Multi-walk Parallel Pattern Search Approach on a GPU Computing Platform

Weihang Zhu and James Curry*

P.O.Box 10032, Department of Industrial Engineeirng,
Lamar University, Beaumont, Texas, 77710, USA
Weihang.Zhu@lamar.edu, jcurry@my.lamar.edu

Abstract. This paper studies the efficiency of using Pattern Search (PS) on bound constrained optimization functions on a Graphics Processing Unit (GPU) computing platform. Pattern Search is a direct search optimization technique that does not require derivative information on non-linear programming problems. Pattern Search is ideally suited to a GPU computing environment due to its low memory requirement and no communication between threads in a multi-walk setting. To adapt to a GPU environment, traditional Pattern Search is modified by terminating based on iterations instead of tolerance. This research designed and implemented a multi-walk Pattern Search algorithm on a GPU computing platform. Computational results are promising with a computing speedup of 100+ compared to a corresponding implementation on a single CPU.

Keywords: Nonlinear Optimization, Pattern Search, GPU, CUDA.

1 Introduction

Graphics Processing Unit (GPU) computing is an emerging technology of parallel computing due to its low cost per instruction. Modern GPU computing technology features a 'Single Instruction – Multiple Threads' mode, which is amenable to heavy and repetitive computation. Researchers have employed GPU in many fields such as physically-based simulation, financial engineering, and image/video processing [8]. Several researchers have employed GPU for optimization. Li et al. (2006) studied parallel Particle Swarm Optimization [6]. Zhu et al. (2008) examined parallel Tabu Search for the Quadratic Assignment Problem [11]. This paper examines using the GPU for unconstrained nonlinear optimization with bound constraints with Pattern Search, a direct search method commonly found in non-linear optimization. The objective is to minimize a nonlinear function $f(x)$ subject to range constraints that $a_i \leq x_i \leq b_i$ where $x \in \Re^n$ and $f : \Re^n \to \Re$. Such global optimization problems over continuous spaces are ubiquitous throughout the scientific community. When the objective function is non-linear and non-differentiable, direct search approaches are the methods of choice.

* Corresponding Author.

G. Allen et al. (Eds.): ICCS 2009, Part I, LNCS 5544, pp. 984 – 993, 2009.

Direct search methods are characterized by neither requiring nor explicitly approximating derivative information. Direct search methods are considered as 'zero-order methods', which are different from 'first-order methods' such as steepest descent method, or 'second-order methods' such as Newton's method. The 'order' of a method indicates the highest order term being used in the local Taylor series approximation to the nonlinear function *f*. With 'zero-order', direct search methods work directly with *f*, instead of with a local approximation to *f*.

Pattern Search methods are a subset of direct search methods. Pattern Search is traditionally employed when the gradient of the function is not reliable when performing the search [3], [4], [9]. This robustness makes Pattern Search a safe choice for solving many different types of problems [1]. Parallel Pattern Search methods have been proposed in the past research in CPU platforms [2], [3]. However, instead of running multiple simultaneous searches, their methods assign the individual search direction to CPUs within the pattern search. In this paper, we present a 'Single Instruction – Multiple Threads – Pattern Search' (SIMT-PS) algorithm developed on a GPU platform. This algorithm takes the approach of starting multiple simultaneous Pattern Searches from massive random starting points to generate a large computation tasks to the GPU. Significant speedup in computation and optimization speed can be achieved with this promising approach, as suggested by our computational experiments results.

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

2 GPU Computing

GPU computing is an exciting new computing environment that is fundamentally different than the traditional CPU environment. A GPU is designed to process thousands of threads simultaneously enabling high computational throughput across large amounts of data. This research selects the Compute Unified Device Architecture (CUDA) technology from nVidia™ to implement our algorithm. The CUDA environment allows a software developer to program a GPU for general purpose computing in a C programming environment [8].

The CUDA environment is designed to run thousands of threads concurrently with a same instruction set in a data parallel manner. Each thread runs an instruction set called a 'kernel'. Developers in a GPU environment have several different memory locations to store data. 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 during kernel run time. However, during a Pattern Search, all the searches are completed independently. Hence there is no need to use any 'shared memory' to exchange the information between threads during run time.

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. Constant memory is fast read-only memory whose size cannot be dynamically changed in runtime. An investigation of the Pattern Search reveals that there is no read-only data during the searches, as the data are constantly updated. Therefore, texture memory and constant memory are not used in our implementation of the SIMT-PS algorithm. The nVidia GeForce GTX 280 GPU hardware employed in this paper has 30 multi-processors. Each multi-processor has 8 processors (or cores). This amounts to 240 data-parallel processors (cores) on one GPU board. Each multi-processor has 16K shared memory, 16K registers, 64K constant memory, and access to 1GB global device memory and texture memory for larger data storage [7]. Due to the different performance of memory locations and the limited amount of fast and flexible memory locations, algorithm implementation and design choices must be guided by memory limitations.

After compilation by the CUDA environment, a program runs as a kernel in a GPU. A kernel takes input parameters, conducts computations, and outputs the results to device memory where the result can be read by the CPU. Each thread must perform the same operation in the kernel, but the input data can be different. The CPU owns the host code that prepares input data and accepts output values from the GPU. The CPU is also responsible for reading and writing data files, storing solution values, and managing threads. The output data from the GPU is written to global device memory to be retrieved by a CPU program. Also, code with excessive conditional branching should also be avoided in a GPU environment since all threads execute the same instructions. The communication between the CPU and GPU requires some overhead time. To compensate for the overhead time, developers must send tasks to the GPU that are significantly larger than the communication overhead.

In our SIMT-PS algorithm, the initial solutions are generated by the CPU and then passed to GPU for Pattern Search. The solutions are constantly changing during the Pattern Search procedure. Hence, the solution data must be kept in the Global Device Memory. To minimize the impact of repetitive read and write access with the Device Memory, the solution array data are organized into a coalesced structure. At the end of Pattern Search, the solution data are copied back to the CPU host memory for further processing. The CPU-GPU communication is thus minimized. The computational experiments results also suggest that a multi-walk Pattern Search is an appropriate design for the GPU computing platform due to its low memory requirement and simple repetitive tasks.

3 Pattern Search

The basic Pattern Search algorithm is a direct search method that does not require derivative or second derivative information [9]. While this robustness makes Pattern Search a safe choice for solving many different types of problems [1], this research selects Pattern Search due to the memory limitation and the desire to avoid excessive branching in GPU hardware. The basic PS algorithm moves along the coordinate axes or other user defined positive spanning set to improve an existing solution in a greedy way. The step size Δ is reduced, typically in half, when an improvement is not found for any direction.

The feature of Pattern Search makes it amenable to parallel computing as it involves frequent objective function evaluations. For a complex objective function,

this repetitive function evaluation computation task becomes very heavy and thus makes the Pattern Search a questionable selection as a direct search method on a single CPU platform. Parallel computing can alleviate this problem. Parallel Pattern Search methods have been proposed in the past research in CPU platforms [2], [3]. In a GPU parallel computing platform, the benefit can be much more evident, as GPU computing does not have the large overhead of a CPU cluster which often seriously reduces the benefit of parallel computing. Therefore, the savings on the computation time of Pattern Search on a GPU computing platform can be tremendous.

Alternatives to Pattern Search include Nelder-Mead simplex search method and gradient based methods [5]. The Nelder-Mead simplex search requires data storage for $d+1$ solution vectors each of size d where d is the number of variables in the problem. Gradient methods based methods require $O(d^2)$ memory. The Pattern Search method requires a single solution vector with d elements, the cost of the prior solution, and the current step sizes for $d+3$ memory elements per search thread. For current GPU hardware and our algorithm design, a number of 15360 parallel threads achieved the best speedup performance, as presented in Section 5. For example, with a 100 variable problem, single precision storage, and 15360 threads, Pattern Search requires 6 MB of memory compared to over 600 MB for a gradient based method and Nelder-Mead simplex search method. Given the current limitation in memory in GPU hardware, this difference in memory requirement is a tremendous advantage for Pattern Search over alternative methods that require more memory. In addition, this small memory consumption makes it possible for Pattern Search to work as a sub-component in a bigger problem-solving framework.

4 'Single Instruction – Multiple Threads – Pattern Search' (SIMT-PS) Algorithm

Pattern Search is modified for the GPU SIMT computing environment by conducting multiple searches in parallel and searching a fixed number of iterations instead of a limit based on solution tolerance. Each thread represents one independent search. Let x denote the solution vector of a particular thread composed of d elements. The initial values of x are set randomly based on a uniform distribution from the lower bound to the upper bound.

$$x_i = \text{Uniform}[a_i, b_i] \quad i = 1, 2\ldots, d \tag{1}$$

From this initial seed point, we improve each thread using a Pattern Search algorithm.

Our pattern denoted by $\mathsf{D} \equiv \left\{e_1^+, e_1^-, \ldots, e_d^+, e_d^-\right\}$ is defined by the unit coordinate axes where d is the dimension of the problem where e_j is the standard unit base vectors. The PS is initialized by setting the step size Δ_i to a user specified initial Δ_o. The PS in this paper explores the coordinate axes for improving solution for k iterations. In general, the pattern can be user defined positive spanning set but the coordinate axis is the easiest to implement. If the search does not find an improvement for *any direction*, then the step size Δ_i is reduced by half. A classical PS is typically stopped once the step size is less than a user specified tolerance, but in our algorithm, the PS is stopped after a fixed number of iterations. With a large number of threads in a SIMT

computing environment, waiting until all threads reach convergence is not an effective use of computing resources. 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. Fig. 1 presents the steps of the algorithm. All threads are initialized to random starting solutions for the solution vector x and step sizes of Δ_o. For k_{max} iterations the algorithm performs a Pattern Search operation on all threads for each variable of the problem. If no improvement is found for any variable during an iteration, the step size is reduced by half. Finally, the best solution is selected from all of the threads (Fig. 1).

For all threads (done in GPU)

a) Initialize the step size to Δ_o and the solution x_i to a random starting solution where x_i= Uniform[a_i, b_i];

// loop until max iteration

For (iteration $k = 0 \ldots k_{max}$)

For (dimension $j = 1 \ldots d$)

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)

End For (threads)

e) Select best thread based on solution cost (done in CPU)

Fig. 1. Procedure of the SIMT-Pattern Search algorithm

5 Computational Results and Analysis

The proposed Single Instruction Multiple Thread –Pattern Search (SIMT-PS) algorithm for the general optimization functions has been 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® Core™ 2 Duo 2.5GHz CPU, 3GB 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. The GPU version of the algorithm requires many threads to unleash its full potential. The peak speedup performance is expected when the number of threads is sufficient to keep the multi-processors busy at the same time. The GeForce GTX 280 GPU employed in this research has 30 multi-processors. Due to register constraints, each multi-processor can support 2 blocks with each block having 256 threads. In our hardware, 15360 (30 multi-processors * 2 blocks per multi-processor * 256 threads per block) provides good performance per thread. As anticipated by this analysis, Table 1 shows that the GPU code reaches peak performance relative to the CPU code when the number of threads is 15360.

Twelve benchmark functions have been selected for these computational experiments [10]. These benchmark functions are listed in the Appendix A. The Δ_o is set to be equal to (parameter range / 20). During the search process, Δ can be reduced to as small as Δ_{tol}, which is set to be $\Delta_o/2^{16}$.

We hope to first gain better understanding of how and where the SIMT-PS improve over its corresponding CPU implementation. To attain this goal, we picked the first benchmark function in the Appendix A, the Ackley function, for this investigation. Table 1 shows a comparison of computation times between the proposed SIMT-PS and CPU implementation of the same algorithm. Both software implementations execute the same search to ensure a fair comparison. The tests were conducted on Ackley function with dimension 20 (i.e., 20 variables). For the SIMT-PS algorithm, the recorded time is combined CPU and GPU code running time, including initialization overhead. As shown in Table 1, the computation time increases as the number of threads (parallel searches) increases. As the number of threads increases, the GPU codes performance per thread improves whereas the CPU performance remains relatively constant. The speedup values are calculated by dividing the CPU PS algorithm times by the SIMT-PS algorithm times.

Table 1. Comparison of computation times between the SIMT-PS and the PS algorithms as function of thread numbers (Ackley Function, 20 variables, 20 PS iterations, average of 10 Monte Carlo Runs, time shown in milliseconds)

Threads	GPU Code	CPU Code	Speedup
2048	21.9	1267.1	57.86
4096	29.7	2543.7	85.65
8192	51.6	5098.4	98.81
12288	64.1	7629.6	119.03
15360	76.6	9535.9	124.49

To understand the performance improvement on specific computational tasks within the algorithm, we examined the run time of different portions of the code in both the GPU and CPU algorithm implementations. As can be seen from Table 2, Pattern Search on the GPU platform significantly reduced the computation time for the search process. The 'Other functions' listed in the Table 2 include problem initialization, initial objective function evaluations and summary computation. The 'Select Best Thread' operations in both versions are completed with CPU code, but in the SIMT-PS algorithm, additional time is needed for memory manipulation.

Table 2. CPU and GPU time breakdowns for each task and their comparison (15360 threads, 20 variables, 20 PS iterations, time shown in milliseconds)

Task	GPU Code	CPU Code	Time %
1) Other functions	33.81	46.60	72.5%
2) Select Best Thread	6.57	4.65	141.5%
3) Pattern Search	35.73	9446.68	0.4%

Two concerns when designing GPU programs are CPU overhead for preparing for GPU tasks and the communication overhead between the CPU and the CPU. We conduct several tests with different numbers of threads to determine the times spent on each GPU task. The results are presented in Table 3. These tests are conducted with the assistance of *CUDA Visual Profiler* software, a tool for analyzing GPU kernel performance, available for download from nVidia website [7]. The 'GPU' column is the total time on GPU in executing the task. The 'CPU' column is the overhead time needed in CPU in order to run the corresponding GPU task. The '% GPU time' column is found by dividing each GPU task times by the total time spent in the GPU. The actual time spent on each GPU task is the sum of the 'GPU' and 'CPU' columns. The results show that the Pattern Search kernel take the majority of the time spent on GPU. The CPU-GPU memory synchronization takes a very small portion of the GPU time. The results also show that the CPU overhead is reasonably small in our implementation. Tracking CPU and communication overhead is an important performance tuning tool when developing GPU-enabled algorithms.

Table 3. GPU time and corresponding CPU overhead time used on each GPU task (15360 threads, 20 variables, for a short run, time shown in μ-seconds)

GPU Task	Calls	GPU μsec	CPU μsec	Percent of Time
Pattern Search	21	729850	110	98.5%
Initial Fitness Functions	22	1138	199	0.2%
Memory Synchronization	107	9717		1.3%

As shown in Table 4, the performance of the GPU version of the code is significantly faster than the CPU version of the code on 12 test problems as listed in the Appendix A. The speedup ranged from 30 to 138. The performance improvement is generally greater on problems where the objective function requires more time to evaluate and the overall solution time is longer.

Time to solution is a critical performance measure for an optimization procedure. Table 5 shows the comparison between the SIMT-PS and the PS with a run time limit of 1 second. Since the SIMT-PS algorithm is very fast (typically finishes within 100 milliseconds for a 30 variable and 20 iteration Pattern Search), we make it run repetitively from different initial solutions once one round of parallel Pattern Search is over. Given enough time, both CPU and GPU versions of our implementation should give satisfactory results. When time is limited, the result obtained with GPU-accelerated SIMT-PS algorithm is better, as it has time to search wider search space. The optimal solutions of all test problems are 0. In fact, the CPU PS implementation for some test functions cannot finish one round of Pattern Search within 10 seconds, but we let the CPU code run to finish at least one Pattern Search. The actual time spent by each algorithm is also listed in Table 5.

Table 4. Comparison of Computation Times between the SIMT- PS and its corresponding CPU Implementations on all 12 test functions (15360 threads, 20 variables, 20 PS iterations, time shown in milliseconds)

#	Problems	CPU	GPU	Speedup
1	Ackley	9523.4	78.2	121.8
2	Griewank	9675	92.2	104.9
3	Penalty1	63537.5	484.3	131.2
4	Penalty2	61865.6	509.4	121.4
5	Quartic	2375	51.6	46.0
6	Rastrigin	9465.6	70.3	134.6
7	Rosenbrock	6421.9	70.3	91.3
8	Schwefel 1.2	15353.1	117.2	131.0
9	Schwefel 2.22	1890.7	50	37.8
10	Schwefel 2.21	2120.3	46.9	45.2
11	Sphere	1393.7	45.3	30.8
12	Step	6943.8	50	138.9

Table 5. Time limited results comparison between the SIMT-PS and its corresponding CPU Implementation (10 Monte Carlo runs, 15360 threads, 30 variables, 20 PS iterations, max 10 seconds, time shown in milliseconds)

#	Problems	Best Solution		Mean Best Solution		Actual Time Spent	
		CPU PS	SIMT-PS	CPU PS	SIMT-PS	CPU PS	SIMT-PS
1	Ackley	0.000	0.000	0.000	0.000	19,907	10,074
2	Griewank	0.064	0.004	0.126	0.007	21,271	10,070
3	Penalty1	0.000	0.000	0.001	0.000	140,970	10,041
4	Penalty2	0.008	0.000	0.021	0.000	136,574	10,374
5	Quartic	0.000	0.000	0.000	0.000	10,131	10,066
6	Rastrigin	54.588	18.196	73.593	24.666	21,036	10,076
7	Rosenbrock	22.340	1.020	25.946	13.146	14,430	10,078
8	Schwefel 1.2	4,417.340	279.460	5,036.064	539.862	46,755	10,251
9	Schwefel 2.22	0.000	0.000	0.000	0.000	13,441	10,042
10	Schwefel 2.21	12.900	3.200	17.660	3.510	12,494	10,027
11	Sphere	0.000	0.000	0.000	0.000	11,708	10,045
12	Step	0.000	0.000	0.000	0.000	15,142	10,030

6 Conclusions and Future Research

On a GPU computing platform, Pattern Search is an effective optimization method for nonlinear bound constrained optimization due to its small memory requirement and fast parallel objective function evaluations. Using GPU hardware can generally speed

up Pattern Search by a factor of 100 compared to CPU implementations. On a hardware cost per computing operation comparison, GPU computing is currently attractive for optimization heuristics. Research and practitioners who solve optimization problems with heuristics always benefit from low cost computing resources. Improved computing resources can be deployed to solve problems quicker or to find better solutions. This low cost platform should encourage heuristic designers to develop algorithms that are effective in a data parallel, low memory environment. The approach taken in this paper of using a simple search procedure with a massive number of search threads is a promising approach for applying GPU technology to optimization problems. As future research, computational studies exploring the performance of alternative heuristics in low memory data parallel environments would aid the adoption of GPU technology to optimization problems.

Acknowledgments. 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. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF. Their support is greatly appreciated.

References

1. Audet, C., Dennis Jr., J.E.: Analysis of Generalized Pattern Searches. SIAM Journal of Optimization 13, 889–903 (2003)
2. Hough, P.D., Kolda, T.G., Torczon, V.J.: Asynchronous Parallel Pattern Search for Nonlinear Optimization, SAND2000-8213, Sandia Lab Reports (January 2000)
3. Kolda, T.G.: Revisiting Asynchronous Parallel Pattern Search for Nonlinear Optimization. SIAM Journal of Optimization 16, 563–586 (2005)
4. Lewis, R.M., Torczon, V.J.: Why Pattern Search Works, NASA/CR-1998-208966, ICASE Report No. 98-57 (1998)
5. Lewis, R.M., Torczon, V.J., Trosset, M.W.: Direct Search Methods: Then and Now. Journal of Computational and Applied Mathematics 124(1-2), 191–207 (2000)
6. Li, J., Wan, D., Chi, Z., Hu, X.: A Parallel Particle Swarm Optimization Algorithm based on Fine-grained Model with GPU-Accelerating. Journal of Harbin Institute of Technology 38, 2162–2166 (2006)
7. nVidia: CUDA Programming Guide V 2.0 (2008), `http://www.nvidia.com/object/cuda_get.html`
8. Nguyen, H. (ed.): GPU Gems 3. Addison-Wesley, New York (2007)
9. Torczon, V.J.: On the Convergence of Pattern Search Algorithms. SIAM Journal of Optimization 7, 1–25 (1997)
10. Yao, X., Liu, Y., Lin, G.: Evolutionary Programming Made Faster. IEEE Transactions on Evolutionary Computation 3, 82–102 (1999)
11. Zhu, W., Curry, J., Marquez, A.: SIMD Tabu Search with Graphics Hardware Acceleration on the Quadratic Assignment Problem. Accepted by the International Journal of Production Research (2008)

Appendix A – Test Functions

Due to the page limit, only two test functions are described here. Other functions definition can be found in reference [10].

1) Ackley Function:

$$f_1(x) = 20 + e - 20e^{-\frac{1}{5}\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}} - e^{\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)}, \quad -30 \le x_i \le 30, \quad i = 1,2,...,n$$

$$\min(f_1) = f_1(0,...,0) = 0$$

2) Griewank Function

$$f_2(x) = 1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n}\cos\left(\frac{x_i}{\sqrt{i}}\right), \quad -600 \le x_i \le 600, \quad i = 1,2,...,n$$

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

3) Generalized Penalized Function 1
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