An Implementation and Evaluation of CUDA-based GPGPU Framework by Genetic Algorithms

Masato Yoshimi[†] and Yuki Kurano[†] and Mitsunori Miki[†] and Tomoyuki Hiroyasu[††],

[†]Faculty of Science and Engineering, Doshisha University, Kyoto, Japan
[††]Faculty of Department of life and Medical Science, Doshisha University, Kyoto, Japan

Summary
Graphic Processing Unit (GPU), which was traditionally used for image processing, has been widely applied to general computation called GPGPU. Nowadays, a lot of studies using GPUs are progressing and various products are being developed. GPU has many processor cores, and thereby has low power consumption per unit volume. Even several developing environments are already provided, software developing cost remains high, due to the art of programing and a technical knowledge required for the implementation of GPGPU program of the target algorithm exploiting parallelism requires not only realization of the target algorithm, but also knowledge of architecture such as memory hierarchy. In this paper, we propose a framework which enables easy implementation of parallel computing on GPU. This framework can popularize GPU programming. We confirm that we are able to do parallel computing on this framework by implementing and evaluating simple genetic algorithms (SGA). We discuss the relationship between computational speed and execution condition.

Key words:
GPGPU, CUDA, Parallel Computing, Genetic Algorithm

1. Introduction

Performance of computer has increased according to improvement of operating frequency of processor cores in CPU. However, as the power wall suppresses the advancement of operating frequency, the primary factor in performance advancement turns to parallel computing with many-core processor. On the other hand, GPUs (which stands for Graphical Processing Units) have widely been used in image processing. GPU has hundreds of tiny processing cores exploiting parallelism from graphics operation. In recent years, a lot of researchers and developers utilize many cores in GPU as an accelerator of their own software other than graphics computing, and such way of utilization of GPU is called GPGPU (General Purpose computing on GPU). In high performance computing, several supercomputers adopt GPU focusing on its potential. Three machines in the top five of top500 ranking embed GPU to achieve higher computing performance with low-energy consumption.

Program code of GPGPU is written in Shading Language. Shading Language requires the knowledge of graphics operation to write general computation program. In recent years, several extended languages of C language such as CUDA and OpenCL have reduced development cost for GPGPU. However, implementation of GPGPU still requires developers to learn specialized knowledge about GPU architecture including memory hierarchy and structure of processing core in order to achieve efficient parallel computation.

To popularize GPGPU, easy way to utilize GPU may be required to shorten the developing period. Most beginners of GPU may struggle in two difficulties; (1) allocating memories in GPU and (2) overlapping multiple GPU functions and data-transfer. A way to lighten these problems may be a framework to wrap allocating memories and data-transfer. The framework enables Developers to focus on implementation of their GPU functions and applications excluding complicated problems around memories on GPU.

This paper proposes a framework, which packs data-transfer and computation on GPUs. The implementations of Simple Genetic Algorithm are also evaluated and discussed with several parameters, such as the size of the problem, parallel granularity, or the number of GPU, as a case study of an application on the framework.

2. CUDA

2.1 The implementation of GPU program code by CUDA

The framework proposed in this paper supports the GPGPU by CUDA, which is an integrated development environment provided by NVIDIA. CUDA enables a machine equips NVIDIA's GPU to be a parallel computer. CUDA can be written as an extension of C language with several controlling functions such as memory allocation of the GPU device and data-transfer between host and GPU.

Fig. 1 shows an operating flow to utilize GPU device. CUDA divides operations in hosts and device obviously. Only particular function called kernel function is executed on GPU device and other functions are operated on the host. Operating flow to control the device is as the follows;

Manuscript received December 5, 2010
Manuscript revised December 20, 2010

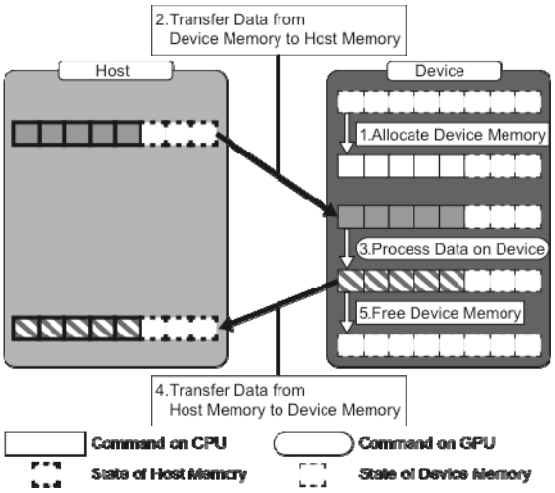

Fig. 1 The order of Job off-load.

(1) The host allocates the memory in the device,
(2) transfers the data from the memory in the host to the memory in the device,
(3) indicates the device to compute the data through executing kernel function,
(4) writes back the computing result from the memory in the device to the memory in the host, and
(5) frees the memory in the device.

Operations in the device and the host can be executed in parallel. When the transferring function from device to host is called, host waits transferring computing result until completion of the kernel function. The feature improves operating efficiency and maintains data consistency.

2.2 Architecture of GPU device and computing resources

Architecture of GPU consists of processing element and memory to store the data used in computation. GPU is regarded as the integration of several resources to execute parallel computing.

GPU consists of multiple Streaming Multiprocessors, which is abbreviated as MP. Each MP includes multiple Streaming Processors, which is abbreviated as SP. CUDA manages these computing resources as three types of units; *grid*, *block* and *thread*. *grid* and *block* are defined as assembly of *block* and *thread*, respectively. A *grid* corresponds to a GPU, a *block* corresponds to an MP, and a thread corresponds to an SP, respectively. As GPU instructions are issued in a unit of 32 threads called warp and executed simultaneously, computing efficiency is maximized when the number of thread is a multiple of 32. On the other hand, all *block* and thread cannot be allocated at a time when declared numbers of *block* and thread are greater than the number of MP and SP. In that case, the next idling *block* computation is allocated to the vacant MP after completion of the *block* computation. Similarly, idling thread is allocated to vacant SP which completed the computation.

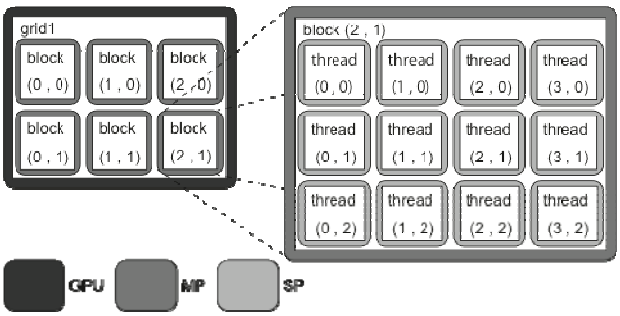

Fig. 2 The way to manage the calculation resources.

3. Related works

A lot of studies report that GPU accelerates problems with highly parallelism such as N-body and molecular dynamics. However, these results are derived from much tuning, investigation and adjustment by specialist of the target application. As technical knowledge is also required to tune CUDA code, developing cost of GPGPU should be high. Several mechanisms and frameworks are proposed to make utilizing GPGPU by CUDA easier. CuPP is a framework to easily integrate CUDA into the application written in C++. CuPP also provides two features; (1) easy access to memory management and kernel function call and (2) several libraries for C++ which can be imported to CUDA.

Frameworks are released not only for GPGPU, but also for parallel computing environment by PC-clusters. For example, our research group has been developing a study of framework in which an evaluation function of Genetic Algorithm (which is abbreviated as GA) is offloaded to remote nodes in the PC-cluster. As a large part of its computation can be exploited various parallelism, GA is adopted for case study. A function call specified by the framework enables off-loading operations to nodes connected to the network. As the result of the report, the framework is confirmed both easy implementation of the parallel program and acceleration of the computing speed. As the framework focuses only on GA, operations in each remote node cannot be modified by the programmer. The framework proposed in this paper focuses on the flexible implementation maintaining advantages mentioned above.

Besides some studies of accelerating existing algorithm, the study to realize easy parallel operation is being excepted.

4. Framework

4.1 Overview of the framework

The one of the most important features for GPGPU is that GPU accelerates a part of the operation by parallel computing. On the other hand, as users are required to learn technical knowledge of parallel computing and GPU architecture, software development cost frequently becomes high. This section explains the framework which enables implementation of GPGPU without technical knowledge. Popularization of GPGPU by reducing the difficulty of implementation is also the objective of the framework.

4.2 Functions and constraints of the framework

4.2.1 Functions of the framework

The framework has functions which are allocating and deallocating memories, transferring data with the device and executing kernel function, instead of implementation by users. Operations in GPU can be completely free to shift by implementing the kernel function by the user.

4.2.2 Constraints of the framework

As input and output is limited to only a pointer referred to the data, the region stored in the intermediate data is common to the region of the result of the kernel function. The constraint requires transferring data in every call of the framework. For example, even executing iteration on the data, it is required to allocate, transfer, and deallocate data. In addition, the number of types of kernel function to offload GPU is limited to only one.

4.2.3 The framework in multiple GPUs

The user can use multiple GPUs by obtaining the number of GPU in the framework. A thread of the host program manages a status of the GPU in CUDA 3.2, which is the development environment of the framework. Therefore, it needs to generate the same number of thread as the number of GPU. The framework use OpenMP to the multithreading in the host program.

An actual implementation to use multiple GPUs is as follows. At first, by supporting OpenMP, the same number of threads as the number of GPUs specified by the user is generated. Each ID of threads is associated to the ID of GPU to allocate GPU. Each thread performs operations such as allocating memories and executing kernel function mentioned above for its managing GPU.

However, the current implementation of the framework splits the data equally the number of GPUs to divide the operation. Therefore, it cannot be changed the load of computation or allocated variant kernel functions according to the potential of GPUs. As data communication among GPUs is also not provided, exchanging data among threads by multiple GPUs cannot also be implemented.

4.3 Structure of the framework

Fig. 3. shows the structure of the framework. The framework consists of three functions; throw_to_gpu, get_from_gpu, and kernel function. throw_to_gpu function submits a job to the GPU device.

4.3.1 throw_to_gpu function

List 1. Shows the codes of functions provided by the framework. The tasks throw_to_gpu function progresses following steps;

(1) allocating memory to store data for input and output,
(2) transferring data to the device memory, and
(3) executing kernel function.

At the MallocDeviceMemory function, Device_Data, which is memory in the GPU device, is allocated by cudaMalloc function provided by CUDA library. Secondly, Host_Data which is memory in the host is also allocated by malloc function in MallocHostMemory function. Data in the argument indicates the address of input data to transfer to the device. Data is copied to Host_Data and Device_Data by LoadData function and MemCopyH2D function, respectively. cudaMemcpy function is called to transfer Data from the host to the device inside MemCopyH2D function. At the last of the function, KernelLaunch function calls the predefined kernel function. As the kernel function is called as a non-blocking function, the host can progress its own calculation in parallel with the computation of the device.

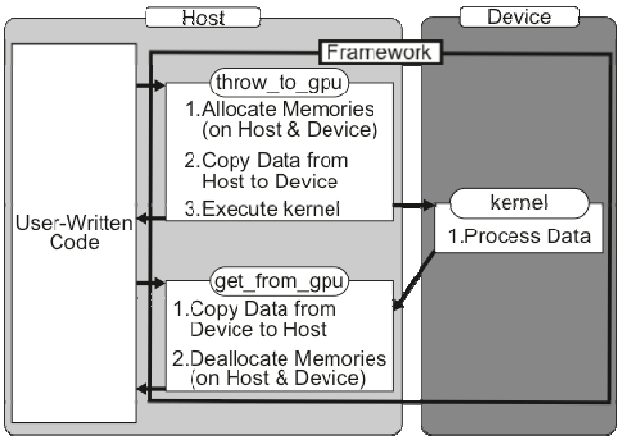

Fig. 3 Structure of the framework.

List1 Code of the framework.

```
 1 void** Host_Data;
 2 void*** Device_Data;
 3
 4 ...
 5
 6 void throw_to_gpu(void* Data, int Data_Size, int Number_of_Threads,
                     int Number_of_Blocks, int Number_of_Gpus){
 7    int Size = _msize(void* Data);
 8    int Size_per_Gpu = Size / Number_of_Gpus;
 9    Device_Data = MallocDeviceMemory(Number_of_Gpus, Data_Size);
10    Host_Data = MallocHostMemory(Number_of_Gpus, Size_per_Gpu);
11    LoadData(Host_Data, Data, Number_of_Gpus, Size_per_Gpu, Data_Size);
12    MemCopyH2D(Host_Data, Device_Data, Number_of_Gpus, Size_per_Gpu);
13    KernelLaunch(Device_Data, Number_of_Gpus, Size_per_Gpu, Number_of_Threads, Number_of_Blocks);
14 }
15
16 void get_from_gpu(void* Processed_Data, int Data_Size, int Number_of_Gpus){
17    int i;
18    int Size = _msize(void* Processed_Data);
19    int Size_per_Gpu = Size / Number_of_Gpus;
20    MemCopyD2H(Host_Data, Device_Data, Number_of_Gpus, Size_per_Gpu);
21    StoreData(Processed_Data, Host_Data, Number_of_Gpus, Size_per_Gpu, Data_Size);
22    for(i = 0; i < Number_of_Gpus; i ++){
23        free(Host_Data[i]);
24    }
25    free(Host_Data);
26    free(*Device_Data);
27    free(Device_Data);
28 }
```

As operations mentioned above are executed, data is always transferred when the function is called, even the data in the device can be reusable.

4.3.2 get_from_gpu function

The tasks get_from_gpu function progresses following two steps;

(1) transferring result of kernel function to the memory in the host, and
(2) deallocating memory associated with kernel function.

MemCopyD2H function transfers the result of kernel function from Device_Data to Host_Data by cudaMemcpy function. The function waits until the completion of the device when the kernel function is executing.
StoreData function transfers the data from Host_Data to Processed_Data, which is specified as argument by user.

List2 Example of a code which uses the framework (kernel function).

```
1 __global__ void KernelFunction(void* Data){
2     int tid = threadIdx.x;
3     int bid = blockIdx.x;
4     int bdm = blockDim.x;
5     double * CastData;
6     CastData = (double *)Data;
7     CastData[bdm * bid + tid] = tid * bdm;
8 }
```

At this time, since Host_Data is managed by double pointer, it is transformed to single pointer. In the end, Device_Data and Host_Data are deallocated.

4.3.3 kernel function

List 2. Shows an example of kernel function. Note that the kernel function is defined as the name of function and the file name of the code. The framework includes the file implements the kernel function. By constraints of the framework, the argument of the kernel function is designated as a pointer to data, which is corresponding to Data in the List 2. As the pointer is void, it is required to cast any type prior to use. In List 2., the pointer referenced to Data is casted to pointer of double type and stored CastData.

4.4 Method for utilization of the framework

List 3. shows an example of the code when using the framework. User can accomplish operations on GPU through calling a couple of throw_to_gpu function and get_from_gpu function. The following parameters used in each function.

- Data: the pointer associated with the device memory which is stored data used in computation
- Number_of_Threads: the number of thread per a *block*

List.3 Example of a code which uses the framework (host).

```
1 double Evaluation(double* Data, int Data_Size, int Number_of_Datas, double* Processed_Data) {
2     throw_to_gpu(Data, Data_Size, Number_of_Threads, Number_of_blocks, Number_of_Gpus);
3     get_from_gpu(Processed_Data, Data_Size, Number_of_Gpus);
4 }
```

- Nuber_of_Blocks: the number of *block* per a *grid*
- Processed_Data: the pointer associated with the host memory which is stored the result of kernel function
- Number_of_Gpus: the number of GPU used for the computation

Data is treated as the pointer of type void in throw_to_gpu function and get_from_gpu function. Therefore, the size of Data is specified by Data_Size to allocate memories and assign operations. A unit of Data_size is Byte. Processed_Data, which is a pointer of type void, is an argument of get_from_gpu function to assign the region stored the result of kernel function. Other arguments used in get_from_gpu function is specified as same as throw_to_gpu function.

5. Implementation of GA

5.1 Implementation of GA Using the Framework

We evaluated the framework with a program that executes part of its GA computation in parallel on the GPU. GA is a heuristic algorithm for finding optimum or approximate solutions in optimization or search problems. As mentioned in the section on related studies, we chose this particular algorithm because it has a high parallelism. Fig.4 is the flowchart of the algorithm.

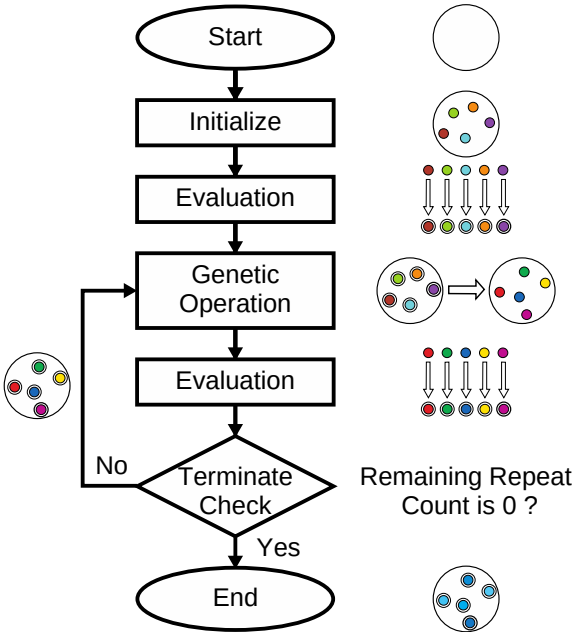

Fig. 4 Flowchart of GA.

The Initialize step in Fig.4 provides a sufficiently large number of random individual solutions. The Genetic Operation, which is repeated until the termination conditions are met, modifies the solutions in certain ways. The termination conditions determine whether the optimum or approximate solution is reached, or the process has been repeated a certain number of times. In the Evaluation step, each of the individual solutions is evaluated on its fitness, or how close it is to the optimum solution. More specifically, the individual solutions are applied to the problem formula and the results are used for fitness values. The more complex a problem the longer it will take to evaluate the individuals, and this could take up the majority of execution time in GA. The evaluation process also has thread level parallelism because it is done for each of the distinct individuals; thus we speed up this process by offloading it to the GPU and parallelizing.

We used the Rastrigin and Rosenbrock functions for evaluation, which are commonly used in GA testing. Their expressions are as follows:

$$F_{Rastrigin}(x) = 10n + \sum_{i=1}^{n}\left(x_i^2 - 10\cos(2\pi x_i)\right) \quad (-5.12 \le x_i < 5.12)$$
$$min(F_{Rastrigin}(x)) = F(0,0,\ldots,0) = 0$$
$$F_{Rosenbrock}(x) = \sum_{i=1}^{n-1}\left(100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right) \quad (-2.048 \le x_i < 2.048)$$
$$min(F_{Rosenbrock}(x)) = F(1,1,\ldots,1) = 0$$

x is an individual solution and consists of multiple design variables (x_i). The dimension number n represents the number of design variables. Individuals (x) in SGA are binary, and we refer to the bit length as the individual length. For example, if n = 10, then x consists of x_0 through x_9, and if each of x_i is 10 bits in length, the individual length is 100 bits. A large number of individuals are provided in this form and each of them is to be evaluated.

Each computation in the Rastrigin and Rosenbrock functions while computing the summation is self-contained, so it has instruction-level parallelism in addition to the thread-level parallelism among individuals. We implemented the system so that each of the parallelisms corresponds to the two levels of the hardware parallelism. First, a single *thread* is responsible for one dimension of summation computation. Then a single *block* is responsible for one individual solution. We refer to allotting one individual to one *block* as the "basic implementation". In basic implementation, the number of individuals equals the number of *blocks*, and the number of dimensions equals the number of *threads* per *block*. Fig.5 shows how

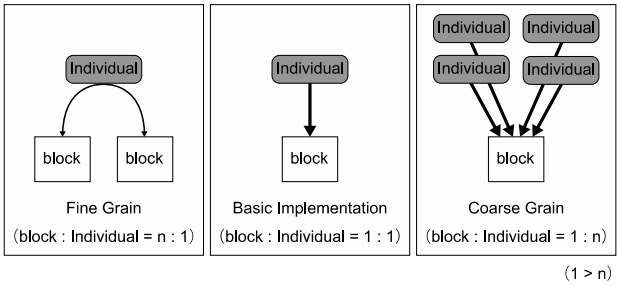

Fig. 5 The Way to Allot Individuals to a Block

individuals are allotted to *blocks*: the center box shows the basic implementation.

Aside from the basic implementation, we can also allot multiple individuals to a single *block*, as shown in Fig.5 (right), or allot one individual to multiple *blocks*, as shown in Fig.5 (left). We shall refer to the former as being coarse in parallel granularity. Generally, parallel granularity represents the size of a single process when breaking up a larger process into parallel pieces. Allotting one individual to multiple *blocks* can be seen as fine in granularity. In other words, the coarseness of granularity is proportional to the number of individuals allotted to a single *block*. The number of individuals per *block* is determined by *number of individual/number of blocks*, and is inversely proportional to the number of *blocks*. Specifying the number of *blocks* using the Number_of_Blocks argument allows us to adjust the parallel granularity.

5.2. Comparison of Execution Time with CPU

We compared the execution time between two settings: one done by the GPU with basic implementation, and one done by the CPU. The tests were done with a dimension number of 10 and a individual length of 100 bits. With 400, 2000, and 4000 individuals, we measured the time taken to evaluate all individuals with both the Rastrigin and Rosenbrock functions 100 times and took the average. Note that because these evaluation functions are light in workload, we made the computations run 1000 times per kernel function call to boost the workload.

Tables 1 and 2 show the hardware specs of the machines where the test was run. Machine 2 has two GTX460 GPUs installed, and we ran tests using one GPU and two GPUs respectively. Both of the two programs, one for the CPU and one for the GPU, have identical code except for evaluation. The program for CPU was run on Machine 2. Figures 6 and 7 show the results. The bar graph shows the execution time, and the line graph shows the throughput, or the number of evaluations done per second.

As a result, both functions showed higher performance with the GPU. Also, aside of the results for dual GTX460 shown in Fig.7, when the number of individuals (or the number of *blocks*) increased from 400 to 2000, evaluation performance with the GPU rose proportionally.

Table1 Systems used in experimentation.

	Machine 1	Machine 2
GPU	Tesla C2050	GeForce GTX 460
CPU	Xeon W3530 2.8GHz	Core i5 2400 3.1GHz
Host Memory	6GB	8GB
GPU code Compiler	CUDA toolkit 3.2	
CPU code Compiler	gcc 4.4.5-15ubuntu1	

Table2 Spec of the GPUs.

	Tesla C2050	GeForce GTX 460
Memory	3GB	1GB
Memory Band Width	144 GBs	115.2GBs
The number of SPs	448	336
The bunber of MPs	14	7

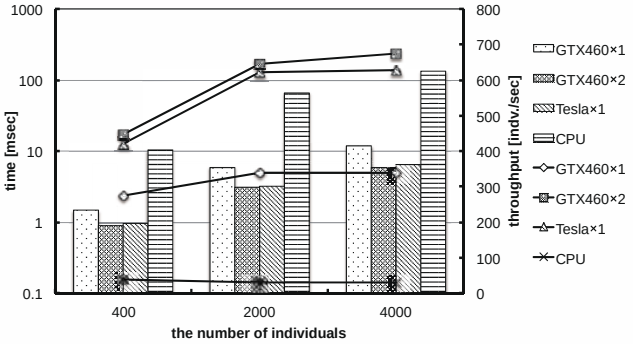

Fig. 6. Execution speed comparison between CPU and GPU.(Rastrigin)

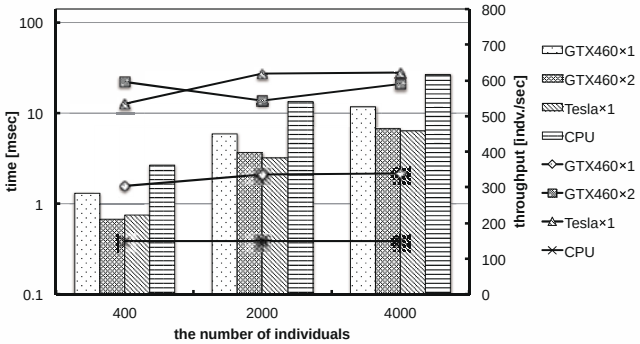

Fig. 7. Execution speed comparison between CPU and GPU. (Rosenbrock)

However, when the number increased from 2000 to 4000, there was neither a large gain nor degradation in performance.

This characteristic is conceivably caused by the overhead on the kernel function. The number of individuals does not affect this overhead, whereas the execution time of the kernel function rises linearly, in proportion to the number of individuals. When the throughput is measured as *number of individuals/evaluation time*, it appears

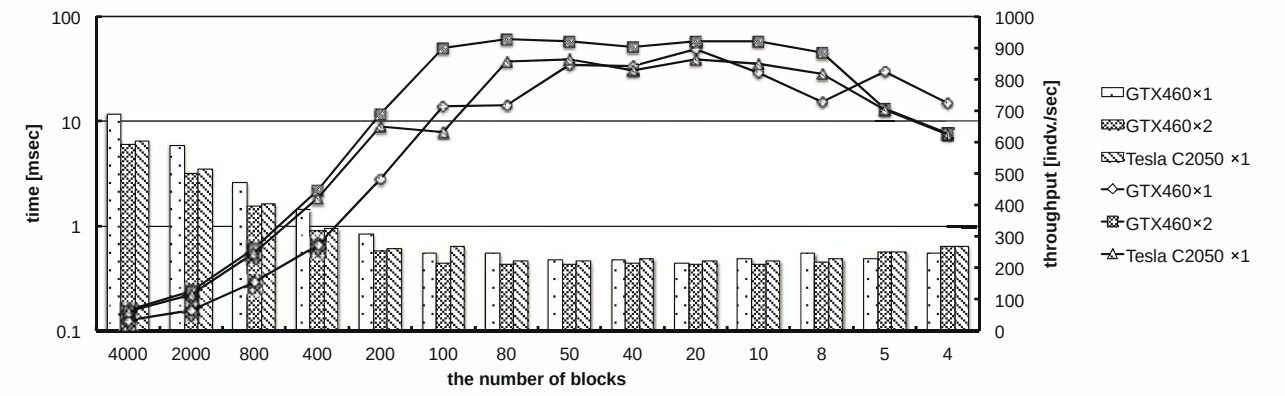

Fig. 8. Relationship between execution speed and changing parallel particle size.(400individuals)

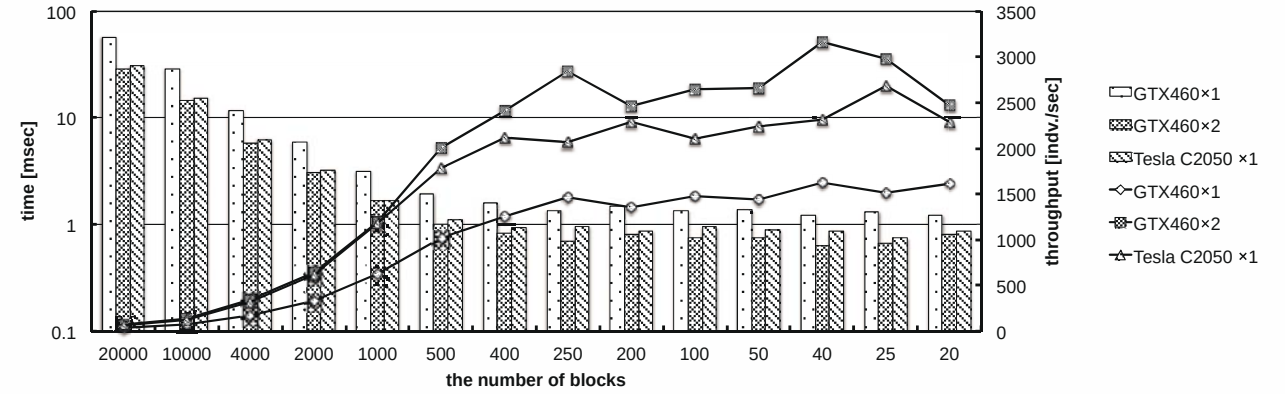

Fig. 9. Relationship between execution speed and changing parallel particle size.(2000individuals)

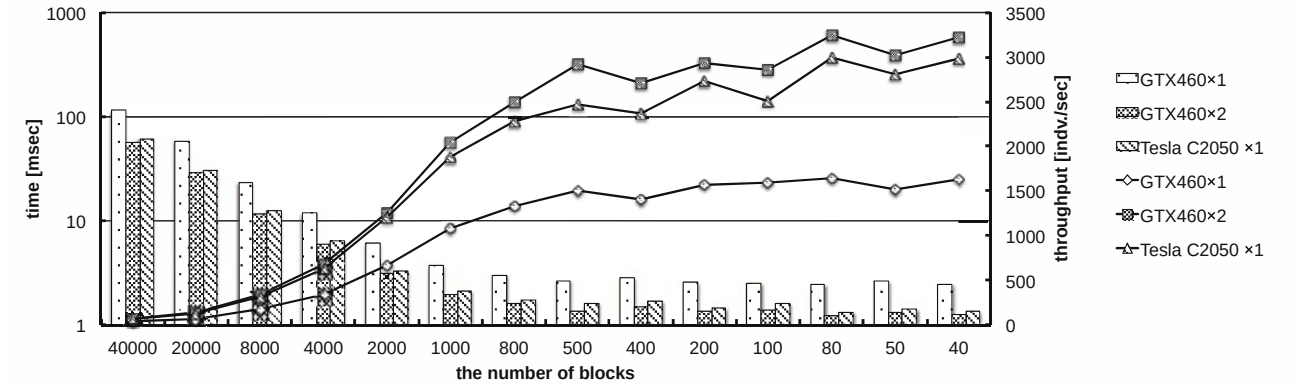

Fig. 10. Relationship between execution speed and changing parallel particle size.(4000individuals)

to become higher when there are more individuals because the kernel function overhead has a relatively smaller impact. With 2000 and 4000 individuals, apparently the effect of the overhead was trivial enough to keep the performance gain smaller.

5.3. Correlation between Parallel Granularity and Execution Time

We studied the correlation between parallel granularity and execution time. These tests were run with a dimension number of 10, an individual length of 100 bits, and individual numbers of 400, 2000, and 4000. We measured the time taken to evaluate each of the individuals 1000 times using the Rastrigin function. We ran the test 100 times and took the average. Figures 8, 9, and 10 show the results. In

each of the cases, we saw better performance when granularity was more coarse and worse performance when granularity was finer. However, this is not always true: there are drops in performance even with high granularity. The optimum number of *blocks* depends on the type of GPU and the number of GPUs installed.

5.3.1. Correlation Between Execution Speed and Number of Threads

One reason the execution time was better with more coarse granularity is that there were more *threads* per *block*. In basic implementation, the number of *threads* per *block* is 10, which is fewer than 32, the number of *threads* per *warp*. This caused computations worth of 32 *threads* even when computing on 10 *threads*, which was ineffective and resulted in lower execution speeds.

5.3.2. Correlation Between Execution Speed and Number of Blocks

The number of *blocks* also affects the execution speed. When there are too many *blocks*, there is a bigger effect of context switch latencies. More *blocks* means more memory access, but when there is a sufficient number of *blocks*, memory access latencies can be hidden by executions of other blocks. That is to say, when there are too few *blocks*, there is a bigger effect of memory access latencies. Thus we need to strike a balance between context switch and memory access latencies. Because *blocks* are allotted to MPs, we can assume higher performance when the number of *blocks* is several times or several ten times larger than the number of MPs. For example, Fig.8 and Fig.9 show that performance degrades when the *block* count is 20 or less. However, as shown in Fig.9, when the block count is 20, performance degrades with dual GTX460 and TeslaC2050 but not with single GTX460. The reason for this is that for the former, the MP count is 14 (making the *block* count 1.4 times the MP count), whereas for the latter, the MP count is 7 (making the *block* count 3 times the MP count). The latter was successful in hiding the memory access latencies.

The best performance can be seen in Fig.10 where the *block* count is 80. When the number of individuals is 4000, there are 40000 *threads* with 500 *threads* per *block*. The other case with 500 *threads* per *block* is when the *block* count is 8, as shown in Fig.8, but because 8 *blocks* were not sufficient, it did not yield high performance. Another case with 500 *threads* per *block* is in Fig.9 where the *block* count is 40. The throughput is high due to the large number of *blocks*. However, as seen in the comparison with CPU, more *blocks* should yield higher performance when the number of *threads* per *block* is identical. This is why 4000 individuals for 80 *blocks* resulted in better performance than 2000 individuals for 40 *blocks*.

The effect of multiple GPUs became apparent when workload increased, rather than when there were more individuals. Note, however, that this resulted from using two of the same type of GPU. Using different GPUs with different capabilities may not give the same results; namely the one with lower performance may become a bottleneck for the whole process.

6. Discussions

We confirmed that GPU computation with CUDA is possible by invoking just two functions implemented in the framework. The kernel function is flexible as it can be written freely and finely tuned. However, there are many limitations as well. To avert the limitations, the kernel function needs more arguments. Also, by preparing a dedicated storage for computation results, data transferred to the device can be stored for later use. This shall cut down on data transfer load when different computations must be done on the same set of data. More complex computations may be implemented once the framework is extended to use multiple types of kernel functions. Another enhancement would be to allow multiple jobs to be offloaded at once and the results retrieved at given moments, which would allow for more flexible and efficient computations.

There are major limitations when computing with multiple GPUs as well. First, subject data is divided equally among GPUs, so workload balance is not optimal when the GPUs differ in performance. This calls for a feature where the user may specify workload balance, or otherwise a feature where GPU info is collected and workload is balanced automatically. Also, GPUs should be able to communicate with each other. However, because inter-GPU communication is more costly than intra-GPU communication, the framework should allow different kernel functions to be executed for different GPUs so that there is a smaller need (or no need) for inter-GPU communication.
Performance testing showed that by using the framework, better performance could be achieved compared to CPU-only execution. By using a higher parallel granularity than the basic implementation, we saw even higher performance. We shall see more improvements by tuning the memory access and reducing conditional branching.
CUDA supports many features such as asynchronous communication between the device and host, and a feature that allows users to choose the type of host memory. By supporting these various CUDA features, the framework could be even more valuable.

7. Conclusion

This paper proposes a framework of parallel computing by CUDA, which handles memory allocation and data-transfer. Developers can utilize GPU through only two functions. SGA is implemented to validate the framework

and discussed about potential of acceleration based on several parallel and application parameters. The evaluation of the relationship between parallel granularity and computing time obtained that the number of thread and block in the kernel function influences the performance of GPU. In the future work, several features to introduce more flexibility of tuning performance may be implemented and discussed.

References

[1] Lo, Y.-T., Tsai, Y.-L., Wang, H.-W., Hsu, Y.-P. and Pai, T.-W.: Using Solid Angles to Detect Protein Docking Regions by CUDA Parallel Algorithms, 2010 International Symposium on Parallel and Distributed Processing with Applications (ISPA), pp.536–541 (2010).

[2] Oiso, M., Yasuda, T., Ohkura, K. and Matumura, Y.: Accelerating steady-state genetic algorithms based on CUDA architecture, 2011 IEEE Congress on Evolutionary Computation (CEC), pp.687–692 (2011).

[3] Phillips, E. and Fatica, M.: Implementing the Himeno benchmark with CUDA on GPU clusters, 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pp.1–10 (2010).

[4] Top500 Supercomputing Sites, http://www.top500.org/.

[5] Li, X. and Xu, M.: Water simulation based on HLSL, 2009 IEEE International Conference on Network Infrastructure and Digital Content (IC-NIDC), pp.1066–1069 (2009).

[6] Belleman, R.G., Geldof, P.M. and Zwart, S. F.P.: High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA, New Astronomy, Vol.13, pp.103–112 (2008).

[7] Meel, J.A., Arnold, A., Frenkel, D., Zwart, S.F.P, and Belleman, R.G.: Harvesting graphics power for MD simulations, Molecular Simulation, Vol.34, No.3, pp.259-266 (2008).

[8] Breitbart,J.: CuPP - A framework for easy CUDA integration, IEEE International Symposium on Parallel Distributed Processing(IPDPS),pp.1–8 (2009).

[9] T.Hiroyasu, R.Yamanaka, M.Yoshimi and M.Miki: A Framework for Genetic Algorithms in Parallel Environments, The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), pp.751–756 (2011).

Masato Yoshimi recieved the B.E. and M.E., and Ph.D. degrees from Keio University, Japan, in 2004, 2006, and 2009. He is currently Assistant Professor in Doshisha University. His research interests include the areas of intelligent systems, reconfigurable computing and parallel processing.

Yuki Kurano is currently studying in Faculty of Engineering and Science in Doshisha University. His research interests in-clude the high performance computing using graphic processor and productive computing systems.

Mitsunori Miki received Ph.D. degree from Osaka City University, Japan in 1978. He was Research Fellow in Osaka Municipal Technical Research Institute, Associate Professor in Kanazawa Institute of Technology, and Associate Professor in Osaka Prefecture University. He is currently Professor in graduate school of Doshisha University. His current research topics are creative office environment and intelligent lighting systems based on intelligent parallel and distributed optimization.

Tomoyuki Hiroyasu received Ph.D. degree from Waseda University, Japan in 1997. He was Research Associate in Waseda University and Assistant Professor in Doshisha University. He is currently Professor in Doshisha University. His research interests include the areas of evolvable computing, optimum design, and medical image engineering.