Accelerating Steady-State Genetic Algorithms based on CUDA Architecture

Masashi Oiso
Toshiyuki Yasuda
and Kazuhiro Ohkura
Graduate School of Engineering
Hiroshima University
Hiroshima, Japan
Email: oiso@ohk.hiroshima-u.ac.jp

Yoshiyuki Matumura
Faculty of Textile
Shinshu University
Ueda, Japan
Email: matsumu@shinshu-u.ac.jp

Abstract—**Parallel processing using graphic processing units (GPUs) have attracted much research interest in recent years. Parallel computation can be applied to genetic algorithms (GAs) in terms of the processes of individuals in a population. This paper describes the implementation of GAs in the compute unified device architecture (CUDA) environment. CUDA is a general-purpose computation environment for GPUs. The major characteristic of this study is that a steady-state GA is implemented on a GPU based on concurrent kernel execution. The proposed implementation is evaluated through four test functions; we find that the proposed implementation method is 3.0-6.0 times faster than the corresponding CPU implementation.**

I. 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 usually evaluates all solution candidates in a population for every generation. To overcome this disadvantage, parallel processing such as that in the form of cluster computing [7], has been utilized. In addition, in the late 1990s, grid computing, which enables high-performance computing by connecting computational resources through the Internet, began applying EC to high-speed networks [11], [15]. In this manner, 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 the advantage in using GPUs is that they possess an architecture that is specialized for graphics applications, and are good at processing calculations of a simple and repetitive nature on 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 and dependencies. However, the performance of GPUs has been improving rapidly in recent years, such that their peak performance is much higher than that of CPUs. Another point to be noted is that GPUs have a higher power-to-watt ratio than CPUs. The concept of using GPUs not only for graphics applications but also for general-purpose applications, is called general-purpose computation on GPUs (GPGPU) [8], [18], [21], [26], [30]. 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 [2], [10], [13], [14], [27], quickly followed by GA researchers. Pospichal and Jaros [25] adopted the island model for implementation with the conditions of 64 islands and a population size of 256 on each island, and achieved a speedup rate of 2,602 compared to the CPU implementation. Tsutsui and Fujimoto [32] proposed the implementation of a distributed GA similar to the island model for a population size of 11,520 in an experiment on quadratic assignment problems; they achieved a speedup rate 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 in GPU calculation [33]. Debattisti et al. [6] implemented the entire process of a simple GA on a 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 a CPU. We have also previously proposed an implementation method of simple GA using data parallelization method [24], and achieved a speedup rate of 7.6-23.0 with a small population size of 256 compared to sequential execution on a CPU.

In these previous studies, only generational GAs were discussed for implementation on GPUs. On the other hand, steady-state GAs, which have another type of generation alternation model, have not been focused upon; nevertheless, steady-state GAs perform well in solving various optimization problems [29], [23]. Thus, in this paper, an implementation method for steady-state GA on GPUs is proposed based on concurrent kernel execution.

The rest of this paper is organized as follows. Section 2 describes the generation alternation models of GAs. Section 3 describes GPGPU development environments. Section 4 presents the proposed implementation method of steady-state

GA on a GPU. In Section 5, experiments are conducted on function optimization problems to examine the performance of the proposed implementation. Finally, Section 6 presents the conclusions of this paper.

II. Generation Alternation Models

Generation alternation models are very influential in the convergence and diversity of the population in GAs. Thus, it is reasonable to suppose that these models are important as genetic operations, e.g., crossover and mutation to improve the search performace. Generation alternation models are generally divided into two categories: generational models and steady-state models. Most GAs adopt the generational model, wherein all individuals in a population evolve under the selection pressure in every generation. The others adopt the steady-state model, wherein, in contrast, a few individuals are processed and replaced in a generation.

A good balance of exploration and exploitation of problems basically depends on their fitness landscapes, and according to the no-free-lunch theorem, there is no ideal generation alternation model. However, steady-state models are generally better than generational models in two respects. The first is that most steady-state models perform the selection for survival quite locally; therefore, the diversity of the population is robustly maintained. This feature is good for globally multimodal function optimization. The second is that steady-state models are easily applicable to parallel processing, as mentioned in Section 1; this can be attributed to the fact that these models do not require the synchronization with all individuals in the population, especially in the evaluation process. This feature could be effective even for implementation in GPU computation owing to the characteristics of GPU architecture described in Section 3.2.

In the following, we introduce two typical steady-state generation alternation models for implementation on GPUs.

- Steady-state GA (SS)[31]
  This is a classic steady-state model.
  - Selection for reproduction
    Select two individuals as parents from the population by ranking selection (select proportionally to ranking of fitness value).
  - Selection for survival
    Remove the two worst individuals from the population.
    Then add two offspring to the population.
- Minimal Generation Gap (MGG)[28]
  This model shows better performance in terms of its exploration performance as compared to five traditional models; this was ascertained by using a few test functions and traveling salesman problems in [28].
  - Selection for reproduction
    Remove two individuals as parents from the population at random.
  - Selection for survival
    Select the best individual from the family (both parents and offspring).
    Select an individual by roulette selection. Return them to the population.

The selection methods of the SS model, which depends on the ranking of the population based on the fitness value, are not suitable because the operations to sort the population require expensive computation compared to other GA processes[24], which could lead to a possible bottleneck. Therefore, this paper adopts the MGG model for the experiment described in Section V.

III. GPGPU Development Environment

A. Transition of Development Environment

Around 1999, GPUs gained popularity and a specialized architecture was developed to execute a fixed function pipeline. With the installation of a programmable shader in the GPU, the degree of freedom of GPU calculation improved dramatically, and therefore, GPUs have since been used for general-purpose applications as GPGPU.

In the initial stages of GPGPU, programming using low-level assembly languages was indispensable; thus, efforts were focused on the implementation rather than the design of shader algorithms. Then, graphics shading languages such as C for graphics (Cg) [16] by NVIDIA (2002), High Level Shading Language (HLSL) [20] by Microsoft, and OpenGL Shading Language (GLSL) [12] were released, and GPGPU programming became easier. However, an in-depth understanding of GPU architecture and graphics processing was still required. As a result, GPGPU was reserved for graphics programming experts. Therefore, the development of high-level languages is essential.

Since 2004, some development environments have been proposed. Sh [19] was developed at Waterloo University and consists of a C++ library. Brook [5], which was developed at Stanford University, is based on C and was extended to handle stream data.

In particular, Compute Unified Device Architecture (CUDA) [22] was released by NVIDIA as a development environment for their GPU. CUDA also uses the extended C language, and has many GPGPU functions. Currently, the Fermi architecture released in 2010 supports concurrent kernel execution. In this paper, we focus on concurrent kernel execution, which enables the parallel execution of generational processes in a steady-state model.

B. CUDA Environment

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

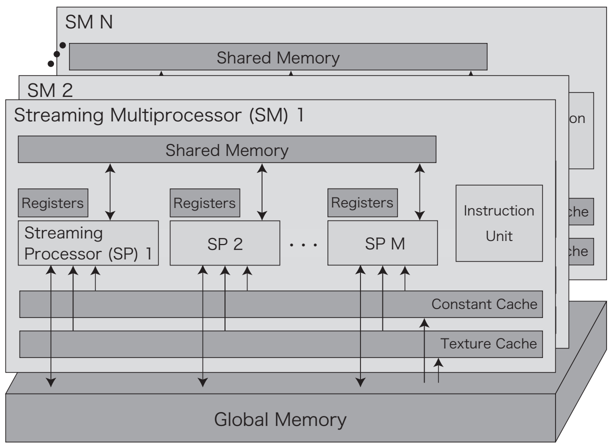


Fig. 1. Hardware model of GPU in CUDA.

threads in parallel using these processors to accelerate the computation.

The CUDA environment employs the single-instruction, multiple-thread architecture. Threads are grouped into thread blocks. Moreover, threads in a thread block are separated into a warp after 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 in each SM. Threads can also refer to some memory areas such as registers, constant cache, texture cache, and global memory. The memory on SMs such as shared memory can be referred to with almost no latency. In contrast, the global memory on video random access memory causes a latency of approximately 400-600 clock cycles when a thread accesses it, but this memory can be referred to by all threads. However, the latency can be concealed by sequentially executing multiple threads.

Processes executed by the devices are described as kernel functions, and the devices execute the kernel in response 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 can not be called on the device.

IV. Implementation Method of steady-state GA

A. Parallel Processing Model

Figure 2 illustrates the parallel processing model of a steady-state GA in the GPU environment proposed in this paper. The proposed implementation executes all genetic operations in a generation of the MGG model in a single kernel function. First, by a kernel function call from the host, an SM receives two individuals (parents) from the population in the global memory. Then, all processes such as *random number generator*, *crossover*, *mutation*, *sorting*, and *selection*, illustrated in Section IV-B, are executed in the SM. Finally, the two selected individuals are sent back to the global memory, and the routine is immediately repeated until the termination criterion is satisfied. Moreover, the processes of the kernel function are executed in parallel in some streams (eight streams in this paper).

In our previous work, we developed an implementation method of generational GAs effective for a small population described in [24]. In doing so, we used the same acceleration method that employs data parallelization. In the following sections, we present the details of the processes in the kernel function.

B. Processes of GA Kernel Functions

As mentioned in Section III-B, 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 the GPU (the device) executes kernel functions by codes from the CPU (the host) side. Data transfer between a host and a device is extremely slow as compared to data transfer on on-chip memory; therefore, to effectively improve speed, such transfers must be decreased. In the following sections, the implementation methods of GA operators on GPUs are illustrated. Note that the kernel allocates an operation of a gene to a thread in order to highly parallelize the operations and conceal the latency in the processes, as mentioned in Section IV-A.

1) Random Number Generator (RNG): Because CUDA libraries do not include random number generator functions, the kernel has to generate random numbers using an original process. The process operates the random seed array in the global memory in order to use the same random seed in a trial, and outputs an array of random numbers to the shared

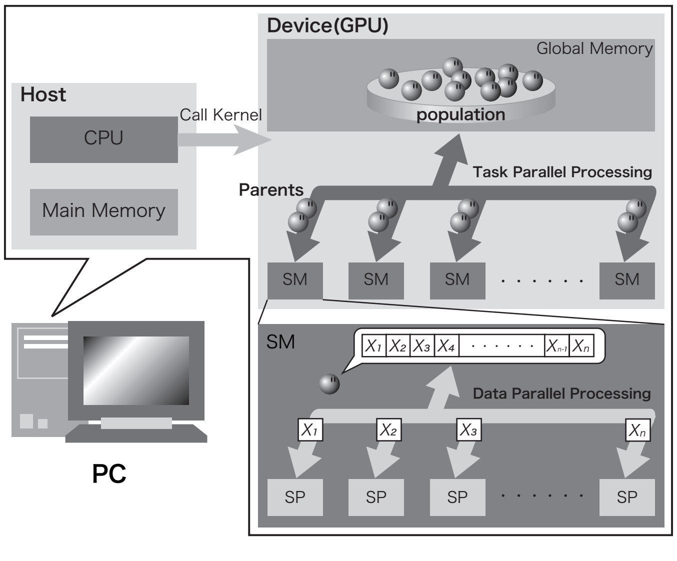


Fig. 2. Parallel processing model of steady-state GA on GPU.

TABLE I
TEST FUNCTIONS.

Function	Expression	Range
Hypersphere	$f_1(x) = \sum_{i=1}^{n} x_i^2$	$-5.12 \leq x_i \leq 5.12$
Rosenbrock	$f_3(x) = \sum_{i=1}^{n-1} [100(x_1 - x_i^2)^2 + (1 - x_i)^2]$	$-2.048 \leq x_i \leq 2.048$
Ackley	$f_8(x) = -20 \exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n} \cos 2\pi x_i\right) + 20 + e$	$-32 \leq x_i \leq 32$
Griewank	$f_5(x) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} cos(\frac{x_i}{\sqrt{i}}) + 1$	$-2.048 \leq x_i \leq 2.048$

memory. In this paper, Xorshift [17] is adopted as the RNG in both CPU and GPU computations.

2) Crossover: The *crossover* process in this paper adopts BLX-α as a blend crossover. The process executes the crossover operation with two parents in an SM, and then the two offspring yielded by the process are stored in the shared memory.

3) Mutation: In this study, the *mutation* process adopts an uniform mutation. The processes operates offspring in turn, and the verification of the mutation and the swapping of genes are executed in a thread. The kernel allocates an individual to a block and a gene to a thread as the sorting kernel.

4) Sorting: The *sorting* process 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 a GPU. Furthermore, in the GA's sort process, Quicksort is used on a CPU, and Bitonicsort [1] [4] is used on a GPU from CUDA SDK. Bitonicsort can be executed only when the element count is 2^n; thus, it is sorted after adding the dummy data to match the number, that is, the element count 2^n. The sorting process is executed on the shared memory, and the results are also stored in the shared memory.

5) Selection: The *selection* process is executed described in Section II by referring to the index from the sorting process. After two individuals are selected, the data array of parents in the global memory is replaced by the selected individuals.

V. EXPERIMENT I: TEST FUNCTIONS

A. Outline for Experiment

To evaluate the basic performance of steady-state GA on GPU and CPU, the GPU and CPU computations are compared using four test functions of the optimization problem shown in Table I. In addition, generational GA on GPU with implementation method in [24] is also compared. For the selection of generational GA, tournament selection with elite preservation is applied. The tournament size is two, and the elite size is one. The dimension of these functions is set to $n = 128$.

B. Experimental Settings

Table II shows the parameter settings of the GA. The genotypes of individuals are real encoded; thus, gene length is equal to the dimension size of the problem.

Table III shows the experimental environment. We used a PC with an Intel Core i7 processor and an NVIDIA Geforce GTX480. Although the CPU has four cores, it executes only a single thread in both CPU and GPU runs in this experiment. For the CUDA program compilation of the extended part of C++ of the GPU implementation, an nvcc compiler is used with its default settings. For the C++ program compilation of the CPU and GPU implementations, Microsoft Visual C++ is used with the default settings of the release mode of Visual Studio 2008.

C. Implementation of Evaluation

Figure 3 shows the implementation of the *evaluation* process. The offspring yielded in an SM are in turn operated in the SM. The operation of each gene is distributed to SPs included in the SM. Accordingly, the number of threads in the SM is 128, corresponding to the dimension of the function.

The execution flow of the *evaluation* process in an SM is shown below (Figure 3):

1) Each thread loads a variable 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 the shared memory, and all threads are synchronized.
3) The function value is calculated by parallel reduction.
4) Thread 0 writes the function value to the shared memory.

D. Experimental Results

Table V shows the experimental results of steady-state GA with CPU and GPU implementations, and generational GA with GPU implementation. The performance summarized in the table is the average value of 10 trials. First, let us see the GPU implementation:CPU implementation speedup ratios for each generation alternation model. The GPU with the proposed implementation method yielded a speedup ratio of 3.0-6.0.

TABLE II
GA SETTINGS.

Population size	256
Crossover	BLX-α
Crossover rate	1.0
α	0.5
Mutation	uniform mutation
Mutation rate	1/128
Generations	10000

TABLE III
EXPERIMENTAL ENVIRONMENT.

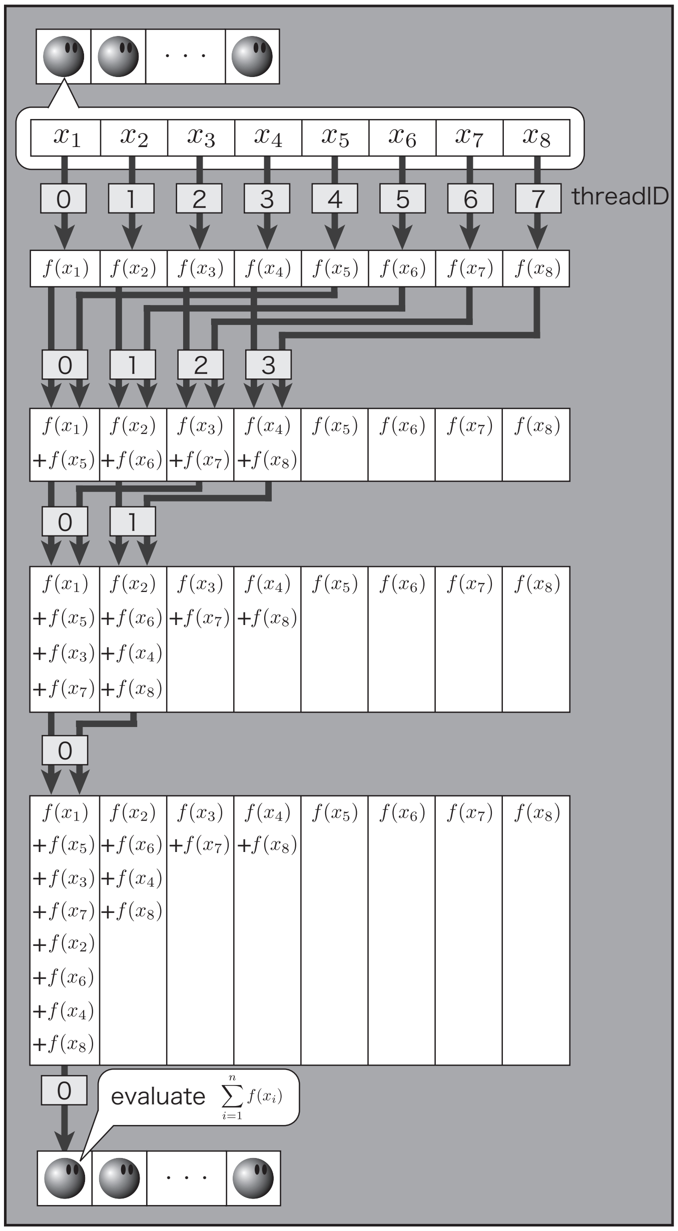
CPU	Intel Corei7 (2.8GHz)
GPU	NVIDIA Geforce GTX480 (1.401GHz)
Main Memory	6GB
OS	Windows7 Enterprise
Develop tool	Visual C++ 2008 Express Edition NVIDIA CUDA SDK Ver.3.2

Then, focusing on the difference in generation alternation models, the ratios of steady-state GA on GPU are not as good as those of the generational GA on GPU. This is because the computational granularity is so small that the latency caused between kernel calls occupies a large amount of the execution time. In fact, steady-state GA calls 128 kernels per 256 offspring (per generation of generational GA), although generational GA calls only 10 kernels. The effect of the latency is serious with such a small computational granularity; thus, we have to improve the implementation for this case.

However, the function values obtained by steady-state GA are significantly better than those of generational GA. In the experiment, steady-state GA could obtain better solutions than generational GA if the experiment was executed during the same period. In the future, we will try to adopt other problems that have different characteristics, and examine the performance of our implementation in detail.

VI. CONCLUSION

In this paper, we proposed an implementation method of steady-state GA on a GPU using CUDA, and adopting data parallelization in consideration of the GPU architecture. In terms of the experimental results, the proposed implementation method yielded approximately 6 times faster results than those of a CPU implementation on benchmark tests. The results also showed that the steady-state model could improve search performance in some problems. In the future, we will conduct experiments in which we adopt different tasks that have nonuniform computational granularity, which would cause some problems in the parallel processing of generational GA. Moreover, we will improve steady-state models to make them more suited to the GPU environment.



x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
0 1 2 3 4 5 6 7 threadID
$f(x_1)$ $f(x_2)$ $f(x_3)$ $f(x_4)$ $f(x_5)$ $f(x_6)$ $f(x_7)$ $f(x_8)$
0 1 2 3
$+f(x_5)$ $+f(x_6)$ $+f(x_7)$ $+f(x_8)$
0 1
$+f(x_3)$ $+f(x_4)$ $+f(x_7)$ $+f(x_8)$
0
$+f(x_2)$ $+f(x_6)$ $+f(x_4)$ $+f(x_8)$
0
evaluate $\sum_{i=1}^{n} f(x_i)$

Fig. 3. Implementation of evaluation kernel for test functions on GPU.

REFERENCES

[1] Batcher, K., ‘Sorting networks and their applications’, *Proceedings of the AFIPS Spring Joint Computing Conference*, Vol. 32, pp.307–314, 1968.

[2] Banzhaf, W., Harding, S., Langdon, W. B. and Wilson, G., ‘Accelerating Genetic Programming Through Graphics Processing Units’, *Genetic and Evolutionary Computation*, pp.1–19, 2009.

[3] Beer, R., ‘Toward the Evolution of Dynamical Neural Networks for Minimally Cognitive Behavior’, *From Animals to Animats 4, Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, pp.421–429, 1996.

[4] Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J. and Zagha, M., ‘An Experimental Analysis of Parallel Sorting Algorithms’, *Theory of Computing Systems*, Vol. 31, No. 2, 1998.

[5] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M. and Hanrahan, P., ‘Brook for GPUs: Stream Computing on Graphics Hardware’, *SIGGRAPH ’04, ACM Transactions on Graphics*, Vol. 23, No. 3, pp.777–786, 2004.

[6] Debattisti, S., Marlat, N., Mussi, L. and Cagnoni, S., ‘Implementation of a Simple Genetic Algorithm within the CUDA Architecture’, *Proceed-*

TABLE IV
EXPERIMENTAL RESULTS OF STEADY-STATE GA WITH CPU AND GPU IMPLEMENTATION, AND GENERATIONAL GA WITH GPU IMPLEMENTATION.

n	function	CPU		GPU				Speedup Ratio	
		Processing Time (sec)	stddev	Processing Time (sec)		stddev		steady-state	generational
				steady-state	generational	steady-state	generational		
128	Hypersphere	14.46	0.010	4.874	2.571	0.151	0.013	3.0	5.6
	Rosenbrock	14.59	0.016	4.776	2.644	0.024	0.038	3.1	5.5
	Ackley	22.02	0.027	4.476	2.640	0.032	0.015	4.9	8.3
	Griewank	28.56	0.003	4.780	2.615	0.037	0.024	6.0	10.9

TABLE V
COMPARISON OF FUNCTION VALUES OF STEADY-STATE GA AND GENERATIONAL GA.

n	function	steady-state		generational	
		function value	stddev	function value	stddev
128	Hypersphere	43.95	6.80	122.72	8.46
	Rosenbrock	533.28	316.36	1888.29	288.65
	Ackley	10.85	0.50	16.23	0.29
	Griewank	159.17	26.61	442.39	36.26

ings of the GECCO 2009 Workshop on Computational Intelligence on Consumer Games and Graphic Hardware (CIGPU-2009), 2009.

[7] Dorigo, M. and Maniezzo, V., ‘Parallel genetic algorithms: Introduction and overview of current research’, *Parallel Genetic Algorithms: Theory and Applications, Frontier in Artificial Intelligence and Applications*, pp.5–42, 1993.

[8] Fujimoto, N., ‘Dense Matrix-Vector Multiplication on the CUDA Architecture’, *Parallel Processing Letters*, Vol. 18, No. 4, pp.511–530, 2008.

[9] Goldberg, D.E., ‘Genetic Algorithms in Search’, *Optimization and Machine Learning*, 1989.

[10] Harding, S. and 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)*, pp.2, 2007.

[11] Imade, H., Morishita, R., Ono, I., Ono, N. and Okamoto, M., ‘A grid-oriented genetic algorithm framework for bioinformatics’, *Grid systems for life sciences, New Generation Computing*, Vol. 22, No. 2, pp.177–186, 2004.

[12] Kessenich, J., Baldwin, D. and Rost, R., ‘The OpenGL Shading Language version 1.10.59’, http://www.opengl.org/documentation/oglsl.html, 2004.

[13] Langdon, W.B. and Banzhaf, W., ‘GP on SPMD parallel graphics hardware for mega Bioinformatics data mining’, *Soft Computing*, Vol. 12, pp.1169–1183, 2008.

[14] Langdon, W.B. and Banzhaf, W., ‘A SIMD interpreter for Genetic Programming on GPU Graphics Cards’, *Proceedings of the 11th European Conference on Genetic Programming (EuroGP 2008)*, Vol. 4971, pp.73–85, 2008.

[15] Lim, D., Ong, Y., Jin, Y., Sendhoff, B. and Lee, B., ‘Efficient Hierarchical Parallel Genetic Algorithms using Grid computing’, *Future Generation Computer Systems*, Vol. 23, Issue 4, pp.658–670, 2007.

[16] Mark, W.R., Glanville, R.S., Akeley, K. and Kilgard, M.J., ‘Cg: A System for programming graphics hardware in a C-like language’, *ACM Transaction on Graphics (Proceedings of SIGGRAPH 2003)*, Vol. 22, No. 3, pp.867–907, 2003.

[17] Marsaglia, G., ‘Xorshift RNGs’, *Journal of Statistical Software*, Vol. 8, Issue 14, pp.1–6, 2003.

[18] Matsuoka, S., Aoki, T., Endo, T., Nukada, A., Kato, T. and 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.

[19] McCool, M., Toit, S.D., Popa, T., Chan, B. and Moule, K., ‘Shader algebra’, *SIGGRAPH ’04, ACM Transactions on Graphics*, Vol. 23, No. 3, pp.787–795, 2004.

[20] Microsoft, ‘High-Level Shading Language’, http://msdn.microsoft.com/en-us/library/ee418149(VS.85).aspx, 2004.

[21] Nukada, A., Ogata, Y., Endo, T. and Matsuoka S., ‘Bandwidth Intensive 3-D FFT kernel for GPUs using CUDA’, *Proc. ACM/IEEE Supercomputing 2008 (SC2008)*, pp.1–11, 2008.

[22] NVIDIA Corporation, ‘CUDA Zone’, http://www.nvidia.com/object/cuda_home_new.html, 2007.

[23] Oiso, M., Matsumura, Y., Yasuda, T., and Ohkura, K., ‘Evaluation of Generation Alternation Models in Evolutionary Robotics’, 4th International Workshop on Natural Computing (IWNC 2009), pp.236-243, 2009.

[24] Oiso, M., Matsumura, Y., Yasuda, T., and Ohkura, K., ‘Implementation Method of Genetic Algorithms to CUDA Environment using Data Parallelization’, *Proceedings of the 14th Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES ’10)*, pp.51–60, 2010.

[25] Pospichal, P. and 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.

[26] Preis, T., Virnau, P., Paul, W. and Schneider, J.J., ‘Accelerated fluctuation analysis by graphic cards and complex pattern formation in financial markets’, *New Journal of Physics*, Vol. 11, pp.1–21, 2009.

[27] Robilliard, D., Marion-Poty, V. and Fonlupt, C., ‘Population Parallel GP on the G80 GPU’, *Lecture Notes in Computer Science*, Vol. 4971, pp.98–109, 2008.

[28] Sato, H., Yamamura, Y., and Kobayashi, S., ‘Minimal Generation Gap Model for GAs Considering Both Exploration and Exploitation’, *Proceedings of 4th International Conference on Soft Computing (IIZUKA ’96)*, pp.494–497, 1996.

[29] Sato, H., Ono, I., and Kobayashi, S., ‘A new generation alternation model of genetic algorithms and its assessment’, *Journal of Japanese Society for Artificial Intelligence*, Vol. 12, No. 5, pp.734–744, 1997.

[30] Stone, S.S., Haldar, J.P., Tsao, S.C., Hwu, W.-m.W., Sutton, B.P. and Liang, Z.-P., ‘Accelerating advanced MRI reconstructions on GPUs’, *Journal of Parallel and Distributed Computing*, Vol. 68, Issue 10, pp.1307–1318, 2008.

[31] Syswerda, G., ‘Uniform Crossover in Genetic Algorithms’, *Proceedings of the third international conference on Genetic algorithms*, pp.2-9, 1989.

[32] Tsutsui, S. and 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)*, pp.2523–2530, 2009.

[33] Tsutsui, S. and 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)*, pp.889–896, 2010.