1)*int(pow(-1, float(colonyNum)))*N
- M + 1 )%BLOCK_SIZE; //odd colony get the best from left,  even get the best from
right
```

```
                replace(threadIdx.x,  target, s_params, s_values, s_fitness);
            }
        }

        //Copy the data to global memory from shared memory
        __syncthreads;
        d_values[idx] = s_values[threadIdx.x];
        d_fitness[idx] = s_fitness[threadIdx.x];
        for(int j=0; j<DIM; j++)
            d_params[j*TOTAL_BEES+idx] = s_params[j*BLOCK_SIZE+threadIdx.x];

}

/****************************************************/
//      Initialize memory and call CUBA
/****************************************************/
float callCUBEES(int m_cycle, int r_offset, float* h_params, float* h_values, float*
h_fitness) {
    cudaEvent_t start, stop;
    float elapsedTime;
    float *dev_params = 0;
    float *dev_values = 0;
    float *dev_fitness = 0;
    cudaError_t cudaStatus;

    //Record algorithm execution time
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

    //Choose which GPU to run on, change this on a multi-GPU system.
   cudaStatus = cudaSetDevice(0);
   if (cudaStatus != cudaSuccess) {
       fprintf(stderr, "cudaSetDevice failed!  Do you have a CUDA-capable GPU
installed?");
       goto Error;
   }
    //Allocate GPU buffers for three vectors   .
    cudaStatus = cudaMalloc((void**)&dev_params, TOTAL_BEES * DIM * sizeof(float));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "dev_params cudaMalloc failed!");
        goto Error;
    }
     cudaStatus = cudaMalloc((void**)&dev_values, TOTAL_BEES * sizeof(float));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "dev_values cudaMalloc failed!");
        goto Error;
    }
    cudaStatus = cudaMalloc((void**)&dev_fitness, TOTAL_BEES * sizeof(float));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "dev_fitness cudaMalloc failed!");
        goto Error;
    }

     //Copy the host vectors to device vectors.
     cudaStatus = cudaMemcpy(dev_params, h_params, TOTAL_BEES * DIM * sizeof(float),
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "dev_foods cudaMemcpy failed!");
        goto Error;
    }
     cudaStatus = cudaMemcpy(dev_values, h_values, TOTAL_BEES * sizeof(float),
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "dev_values cudaMemcpy failed!");
        goto Error;
    }
     cudaStatus = cudaMemcpy(dev_fitness, h_fitness, TOTAL_BEES * sizeof(float),
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "dev_fitness cudaMemcpy failed!");
        goto Error;
    }

     //Call kernal functions
     initial<<<GRID_SIZE, BLOCK_SIZE>>>(dev_params, dev_values, dev_fitness);
    cubees<<<GRID_SIZE, BLOCK_SIZE>>>(m_cycle, r_offset, dev_params, dev_values,
dev_fitness);


   cudaStatus = cudaDeviceSynchronize();
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, "cudaDeviceSynchronize returned error code %d after
launching cuabc!\n", cudaStatus);
      goto Error;
   }

    // Copy the device vectors to host vectors    .
    cudaStatus = cudaMemcpy(h_params, dev_params, TOTAL_BEES * DIM * sizeof(float),
cudaMemcpyDeviceToHost);
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, "h_foods cudaMemcpy failed!");
      goto Error;
   }
    cudaStatus = cudaMemcpy(h_values, dev_values, TOTAL_BEES * sizeof(float),
cudaMemcpyDeviceToHost);
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, "h_values cudaMemcpy failed!");
      goto Error;
   }
    cudaStatus = cudaMemcpy(h_fitness, dev_fitness, TOTAL_BEES * sizeof(float),
cudaMemcpyDeviceToHost);
   if (cudaStatus != cudaSuccess) {
      fprintf(stderr, "h_values cudaMemcpy failed!");
      goto Error;
   }

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&elapsedTime, start, stop); // that's our time!
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    //Free device memories
    Error:
    cudaFree(dev_params);
    cudaFree(dev_values);
   cudaFree(dev_fitness);
    return elapsedTime;
}

int main(){
    float exec_time = 0;
    int maxID = 0;
    ofstream recordFile;
    exec_time = callCUBEES(MAX_CYCLE, RAND_OFFSET, params, values, fitness);
    maxID = findMax(fitness, 0, TOTAL_BEES, N);
    cout<<"value: "<<values[maxID]<<" execution time: " <<exec_time<<endl;
    system("Pause");
    return 0;
}
```