Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics
San Antonio, TX, USA - October 2009

Parallel Ant Colony for Nonlinear Function Optimization with Graphics Hardware Acceleration

Weihang Zhu
Department of Industrial Engineering
Lamar University
Beaumont, Texas, USA
Weihang.zhu@lamar.edu

James Curry
Department of Industrial Engineering
Lamar University
Beaumont, Texas, USA
jcurry@my.lamar.edu

***Abstract*— This paper presents a massively parallel Ant Colony Optimization – Pattern Search (ACO-PS) algorithm with graphics hardware acceleration on nonlinear function optimization problems. The objective of this study is to determine the effectiveness of using Graphics Processing Units (GPU) as a hardware platform for ACO-PS. GPU, the common graphics hardware found in modern personal computers, can be used for data-parallel computing in a desktop setting. In this research, the classical ACO is adapted in the data-parallel GPU computing platform featuring 'Single Instruction – Multiple Thread' (SIMT). The global optimal search of the ACO is enhanced by the classical local Pattern Search (PS) method. The hybrid ACO-PS method is implemented in a GPU+CPU hardware platform and compared to a similar implementation in a Central Processing Unit (CPU) platform. Computational results indicate that GPU-accelerated SIMT-ACO-PS method is orders of magnitude faster than the corresponding CPU implementation. The main contribution of this paper is the parallelization analysis and performance analysis of the hybrid ACO-PS with GPU acceleration.**

***Keywords*—Ant Colony, Pattern Search, Graphics Hardware Acceleration, GPU, CUDA**

I. INTRODUCTION

Ant Colony Optimization (ACO) is a stochastic, population-based evolutionary meta-heuristic algorithm inspired by the behavior of real ant colonies. It is a type of Swarm Intelligence which is based on social-psychological principle that individuals communicate and adjust their behavior based on the performance of others. The ACO algorithm was first introduced in 1992 by Marco Dorigo in his Ph.D. dissertation [2]. It is inspired by the behavior of ants in finding paths from the colony to food. ACO is computationally intensive when it comes to complex problems.

Graphics Processing Unit (GPU) has emerged as a possible *desktop parallel computing* solution for large scale scientific computation within a desktop PC setting [5]. In this project, we explored its ability and limitations with the ACO algorithm on a set of bound constrained optimization functions. In the local search phase, traditional Pattern Search (PS) method is used [7]. The objective of this research is to find out how and how much the ACO algorithm can be potentially accelerated on a GPU platform. The GPU-based SIMT-ACO-PS algorithms are implemented within the Compute Unified Device Architecture (CUDA) ™ environment, on an nVidia Graphics Processing Unit (GPU). We have demonstrated the potential of GPU technology in Tabu Search on the Quadratic Assignment Problem in a recent paper [9]. We have also designed a GPU-accelerated Particle Swarm Optimization and Pattern Search on function optimization problems [10][11].

The remainder of this paper is organized as follows. Section 2 presents background information on the GPU computing, Ant Colony Optimization, and Pattern Search. Section 3 provides an overview of the SIMT-ACO-PS algorithm. Section 4 discusses the parallelization analysis and implementation of the SIMT-ACO-PS on a GPU hardware platform. Section 5 presents computational experiment results and analysis. Conclusions of our investigation and future research tasks are summarized in Section 6.

II. RESEARCH BACKGROUND

A. GPU Computing

GPU parallel computing follows a different pattern, namely 'Single Instruction – Multiple Thread' (SIMT). With SIMT, a GPU executes the same instruction set on different data elements at the same time. A GPU can process thousands of threads simultaneously enabling high computational throughput across large amounts of data. The CUDA technology allows a software developer to program a GPU for general purpose computing in many possible applications [5].

In the CUDA environment, thousands of threads can run concurrently with a same instruction set. Each thread runs a same program called a 'kernel'. A kernel can employ 'registers' as fast access memory. The communication among threads can be realized with 'shared memory', which is a type of very fast memory that allows both read and write access. The communication between CPU and GPU can be done through global device memory, constant memory, or texture memory on a GPU board. Global device memory is a relatively slow memory location that allows both read and write operations. Texture memory is relatively fast memory that is read-only. We employ texture memory to keep a copy of the ant population, the Probability Vector (with information of Pheromone) and pre-generated random numbers. Constant memory is fast read-only memory whose size cannot be dynamically changed. Texture memory and constant memory is fast because it is cached for quicker access. The nVidia

978-1-4244-2794-9/09/$25.00 ©2009 IEEE

GeForce GTX 280 GPU hardware employed in this paper has 30 multi-processors. Each multi-processor has 8 processors. This amounts to 240 data-parallel processors (cores) on one GPU board. For management purpose, all threads are organized into blocks. Each 'block' can have a maximum of 512 threads. The threads in the same block can communicate through fast shared memory; while between the blocks, communication is possible only with slower global device memory. For a detailed description of the CUDA capabilities, readers are referred to the CUDA programming guide [6].

B. Parallel Ant Colony

First introduced in 1992 by Marco Dorigo [2], the Ant Colony Optimization (ACO) algorithm simulates the behavior of ants in finding paths from the colony to food. In reality, ants start with random movements, and upon finding food, they lay down pheromone trails as they return to their colony. If some ants find a path to food, they are likely to follow the trail with higher concentration of pheromone. Pheromone may evaporate over the time and thus the pheromone trail become less attractive. This pheromone evaporation can help avoid the convergence to local optima. After some iterations of the ant swarm, the fitness of the global best solution keeps improving. Eventually the swarm converges to the global optimal. There are many variants of the ACO since it has been used in different applications [3].

In this paper, we used ACO to solve bound constrained continuous function optimization problems. In our design, the parameter bound range is divided into equally spaced steps, say 1000 steps, as the length of the Pheromone vector τ. The ACO starts with a batch of randomly generated ants as the initial solutions. The Pheromone vector τ is assigned a very small value initially. It is typically decayed each time it is update as shown in (1) where ρ_g is the global pheromone decay constant.

$$\tau \leftarrow \tau(1-\rho_g)\,. \tag{1}$$

The local pheromone decay is not used. The k elite solutions are used to update the Pheromone vector τ with (2), where Q is a pheromone update constant value, C_k is the cost of the k-th elite solution and $\tau_{i(j,k)}$ is the Pheromone element mapping with the j-th variable of the k-th elite solution.

$$\tau_{i(j,k)} \leftarrow \tau_{i(j,k)} + \frac{Q}{C_k} \text{ if } C_k != 0\,,$$

$$\tau_{i(j,k)} \leftarrow \max\{\tau_i\}, \text{ if } C_k = 0\,, \tag{2}$$

Corresponding to the Pheromone vector τ, there is a Probability Vector p, whose element p_i is calculated with (3):

$$p_i = \frac{\tau_i^{\alpha}}{\sum_i \tau_i^{\alpha}}\,. \tag{3}$$

where α is the pheromone sensitivity constant. With the Probability Vector p, each ant constructs a candidate solution. During the solution construction process, if a random number is less than the exploration constant q_0, the Selection Index is chosen as the index of the Probability Vector with the maximum probability. Otherwise, a roulette wheel selection method as in the traditional ACO is used to choose the Selection Index using the Probability Vector. This Selection Index is then converted to the variable value of the ant solution. The ant solutions are evaluated and sorted from the best to the worst. A small portion of elite (best) solutions are retained from one generation to the next generation without change. The fitness of the global best solution keeps improving during the ACO evolution.

For complex problems, ACO requires significant computation time. Adding parallelism to ACO is one option for improvement. ACO is naturally amenable to parallelism. A detailed classification of parallel ACO can be found in [1]. This research explores a low level parallel implementation where all ants interact within a single population. In our ACO variant, individual ants are randomly initialized to start the search. These ants serve as candidate solutions. An ant solution vector is composed of elements coded in real numbers. Each element represents a dimension variable in a multi-dimensional bound constrained optimization problem. This model is specifically designed for the GPU parallel computing platform.

C. Parallel Pattern Search

While ACO algorithms are good at quickly finding a reasonable solution, they may be slow to converge to the local optimal solution from a nearby solution and they may be stuck in a local optimum. To overcome this problem, a local search improvement phase, such as Pattern Search (PS), can be added to the ACO.

The basic Pattern Search algorithm is a simple direct search method that does not require derivative or second derivative information. Pattern Search is traditionally employed when the gradient of the function is not reliable when performing the search [7]. This research uses Ant Colony Optimization (ACO) as the global search method and Pattern Search (PS) as the local search method. We defined thousands of ants executed in parallel on a maximum of 240 processors within a single GPU. Using ACO as the global search tool has the advantage of covering a wide amount of the search space. Each member of the population is assigned a separate Pattern Search. This approach does not require a modification to the basic Pattern Search algorithm since each thread is acting on a separate search.

III. Overview of the SIMT-ACO-PS Method

The proposed 'Single Instruction – Multiple Thread Ant Colony – Pattern Search' (SIMT-ACOPS) method has two components: Ant Colony Optimization (ACO) and Pattern Search (PS). The method is tailored for an environment with T ants (threads) that operate in a single instruction multiple thread manner. Each thread generates a new solution based on the Probability Vector and performs Pattern Search improvement to improve the new solution. The ants are sorted based on their costs to find the best solution after each generation. Let x_i

represent the set of solution parameters for individual ant *i*, $f(\vec{x}_i)$ is the solution cost of this ant, and $c(\vec{x}_i)$ is the bound constraint. The Ant Colony component is illustrated in Fig. 1.

A) Initialize T ants, each of which has P dimension variables;
B) Evaluate the cost and constraints of all the ants;
C) Sort the ants based on the cost;
D) Initialize ACO and PS parameters;
While (Not meeting terminate criterion)
E) Evaporate pheromone, with (1);
F) Use the best few ants to update the pheromone vector, with (2);
G) Update the Probability Vector with the pheromone information, with (3);
H) Generate the new ant solutions based on the Probability Vector;
I) Evaluate the cost $f(\vec{x}_i)$ and constraints $c(\vec{x}_i)$ of the ants;
J) Improve all the ants with Pattern Search → See *Fig. 2*;
K) Sort the ants based on the cost $f(\vec{x}_i)$;
L) Collect the generation result;
End While
M) Collect final result.

Figure 1. Procedure of the Ant Colony Component

a) Initialize PS parameters (search step $\Delta_i = \Delta_o$) and start from the ACO solution for the thread;
For (iteration $k = 1,2, \ldots, k_{max}$)
For (dimension $j = 1,2, \ldots, P$)
b) If $f(x_i + e_j^+\Delta_i) \le f(x_i)$ and

$f(x_i + e_j^+\Delta_i) \le f(x_i + e_i^-\Delta_i)$ then $x_i = x_i + e_j^+\Delta_i$.

c) Else If $f(x_i + e_j^-\Delta_i) \le f(x_i)$ and

$f(x_i + e_j^-\Delta_i) \le f(x_i + e_i^+\Delta_i)$ then $x_i = x_i + e_j^-\Delta_i$.

End For (each dimension)
d) If no improvement throughout all *d* dimensions, $\Delta_i = \Delta_i / 2$; If $\Delta_i = \Delta_{tol}$, reset $\Delta_i = \Delta_o$;
End For (k iterations)
e) Collect final result of this Pattern Search

Figure 2. Procedure of the Pattern Search Component

After each iteration of the ACO phase, we improve each ant independently with a local Pattern Search (PS) of neighboring solutions. Our pattern denoted by $\mathcal{D} \equiv \{e_1^+, e_1^-, \ldots, e_d^+, e_d^-\}$ is defined by the unit coordinate axes where *d* is the dimension of the problem. The PS is initialized by setting the step size Δ_i to a user specified initial Δ_o. The PS explores the coordinate axes for improving solution for *k* iterations. If the search does not find an improvement for any direction, then the step size Δ_i is reduced by half. The PS is stopped after a fixed number of iterations. If a thread reaches convergence to a user specified tolerance Δ_{tol} before the iteration limit reached, this research resets the step size Δ_i to the initial Δ_o to make use of computer resources that otherwise would be idle. By setting the number of iterations in the Pattern Search, the modeler can control the amount of resources dedicated to the ACO and PS sections of the search heuristic. The Pattern Search component is illustrated in Fig. 2.

IV. Parallelization Analysis and Implementation

A. Random Numbers

All the stochastic optimization methods need high quality random numbers. In traditional CPU-based ACOs, random numbers are typically generated as needed. In the GPU, all the random numbers are generated before the iterations starts, and these numbers are stored in the texture memory space to enable faster access by the kernels. The random number generator chosen is Mersenne Twister (Matsumoto and Nishimura 2008).

B. Fitness Functions and Feasibility Functions

The purpose of a fitness function is to evaluate the cost of an objective function. The purpose of a feasibility function is to make sure that the current ant meets all the constraints. In this project, all feasibility functions are only to make sure they are between the minimum and maximum values. The parallelization of fitness functions and feasibility functions are straightforward since these evaluations are independent in each thread.

C. Generation of New Ants

The generation of the new ants, i.e. solutions, is the most time-consuming part of the algorithm. It takes the Probability Vector and random numbers as the input. We keep a copy of the Probability Vector and random numbers in the texture memory to enable faster access of these data during new ants generation. This generation process is totally independent for each ant and thus is highly parallelizable. The CPU-GPU communication overhead is small since the information that needs to be frequently passed is only the Probability Vector.

D. Pattern Search

The Pattern Search is implemented as one kernel function. The Pattern Search process invokes frequent fitness function evaluations and feasibility checks. This task is ideally suited for the GPU due to being a large task with no communication between threads.

E. GPU and CPU Task Allocation

We proposed the SIMT-ACO-PS algorithm as in Fig. 3. It is a mixture of CPU and GPU function calls. We put several steps into GPU with each thread operating to construct and improve a single vector (solution).

A) Initialize T ants, each of which has P dimension variables (in CPU);
B) Evaluate the cost and constraints of all the ants (in GPU);
C) Sort the ants based on the cost (in CPU);
D) Initialize ACO and PS parameters (in CPU);
While (Not meeting terminate criterion)
E) Evaporate pheromone, with (1) (in CPU);
F) Use the best few ants to update the pheromone vector, with (2) (in CPU);
G) Update the Probability Vector with the pheromone information, with (3) (in CPU);

H) Generate the new ant solutions based on the Probability Vector (in GPU);
I) Evaluate the cost $f(\vec{x}_i)$ and constraints $c(\vec{x}_i)$ of the ants (in GPU);
J) Improve all the ants with Pattern Search (in GPU);
K) Sort the ants based on the cost $f(\vec{x}_i)$ (in CPU);
L) Collect the generation result (in CPU);
End While
M) Collect final result (in CPU).

Figure 3. Procedure of the SIMT - Ant Colony – Pattern Search

V. Computational Experiments and Results

The proposed Single Instruction Multiple Thread – Ant Colony – Pattern Search (SIMT-ACO-PS) algorithm for the general optimization functions has been implemented by the author. The algorithm was implemented in Visual C++ 2005 environment with the CUDA™ environment for programming the GPU. The computational experiments were executed on a Dell Precision 7400 Workstation computer with an Intel® Xeon™ E5420 CPU, 2GB memory, and an nVidia GeForce™ GTX 280 GPU. For benchmarking, the algorithm was also implemented with CPU-based only functions to compare the computation speed to the GPU-accelerated implementation. The same computer was used in testing both CPU and GPU versions. Twelve benchmark functions have been selected for these computational experiments [8]. Some of them are listed in the Appendix.

ACO performance is affected by its parameter setting [3]. Based on a set of initial testing with several problems of different sizes, we selected the following parameter settings:

- Pheromone update constant Q: 20;
- Exploration constant: 0.4;
- Global pheromone decay rate: 0.7;
- Pheromone sensitivity α: 1.0;
- Pheromone vector length: 1000;
- Initial values of the Pheromone Vector: 1.0E-6;
- Pattern Search: 20 iterations with Δ_0 set to the parameter range / 20. Δ_{min} is set to be $\Delta_0 / 2^{16}$.

A. Computational Performance Analysis

Table 1 shows the comparison of computation times on the Ackley Function with dimension 30, between the proposed SIMT-ACO-PS and CPU implementation of the same algorithm. The speedup values are calculated by dividing the CPU-only algorithm times by the GPU-accelerated algorithm times. Table 1(a) shows the speedup result of the ACO component. Table 1(b) shows the speedup result of the PS component. Table 1(c) shows the speedup result of the hybrid ACO-PS algorithm. The speedup increases as the number of threads (ants) increases. Note that in Table 1(b), it is 50 repetitive runs of 20-iteration Pattern Search, while in Table 1(c), total steps of the Pattern Search is equal to (50 ACO generations x 20 PS iterations). With a larger number of ants, the heavy computation tasks are primarily executed on the GPU greatly reducing run time compared to the CPU-only implementation.

The peak performance is expected at 15360 threads, which is organized into 60 blocks with 256 threads per block. For our GPU kernels, each multi-processor can take 2 blocks due to the 'register' constraints in GPU. It means each multi-processor can run 2 blocks of 256 threads, that is, 30 multi-processor (of GeForce GTX 280 GPU) times 2 blocks times 256 threads is equal to 15360 threads [6]. Therefore 30 multi-processors of a GeForce GTX 280 can take totally 60 blocks or its multiples to reach peak performance.

TABLE 1. Average Run Time Comparison Between CPU and GPU implementation in milliseconds on the Ackely Function with 30 variables, 50 ACO Generations, and 20 PS Iterations for 10 Runs, timed in m-sec

a) The speedup result of the ACO component

Threads	SIMT-ACO	ACO	Speedup
2048	405	3,953	9.77
4096	473	7,895	16.68
8192	623	15,734	25.24
12288	825	23,800	28.85
15360	908	29,783	32.81

b) The speedup result of the PS component

Threads	SIMT-ACO	ACO	Speedup
2048	1,580	107,849	68.27
4096	1,666	215,759	129.53
8192	2,703	431,424	159.60
12288	2,811	647,203	230.24
15360	3,139	808,947	257.70

c) The speedup result of the ACO-PS

Threads	SIMT-ACO	ACO	Speedup
2048	1,811	121,073	66.85
4096	1,913	241,933	126.50
8192	2,884	483,245	167.54
12288	3,027	724,753	239.46
15360	3,394	907,177	267.30

B. Speedup Comparison beteween CPU and GPU Implementations

Table 2 shows the comparison of computation times between the CPU and GPU implementation of the ACO-PS algorithm for all the 12 benchmark functions. We conduct a test with 15360 threads to determine the times spent on solving each benchmark problem. The average results of 10 runs were collected and summarized.

C. Solution Quality under Time Limits

Time to solution is a critical performance measure for an optimization procedure. Given enough time, both CPU and GPU versions can give satisfactory results. When time is limited, the result obtained with GPU-accelerated algorithm is notably better. Table 3 shows the mean best solutions and standard deviation for 10 runs of Ant Colony, Pattern Search, and Ant Colony plus Pattern Search on CPU and GPU hardware. All the tests are limited to one second. The

component or algorithm is allowed to finish any iteration started before the one second limit. It can be seen that the hybrid ACO-PS generally gets better results given the one second time limit. However in the GPU platform, PS-only can achieve better results in some problems because it is able to complete more iterations than ACO-PS given the time limit.

TABLE 2. COMPARISON OF AVERAGE COMPUTATION TIMES IN MILLISECONDS BETWEEN THE SIMT-ACO-PS AND THE ACO-PS ALGORITHMS FOR 10 MONTE CAROL RUNS. THE COMPUTATION SETTING WERE 15360 THREADS (ANTS), 50 ACO GENERATIONS, 30 VARIABLES, AND 20 PS ITERATIONS, TIMED IN M-SEC

	CPU Version			GPU Version			GPU/CPU speedup		
# Problems	ACO-only	PS-only	ACO-PS	ACO-only	PS-only	ACO-PS	ACO-only	PS-only	ACO-PS
1 Ackley	29,791.9	812,524.8	906,588.3	957.8	5,029.7	3,423.4	31.10	161.55	264.82
2 Griewank	29,921.7	892,492.1	1,258,026.3	975.0	6,374.9	4,854.6	30.69	140.00	259.14
3 Penalty1	30,392.0	6,784,877.5	6,813,857.0	1,053.1	44,640.4	29,974.8	28.86	151.99	227.32
4 Penalty2	30,860.8	6,836,257.5	6,749,952.5	1,029.7	46,140.3	30,224.8	29.97	148.16	223.32
5 Quartic	74,687.0	193,570.3	788,262.2	986.0	3,293.7	1,951.6	75.75	58.77	403.91
6 Rastrign	30,113.9	826,028.9	834,011.8	1,012.4	5,287.5	3,920.3	29.75	156.22	212.74
7 Rosenbrock	29,090.4	423,776.2	462,042.4	923.5	4,614.0	3,276.5	31.50	91.85	141.02
8 Schwefel 1.2	30,099.8	1,007,189.1	1,191,618.9	1,204.7	10,429.6	9,250.0	24.99	96.57	128.82
9 Schwefel 2.22	29,313.9	692,343.7	540,001.2	909.3	3,153.1	1,717.1	32.24	219.58	314.48
10 Schwefel 2.21	30,865.4	103,093.8	198,937.8	882.8	2,728.1	1,289.1	34.96	37.79	154.32
11 Sphere	28,649.8	71,187.5	509,240.5	923.5	2,739.1	1,326.6	31.02	25.99	383.87
12 Step	29,303.0	579,032.8	579,848.6	867.1	3,000.0	1,926.5	33.79	193.01	300.99

VI. CONCLUSIONS

In this paper, we present a Single Instruction Multiple Thread – Ant Colony – Pattern Search (SIMT-ACO-PS) algorithm for general bound constrained optimization functions with graphics hardware acceleration. Ant Colony Optimization (ACO) is used in the global search phase and Pattern Search (PS) is used in the local search phase for improvement. Through parallelization analysis, the classical ACO algorithm and the PS algorithm are adapted for data-parallel computing in a GPU desktop parallel computing environment. The computation time was significantly reduced and the better optimization results can be obtained more quickly with massive ants and Pattern Search, by leveraging GPU parallel computing.

ACKNOWLEDGMENT

This work was partially supported by the Lamar Research Enhancement Grants to Dr. W. Zhu and Dr. J. Curry at Lamar University. Partial support for this work was provided by a grant from the Texas Hazardous Waste Research Center to Dr. W. Zhu, Dr. J. Curry and Dr. H. Lou at Lamar University. Partial support for this work was provided by the National Science Foundation's Award No. 0737173 to Dr. W. Zhu. Their support is greatly appreciated.

Appendix: Test Functions

1) Ackley Function:

$$f_1(x) = 20 + e - 20e^{-\frac{1}{5}\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}} - e^{\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)},$$

$$-30 \le x_i \le 30,\ i = 1,2,...,n,\ \min(f_1) = f_1(0,...,0) = 0$$

2) Griewank Function

$$f_2(x) = 1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n}\cos\left(\frac{x_i}{\sqrt{i}}\right),$$

$$-600 \le x_i \le 600,\ i = 1,2,...,n$$

$$\min(f_2) = f_2(0,...,0) = 0$$

3) Generalized Penalized Function 1

$$f_3(x) = \frac{\pi}{30}\left\{10\sin^2(\pi y_1) + \sum_{i=1}^{n-1}(y_i - 1)^2 \bullet \left[1 + 10\sin^2(\pi y_{i+1})\right] + (y_n - 1)^2\right\}$$

$$+ \sum_{i=1}^{n} u(x_i, 10, 100, 4)$$

where

$$u(x_i, a, k, m) = \begin{cases} k(x_i - a)^m, & x_i > a \\ 0, & -a \le x_i \le a \\ k(-x_i - a)^m, & x_i < -a \end{cases}, \qquad y_i = 1 + \frac{1}{4}(x_i + 1)$$

$$-50 \le x_i \le 50,\ i = 1,2,...,n,\ \min(f_3) = f_3(-1,...,-1) = 0$$

4) Generalized Penalized Function 2

5) Quartic Function

6) Rastrigin Function

7) Rosenbrock Function

8) Schwefel's Problem 1.2

9) Schwefel's Problem 2.22

10) Schwefel's Problem 2.21

11) Sphere Function

12) Step Function

REFERENCES

[1] E. Alba, Parallel Meta-heuristics: a New Class of Algorithms, edited by Enrique Alba, John Wiley Inc., ISBN-13 978-0-471-67806-9, 2005.

[2] M. Dorigo, Optimization, Learning and Natural Algorithms, Ph.D. Dissertation, Politecnico di Milano, Italy, 1992.

[3] Dorigo, M. and T. Stuetzle, 2004, Ant Colony Optimization, MIT Press, ISBN 0-262-04219-3.

[4] M. Matsumoto; T Nishimura, Mersenne Twister. A 623-dimensionally equidistributed uniform pseudorandom number generator. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf

[5] Nguyen, H., Editor, 2007. GPU Gems 3, New York: Addison-Wesley.

[6] nVidia, 2008, Cuda Programming Guide Version 2.1. nVidia, Available at: http://developer.nvidia.com/object/cuda.html.

[7] V. Torczon, "On the convergence of Pattern Search algorithms", SIAM J. Optima., 7, pp. 1–25, 1997.

[8] X. Yao, Y. Liu and G. Lin, "Evolutionary Programming Made Faster", IEEE Transactions on Evolutionary Computation (3) pp. 82-102, July 1999.

[9] W. Zhu, J. Curry, A. Marquez, "SIMT Tabu Search with Graphics Hardware Acceleration on the Quadratic Assignment Problem", the International Journal of Production Research, 2008, in press.

[10] W. Zhu, J. Curry, "Particle Swarm with Graphics Hardware Acceleration and Local Pattern Search on Bound Constrained Optimization Problems", IEEE Symposiums Series on Computational Intelligence, Nashville, TN, USA, March 30 ~ April 2, 2009.

[11] W. Zhu, J. Curry, "Multi-walk Parallel Pattern Search on a GPU computing Platform", Proceedings of the IEEE International Conference on Computational Science, Baton Rouge, LA, USA, May 25 ~ 27, 2009.

TABLE 3. MEAN BEST SOLUTIONS COMPARISON BETWEEN THE GPU AND CPU IMPLEMENTATIONS IN A ONE SECOND TIME LIMIT FOR TEN MONTE CARLO RUNS. COMPUTATION SETTING ARE 30 VARIABLES, 20 PS ITERATIONS, AND 15360 THREADS

a) SIMT-ACO-PS

		Mean Best Solutions			Standard Deviation		
#	Problems	ACO-only	PS-only	ACO-PS	ACO-only	PS-only	ACO-PS
1	Ackley	3.7563	0.0000	0.0000	0.9664	0.0000	0.0000
2	Griewank	1.9834	0.0000	0.0000	1.0090	0.0000	0.0000
3	Penalty1	3.0930	0.0000	0.0000	3.8734	0.0000	0.0000
4	Penalty2	3.4548	0.0000	0.0004	6.1503	0.0000	0.0004
5	Quartic	0.0018	0.0000	0.0000	0.0029	0.0000	0.0000
6	Rastrign	20.7891	45.1712	0.0000	17.6445	16.3501	0.0000
7	Rosenbrock	841.1869	0.0080	14.0661	2,393.9290	0.0081	7.7966
8	Schwefel 1.2	28,862.5313	18.7696	0.4840	17,848.6270	5.1311	0.4935
9	Schwefel 2.22	2.0480	0.0001	0.0000	1.7084	0.0000	0.0000
10	Schwefel 2.21	26.6323	0.0001	8.0703	14.2504	0.0000	7.6265
11	Sphere	1.0696	0.0000	0.0000	1.1268	0.0000	0.0000
12	Step	199.0000	0.0000	0.0000	313.9675	0.0000	0.0000

b) CPU ACO-PS

		Mean Best Solutions			Standard Deviation		
#	Problems	ACO-only	PS-only	ACO-PS	ACO-only	PS-only	ACO-PS
1	Ackley	12.6467	0.0508	0.0131	3.5420	0.0026	0.0062
2	Griewank	77.5174	0.0680	0.0117	99.5972	0.0063	0.0185
3	Penalty1	60,567,584.0000	0.0001	0.0001	62,224,704.0000	0.0000	0.0000
4	Penalty2	162,868,992.0000	0.0012	0.0008	161,005,712.0000	0.0002	0.0004
5	Quartic	8.8967	0.0000	0.0000	16.4763	0.0000	0.0000
6	Rastrign	147.7898	77.8069	62.5848	103.0416	7.8190	43.5158
7	Rosenbrock	51,808,708.0000	28.8229	25.3985	46,708,148.0000	2.2270	5.0763
8	Schwefel 1.2	41,001.6484	5,029.4971	3,223.1526	9,058.2969	482.3401	1,558.2773
9	Schwefel 2.22	32.2660	0.0871	0.0305	24.1209	0.0034	0.0235
10	Schwefel 2.21	38.3105	23.4050	24.2147	24.1161	1.9756	11.2045
11	Sphere	85.1837	0.0004	0.0001	110.6736	0.0000	0.0001
12	Step	13,577.0996	0.0000	0.0000	16,216.8896	0.0000	0.0000