ISSN 1330-3651
UDC/UDK 004.92.02:004.421.2

IMPLEMENTING GENETIC ALGORITHMS TO CUDA ENVIRONMENT USING DATA PARALLELIZATION

Masashi Oiso, Yoshiyuki Matsumura, Toshiyuki Yasuda, Kazuhiro Ohkura

Original scientific paper

Computation methods of parallel problem solving using graphic processing units (GPUs) have attracted much research interests in recent years. Parallel computation can be applied to genetic algorithms (GAs) in terms of the evaluation process of individuals in a population. This paper describes yet another implementation method of GAs to the CUDA environment where CUDA is a general-purpose computation environment for GPUs provided by NVIDIA. The major characteristic point of this study is that the parallel processing is adopted not only for individuals but also for the genes in an individual. The proposed implementation is evaluated through eight test functions. We found that the proposed implementation method yields 7,6-18,4 times faster results than those of a CPU implementation.

Keywords: *Genetic Algorithms, GPU, CUDA*

Implementiranje genetskih algoritama u CUDA okruženje upotrebom paralelizacije podataka

Izvorni znanstveni članak

Računarske metode rješavanja paralelnih problema korištenjem grafičkih obradnih jedinica (GPUs) zadnjih su godina pobudile veliki interes. Paralelno izračunavanje može se primijeniti na genetske algoritme (GAs) u odnosu na proces evaluacije jedinki u populaciji. Ovaj rad opisuje još jednu metodu primjene GAs na CUDA okruženje gdje je CUDA računarsko okruženje opće namjene za GPUs koje daje NVIDIA. Osnovna karakteristika ovog istraživanja leži u tome da se paralelna obrada koristi ne samo za jedinke nego i za gene u jedinki. Predložena implementacija se procjenjuje kroz osam ispitnih funkcija. Ustanovili smo da predložena metoda implementacije daje 7,6-18,4 puta brže rezultate od onih kod primjene CPU.

Ključne riječi: *genetski algoritmi, GPU, CUDA*

1 Introduction

Evolutionary computation (EC) is well recognized as an effective method for solving many difficult optimization problems. However, EC generally entails large computational costs because it generally evaluates all solution candidates in a population for every generation. To overcome this disadvantage, parallel processing has been utilized such as in a form of Cluster computing [1]. In addition, in the late 1990s, Grid computing, which enables high- performance computing by connecting computational resources on the Internet, began applying EC for utilizing high-speed networks [2, 3]. In this way, parallel and distributed computing techniques have become widespread as a means of increasing the processing speed of EC.

In recent years, graphics processing units (GPUs), which were originally a device for graphics applications, have attracted much research interest from the viewpoint of low-cost, high-performance computing. Because GPUs possess an architecture that is specialized for graphics applications, they are good at processing simple and same calculations with a large amount of data. On the other hand, because of their special architecture, GPUs are not good at processing tasks that include many conditional branches. However, the performance of GPUs has rapidly been improving in recent years such that their peak performance is much higher than that of CPUs. Another point to be noted is that GPU also has a higher power-to-watt ratio than CPU. The concept of using GPU not only for graphics applications but also for general-purpose applications is called general-purpose computation on GPUs (GPGPU) [4, 5, 6, 7, 8]. GPGPU is growing rapidly in various fields where low-cost and high-performance computation is necessary and valuable.

GPGPU is also gaining popularity among EC researchers, initially those in the field of genetic programming [9, 10, 11, 12, 13]. The GA researchers soon followed this trend. Pospichal and Jaros [14] adopted the island model for implementation with the conditions of 64 islands and the population size of 256 in each island, and they yielded a speedup ratio of 2,602 compared to the CPU implementation. Tsutsui and Fujimoto [15] proposed an implementation of distributed GA similar to the island model for a population size of 11,520 in an experiment for quadratic assignment problems and showed a speedup ratio of 23,9 compared to the CPU implementation. They also analyzed the peak performance of their implementation by eliminating data transition between subpopulations, which was the bottleneck of GPU calculation [16]. Debattisti et al. [17] implemented the entire process of a simple GA to GPU, except for the initialization process. They conducted their experiments using the OneMax problems and achieved a speedup rate of 26 for a population size of 512 and a genome length of 256 compared to sequential execution on CPU.

Here, we can identify that all three implementation methods have achieved high performance, generally because they employed a population size larger than that generally employed in GAs in order to achieve a sufficient parallelization effect. In contrast, in this paper, yet another implementation method of GA to GPU is proposed. This is mainly based on the idea of adopting the parallelization not only of individuals but also of genes. These two parallelizations realize massive parallelization on GPU to accelerate GA computation in a standard population size, which would be more appropriate for utilizing the characteristic of a many-core computation than the previous related work.

The rest of this paper is organized as follows. Section 2 describes the development environments for GPGPU. Section 3 presents the proposed implementation method of GA to GPU. In Section 4, experiments of function optimization problems are conducted to examine the basic performance of the proposed method. Finally, Section 5 presents the conclusions of this paper.

2 Development Environment for GPGPU

2.1 Brief History of Development Environment

Around the year 1999, GPU gained popularity and had the specialized architecture to execute a fixed function pipeline. With the installation of a programmable shader to GPU, the degree of freedom of GPU calculation improved dramatically. Thus GPU has been used for general-purpose applications as GPGPU.

In the initial stage of GPGPU, programming using low-level assembly languages was indispensable, and thus, efforts were made on the implementation rather than on designing of shader algorithms. Then, graphics shading languages such as C for graphics (Cg) [18] by NVIDIA in 2002, High Level Shading Language (HLSL) [19] by Microsoft, and OpenGL Shading Language (GLSL) [20] were released, and GPGPU programming became easier to use. However, an in-depth understanding of GPU architecture and graphics processing is still required. As a result, GPGPU was reserved for graphics programming experts. Therefore, the development of high-level languages has been highly anticipated.

In the next stage, since 2004, some development environments have been proposed. Sh [21], was developed in Waterloo University and consists of the C++ library. Brook [22] was developed in Stanford University, which is based on C language and was extended to deal with stream data. In particular, Compute Unified Device Architecture (CUDA) [23] was released by NVIDIA as a development environment for their GPU. Although the development language of CUDA is also extended C, CUDA has more functions for GPGPU use than other development environments. Therefore, the GPU implementation with CUDA is much easier with previous environments.

2.2 CUDA Environment

In the CUDA environment, CPU, the main memory, is called a "host", and GPU is called a "device". GPU is considered to be a co-processor that can execute many threads in parallel. Fig. 1 shows the hardware model of the device. In Fig. 1, the device has multiple streaming multiprocessors (SMs), and each SM has multiple streaming processors (SPs). The device executes a large amount of threads in parallel on those processors for accelerating the computation.

The CUDA environment employs an architecture called single-instruction, multiple-thread. Threads are grouped into thread blocks. Moreover, threads in a thread block are separated into a warp at every 32 threads, and a single warp is executed in an SM simultaneously. Then, the throughput of GPU calculation decreases if the number of threads in a thread block is not a multiple of 32.

Threads in a thread block can share data through the shared memory on each SM. Threads can also refer to some memory areas: register, constant cache, texture cache, and global memory. The memory on SMs, such as shared memory, can be referred to with almost no latency. On the other hand, the global memory on VRAM causes a latency of approximately 400-600 clock cycles when a thread accesses it, although it can be referred to by all threads. However, the latency can be concealed by executing many threads in sequence.

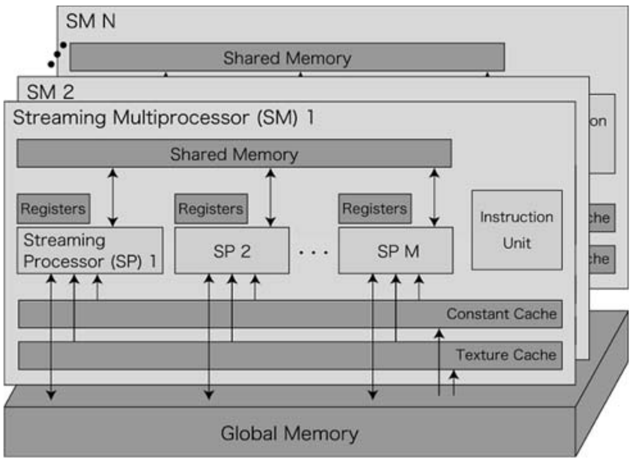


Figure 1 Hardware model of GPU in CUDA

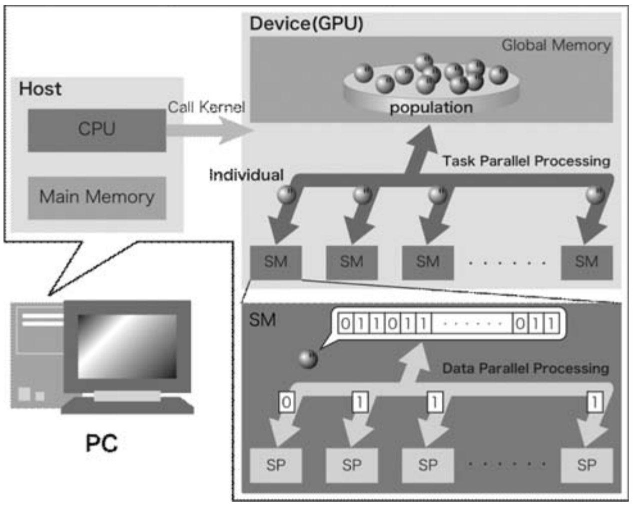


Figure 2 Parallel processing model of GA on GPU

Processes executed by the devices are described as kernel functions, and the devices execute the kernel responding to a call from the host. A kernel function describes the process in a single thread, and the same kernel is executed in many processors. Thus, the device works as single-instruction multiple-data. Note that the functions for executing on the host cannot be called on the device.

3 Implementation Method of GA to GPU

3.1 Parallel Processing Model

When we mount GA on GPU for speeding up calculation, the data transfer between a host and a device should be minimized by executing all operations of evaluation, selection, and reproduction through GPU. In each process, task parallelism is achieved by allocating the operation to each individual in SM. Moreover, operations for the elements of genes are allocated to CUDA cores for data parallelism.

Fig. 2 illustrates the parallel processing model of GA on the GPU environment proposed in this paper. The model allocates an individual to an SM instead of allocating multiple individuals as most related works do. The model allocates each gene of an individual to an SP included in the SM (this method is called data parallelization), considering the CUDA architecture described in Section 2.2. The model enables the acceleration of GA by utilizing many cores and high memory bandwidth, and concealing the latency of memory access by executing many threads sequentially.

3.2 Implementation of GA Operators

As mentioned above, the processes of threads on GPU are described as kernel functions in the CUDA environment. The GA operations of threads are defined as kernel functions, and GPU (the device) executes kernel functions by codes from CPU (the host) side. Data transfer between a host and a device is extremely slower compared with that of on-chip memory, and therefore, to effectively improve speed such transfers must be decreased. In the following sections, the implementation methods of GA operators on GPU are illustrated.

3.2.1 Random Number Generator

Because GA is a type of meta-heuristic algorithm, some processes are executed on a probabilistic basis. However, CUDA libraries do not include any functions of a random number generator (RNG) at present, despite the fact that RNG is naturally necessary for executing GA processes. Then, we generate random numbers using an original kernel of RNG. The kernel is called in each generation and it outputs the array of the random number to the global memory. The other kernels can read random numbers from the array if necessary. In this paper, Xorshift [24] is adopted as RNG in both CPU and GPU computations. Xorshift is so fast and simple that it can be easily implemented to the kernel.

3.2.2 Sorting

The sorting process is executed for the selection process. The sorting kernel sorts the population based on the fitness of the individuals; however, the sorting algorithms generally used in CPU implementation, such as Quicksort and heap sort, are difficult to parallelize on GPU. Furthermore, in the sort process of GA, Quicksort is used in CPU, and Bitonicsort [25, 26] is used in GPU from CUDA SDK. Bitonicsort can be executed only when the element count is a power of 2; thus, it is sorted after adding the dummy data to match the number, that is, the element count a power of 2.

The sorting process is divided into two kernels as illustrated below. The first kernel sorts the index of individuals based on fitness value by using a Bitonic sort algorithm. The first kernel is executed in a block in order to use single shared memory, which can be accessed over 100 times faster than global memory. The kernel must access the same array of indexes multiple times in the sorting process, and thus using shared memory is suitable because the kernel can avoid accessing the low-speed global memory repeatedly (that is, the kernel accesses global memory only twice when it copies the initial array and returns the result). The second kernel exchanges individuals in the population array based on the ranking sorted in the first kernel. The second kernel allocates an individual to a block and allocates a gene to a thread in order to highly parallelize the processes and conceal the latency, as mentioned in Section 3.1. Note that the processing time of sorting for GPU is a sum of both kernels.

In the CPU implementation, a quick sort is used in the experiment. The effect of the difference of the sort algorithm between CPU and GPU versus execution time is discussed in Section 4.4.

3.2.3 Selection

Tournament selection is adopted, and the process is divided into two kernels. The first kernel independently executes a tournament in a thread and writes the index of the champion to global memory. Thus the number of threads of the first kernel is equal to the population size. The second kernel copies the information of a selected individual to the array of population on the next generation. The second kernel allocates an individual to a block and a gene to a thread as the sorting kernel. Note that the processing time of Selection for GPU is a sum of both kernels.

3.2.4 Crossover

The crossover kernel in this paper adopts a one-point crossover. The kernel executes the crossover process with two individuals in a thread, and then, the number of blocks of the kernel is [population size]/2. The kernel checks the crossover point and exchanges the genes of two individuals. The kernel allocates an individual to a block and a gene to a thread as the sorting kernel.

3.2.5 Mutation

The mutation kernel adopts the point mutation. The kernel processes an individual in a block, and the check of the point mutation and the swap of a gene are executed in a thread. The kernel allocates an individual to a block and a gene to a thread as the sorting kernel.

4 Experiment : Test Functions

4.1 Outline for Experiment

To evaluate the basic performance on GPU and CPU, the GPU and CPU computations are compared using eight test functions of the optimization problem shown in Table 1. Their calculation times are different from each other because of their characteristics of functions included, especially whether trigonometric functions are included or not. The dimension of these functions is set on $n = 32$ and $n = 128$.

4.2 Experimental Settings

Tab. 2 shows the parameter settings of GA. The genotypes of individuals are encoded as a binary string of 16 bits per dimension size of the problem. Thus, the gene length is 512 in case of $n = 32$ and 2048 in case of $n = 128$.

Tab. 3 shows the experimental environment. We used a PC that has one Intel Core2Duo E8500 processor and one NVIDIA Geforce GTX280. Although the CPU has two cores, it executes only a single thread in this experiment.

Table 1 Test functions

Function	Expression	Range
Hypersphere	$f_1(x)=\sum_{i=1}^{n} x_i^2$	$-5.12 \le x_i \le 5.12$
Ridge	$f_2(x)=\sum_{i=1}^{n}\left(\sum_{j=1}^{i} x_j\right)^2$	$-100 \le x_i \le 100$
Rosenbrock	$f_3(x)=\sum_{i=1}^{n-1}\left[100(x_1-x_i^2)^2+(1-x_i)^2\right]$	$-2{,}048 \le x_i \le 2{,}048$
Step	$f_4(x)=\sum_{i=1}^{n}\left(\lfloor x_i+0{,}5\rfloor\right)^2$	$-100 \le x_i \le 100$
Griewank	$f_5(x)=\frac{1}{4000}\sum_{i=1}^{n} x_i^2-\prod_{i=1}^{n}\cos\left(\frac{x_i}{\sqrt{i}}\right)+1$	$-2{,}048 \le x_i \le 2{,}048$
Schwefel	$f_6(x)=\sum_{i=1}^{n}-x_i\sin\sqrt{\lvert x_i\rvert}$	$-512 \le x_i \le 512$
Rastrigin	$f_7(x)=\sum_{i=1}^{n}\left\{x_i^2-10\cos(2\pi x_i)+10\right\}$	$-5{,}12 \le x_i \le 5{,}12$
Ackley	$f_8(x)=-20\exp\left(-0{,}2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right)-\exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right)+20+\mathrm{e}$	$-32 \le x_i \le 32$

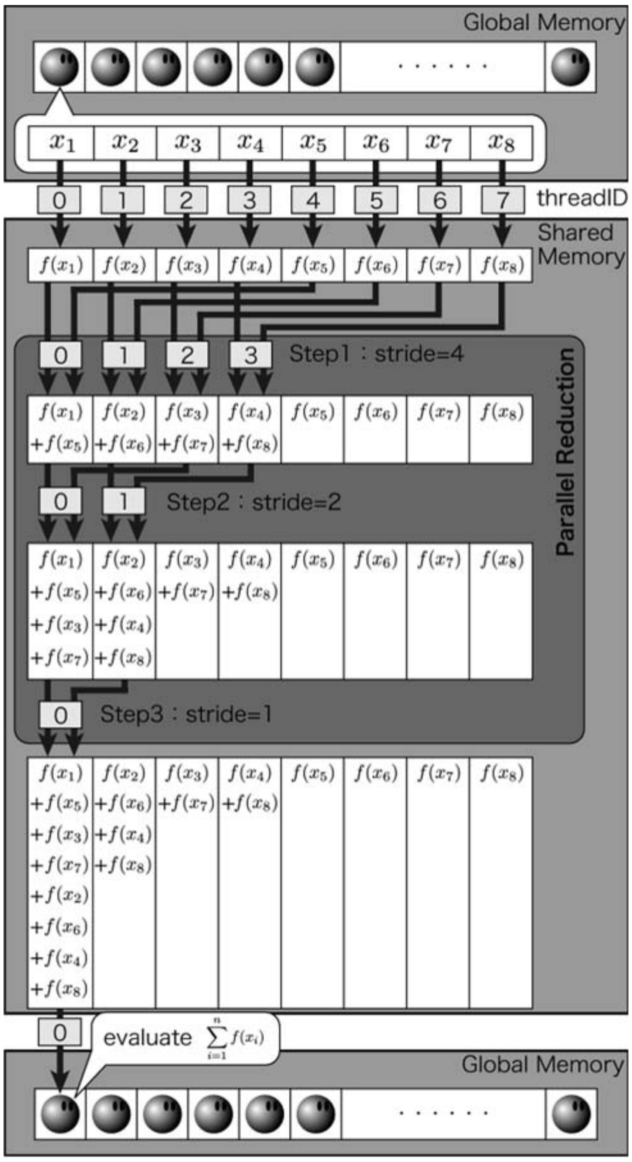


Figure 3 Implementation of evaluation kernel for test functions on GPU

Table 2 GA settings

Population size	256
Selection	tournament selection (size : 2) elite strategy (size : 1)
Crossover	one-point crossover
Crossover rate	1,0
Mutation	bit mutation
Mutation rate	1/512 (*n*=32) 1/2048 (*n*=128)
Generations	5000 (*n*=32) 10000 (*n*=128)

Table 3 Experimental environment

CPU	Intel Core2Duo E8500 (3.16GHz)
GPU	NVIDIA Geforce GTX280 (1.296GHz)
Main Memory	2GB
OS	WindowsXP Home Edition (SP3)
Develop tool	Visual C++ 2005 Express Edition NVIDIA CUDA SDK Ver.2.3

For the CUDA program compilation of the extended part of C++ of the GPU implementation, the nvcc compiler with default settings is used. For the C++ program compilation of the CPU and GPU implementations, Microsoft Visual C++ with the default settings for the release mode of Visual Studio 2005 is used.

4.3 Implementation of Evaluation

Fig. 3 shows the implementation model of the evaluation kernel. The model distributes the evaluation process of an individual to an SM. In addition, the calculation process of each gene is distributed to SPs multiply included in an SM. Accordingly, the number of threads in a block is 32 and 128, corresponding to the dimension of the function.

The execution flow of the evaluation kernel in a block is shown below (Fig. 3):

Table 4 Experimental results

n	function	CPU		GPU		Speedup
		Processing Time / s	stddiv	Processing Time / s	stddiv	Ratio
32	Hypersphere	8,492	0,027	1,111	0,004	7,644
	Ridge	9,073	0,015	1,156	0,001	7,849
	Rosenbrock	8,676	0,035	1,099	0,001	7,894
	Step	8,863	0,006	1,095	0,008	8,094
	Griewank	10,320	0,045	1,134	0,004	9,101
	Schwefel	10,392	0,023	1,116	0,005	9,312
	Rastrigin	9,928	0,022	1,096	0,001	9,058
	Ackley	9,897	0,023	1,123	0,001	8,813
128	Hypersphere	66,154	0,171	4,287	0,001	15,431
	Ridge	86,790	0,024	4,953	0,003	17,523
	Rosenbrock	67,581	0,016	4,275	0,002	15,808
	Step	69,437	0,024	4,252	0,002	16,330
	Griewank	80,737	0,183	4,380	0,002	18,433
	Schwefel	80,033	0,022	4,343	0,004	18,428
	Rastrigin	77,639	0,023	4,274	0,003	18,165
	Ackley	77,230	0,034	4,304	0,002	17,944

1. Each thread loads required variables that corresponds to the allocated dimension (equal to the thread ID) and calculates a part of the function value.
2. Each thread writes the calculated value to shared memory, and all threads are synchronized.
3. The function value is calculated by parallel reduction.
4. Thread 0 calculates the summation of partial solutions, then writes the function value to global memory.

The binary value given from the gene is converted to a decimal value using parallel reduction in the same way as in Fig. 3, before processing the evaluation kernel.

4.4 Results and Discussion

Tab. 4 shows the experimental results of the GPU and CPU implementations. The processing time required for calculating 10 000 generations for each condition is shown in the table. The performance on the table is the average value of 10 trials. GPU with the proposed implementation method yielded a speedup ratio of 7,6-18,4 times compared to the CPU implementation method. It is considered that the GPU implementation can conceal the latency of memory access by executing many threads in parallel, while the CPU implementation executes the GA calculation sequentially. In particular, it is notable that our implementation to parallelize the process of both individuals and their data is more effective, because the implementation enables the execution of more threads than others. In addition, most GA processes are executed on GPU. This can suppress the frequency of data transfer between the host and the device, which is probably the bottleneck to speed up by GPU.

Having discussed the difference between the results of the GPU and CPU implementations, we will now discuss the difference between the functions. The rate of evaluation of the Griewank function against the total computation time is longer than that of the Hypersphere function on CPU. This is considered because the Griewank function contains the trigonometric function that takes 0,03 μs on CPU, although the basic arithmetic operation takes only 0,0025 μs. On the other hand, the GPU kernel can use the fast trigonometric functions in the CUDA library, although it contains a small error (about $2^{-21,44}$ in absolute value maximum in the range $[-\pi, \pi]$). The fast trigonometric function __cosf() takes 0,0043 μs; in contrast, the normal trigonometric function cos() takes 0,0492 μs, and four arithmetic operations take 0,0020 μs. For this reason, the Griewank function on GPU computation is executed nearly as fast as the Hypersphere function, and the speedup ratio increases when evaluating the Griewank function.

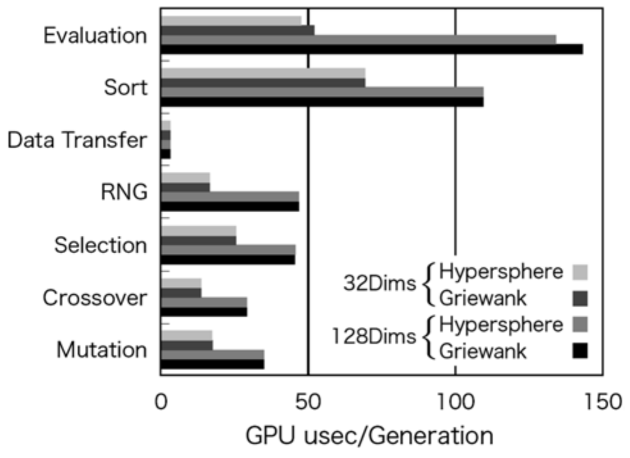
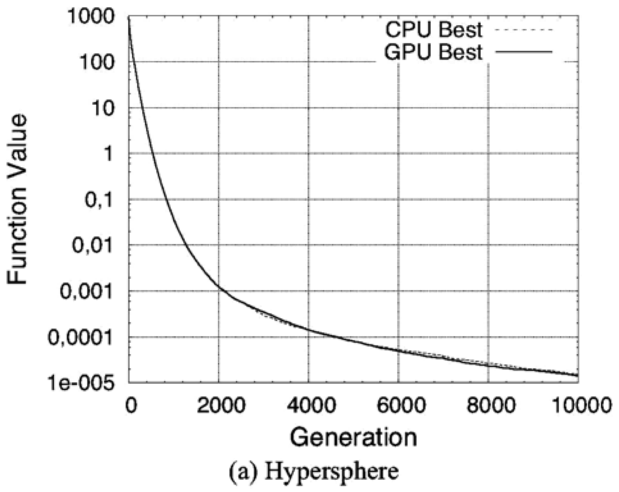


Figure 4 GPU occupancy of GA kernels

Fig. 4 shows the occupation time of each operator of GA. Focusing on the evaluation, the occupation time of the Griewank function is longer than that of the Hypersphere function, as mentioned above. Furthermore, regarding the dimension size, the occupation time of the Griewank function is 51,9 (GPU μs/generation) and that of the Hypersphere function is 47,4 (GPU μs/generation) in cases of $n = 32$, and the occupation time of the Griewank function is 142,9 (GPU μs/generation) and that of the Hypersphere function is 133,9 (GPU μs/generation) in cases of $n = 128$.

These results indicate that the occupation time in cases of *n* = 128 is about 2,8 times higher than that in cases of *n* = 32, nevertheless, the problem scale in cases of *n* = 128 is 4 times greater than that in cases of *n* = 32. This is considered because in case of *n* = 128, a thread can be executed more easily while other threads are accessing memory compared to the case of *n* = 32. In short, the more the number of threads increases as the problem scale is enlarged, the more the latency of memory access decreases.



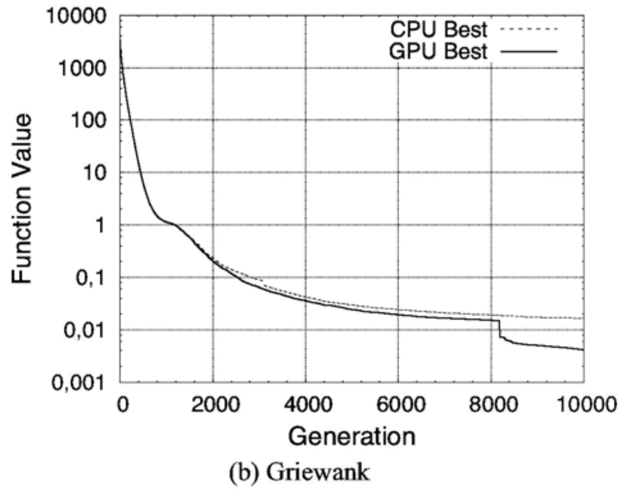


Figure 5 Transition of function value on a 128 dimensional function

Fig. 5 shows the transition of the function value in case of *n* = 128. The same random sequences are used in both CPU and GPU implementations; however, the function values of those are slightly different from each other, because the application order of the random number is different. The transition of the function value shows no negative effect on the quality; thus, it was found that GA with GPU implementation has almost equal search performance to that with the CPU implementation.

As mentioned in Section 3.2.2, different sort algorithms are used for CPU and GPU in the experiment in order to investigate the best performance of each implementation. The processing times of the sorting are as follows: CPU takes 98 μs and GPU takes 173 μs in case of *n* = 32, and CPU takes 259 μs and GPU takes 295 μs in case of *n* = 128. The processing times of the sorting without exchanging the individual information (that is, the processing time of the sort algorithms are simply compared) are as follows: CPU takes 34,5 μs and GPU takes 37,7 μs in case of *n* = 32, and CPU takes 35,6 μs and GPU takes 37,7 μs in case of *n* = 128.

These results clearly show that the difference of processing times between CPU and GPU is much larger in the case that data transfer to/from the Global Memory is executed. From these results, it is considered that exchanging individual information in the second sorting kernel in Section 3.2.2 is a bottleneck, because the kernel must access global memory frequently.

5 Conclusion

In this paper, we proposed an implementation method of GA to GPU using CUDA, adopting parallelization of not only individuals but also the genes by considering the GPU architecture. In terms of the experimental results, the proposed implementation method yielded approximately 18 times faster results than those of a CPU implementation on benchmark tests. The results also showed that the speedup ratio increased as the problem scale increased.

In the future, we plan to adopt some real-world problems to examine the performance of a GPU implementation. Moreover, the Fermi architecture is released in 2010, which supports concurrent kernel execution and improves the performance of double-precision calculation, and so on, is expected to further develop the GPGPU field. We also plan to implement some evolutionary algorithms which can utilize the Fermi architecture, e.g., steady-state Gas.

6 References

[1] Dorigo, M.; Maniezzo, V. Parallel genetic algorithms: Introduction and overview of current research, Parallel Genetic Algorithms: Theory and Applications. // Frontier in Artificial Intelligence and Applications, (1993), pp. 5–42.
[2] Imade, H.; Morishita, R.; Ono, I.; Ono, N.; Okamoto, M. A grid-oriented genetic algorithm framework for bioinformatics, Grid systems for life sciences. // New Generation Computing, 22, 2(2004), pp. 177–186.
[3] Lim, D.; Ong, Y.; Jin, Y.; Sendhoff, B.; Lee, B. Efficient Hierarchical Parallel Genetic Algrorithms using Grid computing. // Future Generation Computer Systems, 23, 4(2007), pp. 658–670.
[4] Matsuoka, S.; Aoki, T.; Endo, T.; Nukada, A.; Kato, T.; Hasegawa, A. GPU accelerated computing from hype to mainstream, the rebirth of vector computing. // Journal of Physics: Conference Series, Vol. 180, No. 1, pp. 0120435, 2009.
[5] Nukada, A.; Ogata, Y.; Endo, T.; Matsuoka, S. Bandwidth Intensive 3-D FFT kernel for GPUs using CUDA. // Proc. ACM/IEEE Supercomputing 2008 (SC2008), pp. 1–11.
[6] Fujimoto, N. Dense Matrix-Vector Multiplication on the CUDA Architecture. // Parallel Processing Letters, 18, 4(2008), pp. 511–530.
[7] Stone, S. S.; Haldar, J. P.; Tsao, S. C.; Hwu, W.-m. W.; Sutton, B. P.; Liang, Z.-P. Accelerating advanced MRI reconstructions on GPUs. // Journal of Parallel and Distributed Computing, 68, 10(2008), pp. 1307–1318.
[8] Preis, T.; Virnau, P.; Paul, W.; Schneider, J. J. Accelerated fluctuation analysis by graphic cards and complex pattern formation in financial markets. // New Journal of Physics, 11, 9(2009), pp. 1–21.
[9] Banzhaf, W.; Harding, S.; Langdon, W. B.; Wilson, G. Accelerating Genetic Programming Through Graphics Processing Units. // Genetic and Evolutionary Computation, (2009), pp. 1–19.

[10] Harding, S.; Banzhaf, W. Fast Genetic Programming and Artificial Developmental Systems on GPUs. // Proceedings of the 21st International Symposium on High Performance Computing and Applications (HPCS'07), 2007, pp. 2.
[11] Langdon, W. B.; Banzhaf, W. GP on SPMD parallel graphics hardware for mega Bioinformatics data mining. // Soft Computing, 12, 12(2008), pp. 1169–1183.
[12] Langdon, W. B.; Banzhaf, W. A SIMD interpreter for Genetic Programming on GPU Graphics Cards. // Proceedings of the 11th European Conference on Genetic Programming (EuroGP 2008), 4971, (2008), pp. 73–85.
[13] Robilliard, D.; Marion-Poty, V.; Fonlupt, C. Population Parallel GP on the G80 GPU. // Lecture Notes in Computer Science, 4971, (2008), pp. 98–109.
[14] Pospichal, P.; Jaros, J. GPU-based Acceleration of the Genetic Algorithm. // Proceedings of the GECCO 2009 Workshop on Computational Intelligence on Consumer Games and Graphic Hardware (CIGPU-2009), 2009.
[15] Tsutsui, S.; Fujimoto, N. Solving Quadratic Assignment Problems by Genetic Algorithms with GPU Computation: A Case Study. // Proceedings of the GECCO 2009 Workshop on Computational Intelligence on Consumer Games and Graphic Hardware (CIGPU-2009), 2009, pp. 2523–2530.
[16] Tsutsui, S.; Fujimoto, N. An Analytical Study of GPU Computation for Solving QAPs by Parallel Evolutionary Computation with Independent Run. // Proceedings of IEEE World Congress on Computational Intelligence (WCCI 2010), 2010, pp. 889–896.
[17] Debattisti, S.; Marlat, N.; Mussi, L.; Cagnoni, S. Implementation of a Simple Genetic Algorithm within the CUDA Architecture. // Proceedings of the GECCO 2009 Workshop on Computational Intelligence on Consumer Games and Graphic Hardware (CIGPU-2009), 2009.
[18] Mark, W. R.; Glanville, R. S.; Akeley, K.; Kilgard, M. J. Cg: A System for programming graphics hardware in a C-like language. // ACM Transaction on Graphics (Proceedings of SIGGRAPH 2003), 22, 3(2003), pp. 867–907.
[19] Microsoft, High-Level Shading Language, http://msdn.microsoft.com/en-us/library/bb509561(v=vs.85).aspx, 2004.
[20] Kessenich, J.; Baldwin, D.; Rost, R. The OpenGL Shading Language version 1.10.59, http://www.opengl.org/documentation/oglsl.html, 2004.
[21] Mark, W. R.; Glanville, R. S.; Akeley, K.; Kilgard, M. J. Cg: A System for programming graphics hardware in a C-like language. // ACM Transaction on Graphics (Proceedings of SIGGRAPH 2003), 22, 3(2003), pp. 867-907.
[22] Buck, I.; Foley, T.; Horn, D.; Sugerman, J.; Fatahalian, K.; Houston, M.; Hanrahan, P. Brook for GPUs: Stream Computing on Graphics Hardware. // SIGGRAPH'04, ACM Transactions on Graphics, 23, 3(2004), pp. 777–786.
[23] NVIDIA Corporation, CUDA Zone, http://www.nvidia.com/object/cuda home new.html, 2007.
[24] Marsaglia, G. Xorshift RNGs. // Journal of Statistical Software, 8, 14(2003), pp. 1–6.
[25] Batcher, K. Sorting networks and their applications. // Proceedings of the AFIPS Spring Joint Computing Conference, 32, (1968), pp. 307–314.
[26] Blelloch, G. E.; Leiserson, C. E.; Maggs, B. M.; Plaxton, C. G.; Smith, S. J.; Zagha, M. An Experimental Analysis of Parallel Sorting Algorithms. // Theory of Computing Systems, 31, 2(1998), pp. 135-167.

Authors' addresses

Masashi Oiso
Graduate School of Engineering
Hiroshima University
1-4-1 Kagamiyama, Higashi-hiroshima
Hiroshima, 739-8527, Japan
phone: +81-90-4107-3129
e-mail: oiso@ohk.hiroshima-u.ac.jp

Yoshiyuki Matsumura
Faculty of Textile
Shinshu University
3-15-1 Tokida, Ueda
Nagano, 386-0018, Japan

Toshiyuki Yasuda
Graduate School of Engineering
Hiroshima University
1-4-1 Kagamiyama, Higashi-hiroshima
Hiroshima, 739-8527, Japan

Kazuhiro Ohkura
Graduate School of Engineering
Hiroshima University
1-4-1 Kagamiyama, Higashi-hiroshima
Hiroshima, 739-8527, Japan