Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment

Satoshi Ohshima[1], Kenji Kise[1], Takahiro Katagiri[1], and Toshitsugu Yuba[1]

Graduate School of Information Systems
The University of Electro-Communications
1-5-1, Chofugaoka, Chofu-shi, Tokyo, Japan
Tel: +81-42-443-5644 / Fax: +81-42-443-5644
ohshima@yuba.is.uec.ac.jp, {kis, katagiri, yuba}@is.uec.ac.jp

Abstract. GPUs for numerical computations are becoming an attractive alternative in research. In this paper, we propose a new parallel processing environment for matrix multiplications by using both CPUs and GPUs. The execution time of matrix multiplications can be decreased to 40.1% by our method, compared with using the fastest of either CPU only case or GPU only case. Our method performs well when matrix sizes are large.

1 Introduction

The performance of Graphics Processing Units (GPU) has been significantly improved in recent years. Compared with the CPU, the GPU is better suited for parallel processing and vector processing and has evolved to perform various types of computation, in addition to graphics processing, including numerical computations. General-purpose computations on GPUs (GPGPU) have been examined for various applications[1–3].

A high-performance computing environment is necessary for numerical computations like physics and earth environment simulations which require enormous computational power. Matrix multiplication is an important operation in numerical computation. Speeding up matrix multiplication results in a corresponding speed up increase in various numerical computations.

Basic Linear Algebra Subprograms (BLAS)[4] is frequently used as a basic numerical calculation library. Automatically Tuned Linear Algebra Software (ATLAS)[5], is a fast implementation of BLAS in CPUs. These libraries have succeeded in exploiting performance enhancing features of a CPU.

In BLAS, matrix multiplication is treated as a computation of $C = \alpha \times A \times B + \beta \times C$ where A, B, and C are matrices, and α and β are scalars. Improving performance of such computations will speedup of various numerical calculations.

We propose a heterogeneous computing environment for parallel processing using both CPUs and GPUs for numerical computations. First we divide the larger problem into two partial problems and assign one to the CPU and the

⋆ Candidate to the best student paper award

other to the GPU. Ideally, this results in achieving high performance of both the CPU and the GPU. We evaluate this method of parallel processing using the NVIDIA GeForce7800GTX and the 6600GT as our GPUs.

Section 2, discusses the background and related work. Section 3, proposes a parallel processing method using 1-CPU and 1-GPU. Implementation and analysis of our method for matrix multiplication are described in Section 4. Section 5 describes the experimental results measured on a real heterogeneous environment and section 6 discusses about future research issues.

2 Background and Related Work

Graphics processors generate large number of polygons at a very high speed. In generating polygons, vector and matrix computations are frequently used. Many computations can be executed in parallel on a GPU. GPUs have evolved rapidly with hardware suited for both vector and highly parallel computations compared with a CPU. In addition, the programmable shader, controls processor's behavior in software level, has become popular in newer GPUs. Since floating point arithmetic of a GPU is advanced these GPUs can efficiently execute various computations rather than generating polygons[2, 3, 6].

The GPGPU aims at resolving target calculations utilizing the computational power of a GPU. The main scope of GPGPU includes the computation of high-level shading and lighting in creating real images[7], various simulations and visualizations[8, 9]. These are examples related to graphics computations, the original use of GPUs. Besides these graphics computations, utilization for numerical computations is a new application domain of the GPU[10–12]. Floating-point computations of GPUs have a lower precision than CPUs[13]. Therefore, further evaluation and improvement of precision are necessary because of the very high arithmetic precision required in numerical calculations.

Matrix multiplication is a popular GPGPU application. Current research includes: efficient utilization of GPU for matrix multiplication, decreasing execution time by using vector computation and programmable shaders and an effective utilization of GPU inner cache[2, 14–17]. However, effective performance evaluation results have not yet been obtained, because of the issues related to memory and bandwidth in inner GPU.

Task parallelization has been used to increase performance in systems having both CPUs and GPUs. For example, in a real-time movie, the CPUs calculate the position of numerous objects and the GPUs calculate the shades and high lights of these objects. However, data parallelization in both the CPUs and GPUs is rare. We propose data parallelization with the CPUs and GPUs for numerical calculation.

The research on parallel processing in heterogeneous environments includes multiple CPUs with different performance. The problems addressed are: scheduling for effective utilization of all processors, load balancing in a dynamically changing environment, and resolving differences in arithmetic precision[18]. We

try to overcome such issues using a new domain as CPU and GPU complex heterogeneous system.

3 Parallel Processing in a CPU and GPU Heterogeneous Environment

3.1 Execution Time Analysis of Parallel Processing

Conventional approaches for execution time analysis for both CPU and GPU include processor speedup, increased processor utilization (various proposals and implementations have been investigated for approaching theoretical performance), and parallel processing with multiple processors.

First, we formulate the execution time for CPUs. The execution time T_{CPU_ALL} is defined in equation (1). We denote the number of operations in the target computation as R. The execution time required for solution using peak CPU performance is denoted as a function of R, or $f_{CPU}(R)$. The effective execution time increases because the CPU cannot always attain peak performance. We denote the increase of execution time relative to the ideal execution time (execution time at effective performance / execution time at peak performance) as a ($a \geq 1$). Ideally, the execution time is divided by n ($n \geq 1$), the number of CPUs used in parallel processing. The execution time is increased by the parallelization overhead when more than two CPUs are used. We neglect this time for simplification.

$$T_{CPU\_ALL} = \frac{f_{CPU}(R) \times a}{n} \tag{1}$$

Similarly, we formulate the execution time for GPUs. The execution time T_{GPU_ALL} is defined in equation (2). In this equation, the execution time required for a solution using the peak GPU performance is $f_{GPU}(R)$, the increase of execution time relative to the ideal execution time is b ($b \geq 1$), and the number of GPUs is m ($m \geq 1$).

$$T_{GPU\_ALL} = \frac{f_{GPU}(R) \times b}{m} \tag{2}$$

In previous research on numerical computations using GPGPUs, the execution time of a GPU system was compared to that of a CPU system, as shown by (1) and (2). However, GPGPU systems often have both CPUs and GPUs. Therefore, we propose a parallel processing method to obtain the overall CPUs and GPUs performance. We divide a target computation into a parts, and assign them to CPUs and GPUs to perform.

Assume that the target computation is divided into two partial computations. One partial computation with the assignment ratio r ($0 \leq r \leq 1$) of the computation is assigned to CPUs. The other, with assignment ratio $1 - r$ of the computation, is assigned to the GPUs. Then, the CPUs' execution time T_{CPU} defined in equation (1) becomes equation (3). Similarly, the GPUs' execution time T_{GPU} defined in equation (2) becomes equation (4).

$$T_{CPU} = \frac{f_{CPU}(R \times r) \times a}{n} \tag{3}$$

$$T_{GPU} = \frac{f_{GPU}(R \times (1-r)) \times b}{m} \tag{4}$$

The execution time for a parallel system is defined as $T_{Parallel}$ in (5). Because the target computation ends when both the CPUs and the GPUs finish computations, the execution time is obtained as the maximum of either T_{CPU} or T_{GPU}. The parallelization overhead. is omitted for simplification.

$$T_{Parallel} = max(T_{CPU},\ T_{GPU}) \tag{5}$$

To attain optimal performance, the execution time $T_{Parallel}$, defined by equation (5), must be minimized.

3.2 Case of One CPU and One GPU

For simplicity, we evaluate the proposed method for the case of one CPU and one GPU. If the parameters $a, n, b, and\ m$ in equations (3) and (4) are constants, then $T_{Parallel}$ is a function of only r as an input parameter. We propose a method by which to achieve high performance by properly estimating the parameter r. For one CPU and one GPU, we have $n = 1 and\ m = 1$ in equation (3) and (4). Therefore, equation (3) can be simplified to (6), and equation (4) can be simplified to (7).

$$T_{CPU} = f_{CPU}(R \times r) \times a \tag{6}$$

$$T_{GPU} = f_{GPU}(R \times (1-r)) \times b \tag{7}$$

3.3 Parallelization of Matrix Multiplication

The interface of matrix multiplication in BLAS is denoted in (8). In this equation, $A, B, and\ C$ are matrices, and $\alpha\ and\ \beta$ are scalars. This function updates the matrix C.

$$C = \alpha \times A \times B + \beta \times C \tag{8}$$

In this equation, matrix size is assigned the three values of $M, N, and\ K$, as shown in Fig.1(a). When a certain element of matrix C is updated, only the updated element of matrix C is referenced. We divide matrices A and C into $McandMg$, where Mc and Mg denote the sizes of matrices allocated to CPU and GPU respectively. Then we assign the partial matrices to the CPU and the GPU as shown in Fig.1(b). Thus, matrix multiplication can be executed in parallel without synchronization, and theassignment ratio $r = Mc/Mc + Mg$ is obtained. A matrix can be divided into any assignment ratio, and the value of r

can be changed freely. So, optimal division, i.e. static load balancing, is easy to achieve.

Parallel processing on 1-CPU and 1-GPU uses two threads. One is a thread handling the CPU, this thread performs the SGEMM function (single precision floating-point GEMM function) using ATLAS. The other is a thread handling the GPU, this thread performs data transfer between the CPU and the GPU, and issues instructions to the GPU. Matrix multiplication on the GPU is implemented as follows. We use DirectX as a graphics API and HLSL as a shading language for creating programs[19]. Vector calculations are used because the GPU can handle vector data and vector operations efficiently. Although a GPU has both vertex and pixel processing units (fragment processing units), the implementation herein uses only pixel processing units.

In the next section, we measure the performance of matrix multiplication using either only one CPU or only one GPU. Based on this measurement, we can predict the performance of parallel execution using both 1-CPU and a 1-GPU heterogeneous environment.

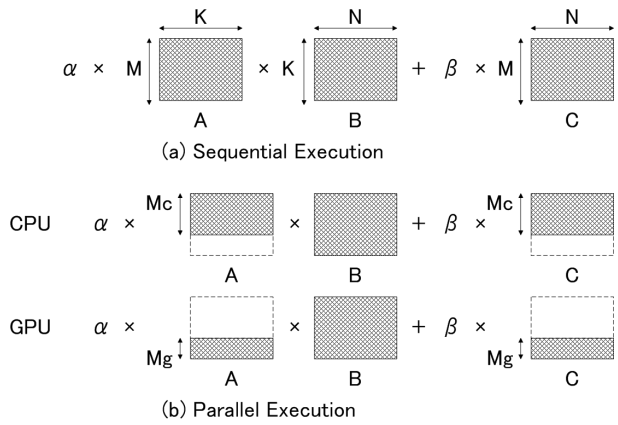


Fig. 1. Assignment of computation in Matrix Multiplication

4 Preliminary Performance Experiments

4.1 Performance of the 1-CPU System

The personal computer we used has a Pentium4 3.0GHz processor as the CPU and NVIDIA GeForce7800GTX as the GPU. Specifications of these processors are given by Table 1. First, we examine the performance of the 1-CPU system in the execution of matrix multiplication.

Table 1. Experimental Environment of Experiments

CPU	Pentium4 3.0GHz
Memory	1.00GB
OS	Windows XP
GPU	GeForce7800GTX
Graphics Bus	PCI-Express
VRAM	256MB
GPU's core clock	430MHz
GPU's memory clock	1.20GHz
amount of vertex shader unit	8
amount of pixel(fragment) shader unit	24

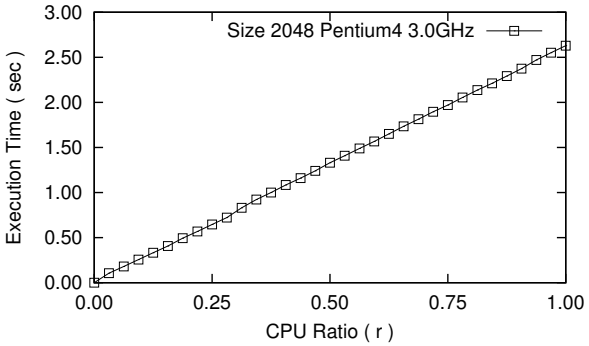


Fig. 2. CPU execution time for one Pentium4 3.0GHz

As described in Section 3.3, the SGEMM function of ATLAS is executed on the CPU in parallel processing in a 1-CPU and 1-GPU heterogeneous environment. We execute the SGEMM function and measure the execution time. The matrix size is 2,048, which means that $r = 1.0$ when $M = N = K = 2,048$. We examine the relationship between the value of r and the execution time by changing the vertical size Mc of matrices A and C.

The results obtained are shown in Fig.2. The horizontal axis denotes r, and the vertical axis denotes the execution time. This is a the graph of equation (6). We observe that the execution time of the SGEMM function is proportional to the computation size, and the amount of computation in matrix multiplication is proportional to r. Measurements are obtained by changing the assignment ratio of matrices of the size of multiples of 64.

4.2 Performance of the 1-GPU System

Next, we examine the performance of the 1-GPU system. The matrix size is defined as 2,048, and the relationship between the matrix size and the execution

time is examined in the same manner as the 1-CPU system. The execution time is measured from the beginning of data transfer from the CPU to the GPU to the end of data read back from the GPU to the CPU. We exclude the time required to initialize DirectX and load the HLSL program from the scope of measurement.

The results obtained are shown in Fig.3. The horizontal axis denotes the value of $1 - r$, and the vertical axis denotes the execution time. This is a graph of equation (7). As a result, the execution time of matrix multiplication using 1-GPU is also proportional to the matrix size. This is the result of changing the assignment ratio as we did with the 1-CPU.

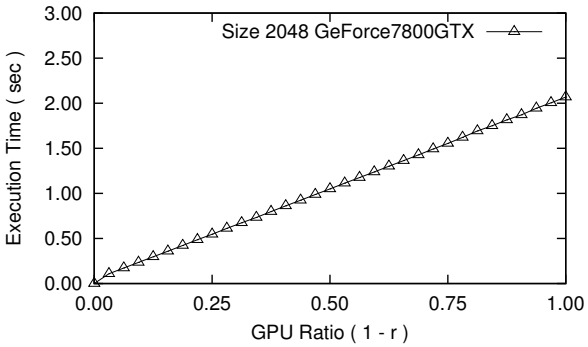


Fig. 3. GPU execution time for one NVIDIA GeForce7800GTX

4.3 Performance Prediction of the Heterogeneous Environment

We can predict the execution time on a parallel heterogeneous environment based on 1-CPU and 1-GPU execution times using the following process: we first put the 1-GPU graph (Fig.3) over the 1-CPU graph (Fig.2), while adjusting the horizontal edge. We obtain Fig.4, which depicts both T_{CPU} and T_{GPU} for the assignment ratio r. As mentioned above, the larger value of T_{CPU} and T_{GPU} is the predicted time for parallel execution of each r, because parallel execution finishes when both the CPU and the GPU complete the calculations. Figure 5 shows a graph of the prediction time obtained from Fig.4. This is a graph of equation (5). Matrix multiplication is executed at the fastest speed at the lowest point of the execution time on this graph, and its assignment ratio is optimal, that is, the value of r is minimized equation (5).

The result of the above prediction is that, in this environment, the execution time is expected to be the minimum when the CPU assignment is 43.8% of the computation. The execution time is expected to be reduced 44.1% compared with the 1-CPU only case and by 59.5% compared with the 1-GPU only case.

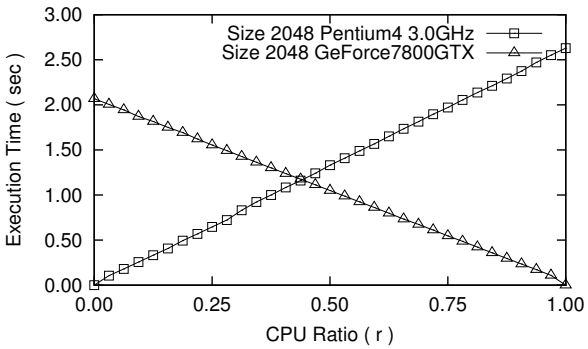


Fig. 4. CPU and GPU execution time

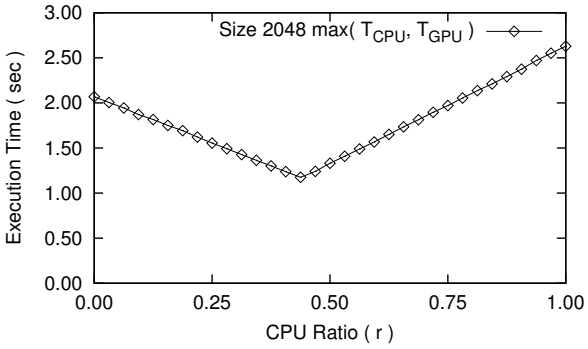


Fig. 5. Predicted execution time for parallel processing with 1-CPU and 1-GPU.

5 Performance Evaluation of the CPU and GPU Heterogeneous Environment

5.1 Performance Evaluation of the Heterogeneous System

In this section, we measure the execution time required in parallel execution on a heterogeneous environment. We implemented a parallel program using both the thread handling CPU and GPU, as described in Section 3.3. The SGEMM function of ATLAS is executed in the CPU thread, and data transfer between the CPU and the GPU and computations using the programmable shader are executed in the GPU thread. These experiments were carried out by changing the matrix size by 64 intervals.

The results obtained are shown in Fig.6. The center of the graph is lower compared with either side. Therefore, the execution time is decreased by parallel execution. The execution time is minimum when the CPU does 40.6% of the computation. The execution time is decreased by 45.1% compared with the CPU

only case, and by 60.8% compared with the GPU only case. Figure 7 shows the performance ratio when the higher performance of the 1-CPU only case and the 1-GPU only case is defined as 1.0. As a result, we got a performance improvement for the parallel case of 1.64 times.

We compare the experimental result with our prediction result. We first confirm that the assignment ratio minimizing the execution time. The execution time is predicted to be decreased the most when the CPU does 43.8% of the computation. Correspondingly, the experimental result also indicated that the execution time is minimum when the CPU does 43.8% of the computation. In addition, these values mean that execution time is minimum when the CPU does $1,152$ of the total $2,048$ size.

Next, we confirm the minimal execution time by using the optimal assignment ratio of the computation. In the prediction, the minimal execution time is 1.23 sec. Here, the ratio of the execution time to the CPU only case is 44.1%, and the ratio of the execution time to the GPU only case is 59.6%. In the experiment, the minimal experimental execution time is 1.26 sec. At this time, the ratio of the execution time to the 1-CPU only case is 45.1%, and the ratio of the execution time to the 1-GPU only case is 60.8%. The ratio of the minimal experimental execution time to the minimal prediction time is 102.4%.

We can conclude that we obtained high performance using parallel processing method. Moreover, we can predict with high precision both the execution time of matrix multiplication on a heterogeneous environment and the assignment ratio of computation for the minimal execution time.

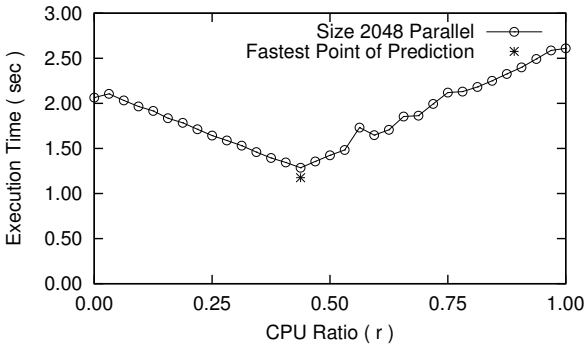


Fig. 6. Execution time measured on the CPU and GPU heterogeneous environment.

5.2 Performance Evaluation with Varying Matrix Sizes

We examine the performance in other matrix sizes of matrix multiplication to confirm whether our method is affected by the size of matrix in the computation. So, we evaluate performance by varying the matrix size of the matrix of

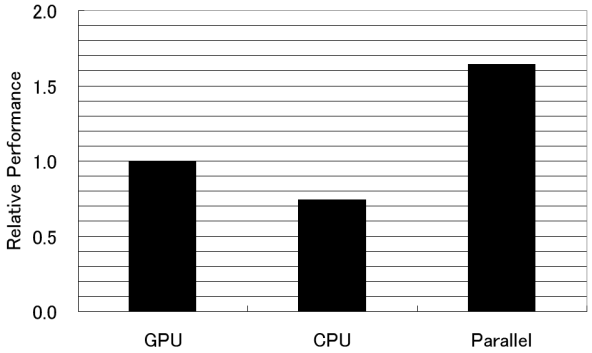


Fig. 7. Relative performance of parallel execution. (The higher performance of the CPU only case and the GPU only case is defined as 1.0.)

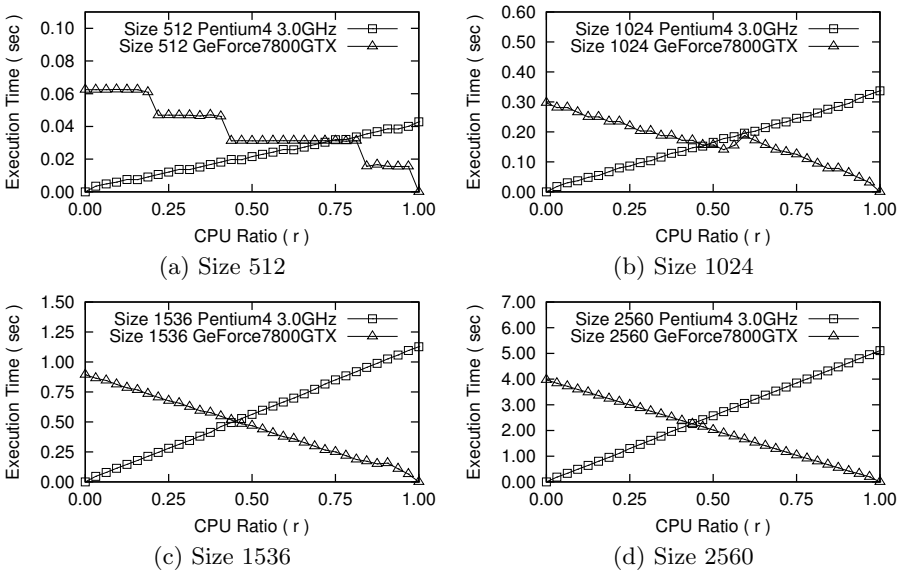
computation. The results obtained are shown in Figs.8 and 9. Figure 8 shows the result of the 1-CPU only case and the 1-GPU only case with varying the matrix sizes: 512, 1024, 1536 and 2560.

Graphs for the small computation sizes were unstable, but the tendency in the larger matrix size was the same as the figures we have already shown. Figure 9 shows the result in parallel execution. As a result, this method didn't work well when the computation size was small. However, when the computation size was large enough, the execution time was decreased by parallel processing. At this time, the assignment ratio for the minimal time in parallel execution was near the prediction point. In this heterogeneous environment, maximum speedup was obtained at the size of 2048. Further research is required to analyze the reason why the best performance was obtained in this size.

5.3 Performance Evaluation on Heterogeneous Environment with Different GPU

We tried to evaluate performance with another GPU of a different type to evaluate the effectiveness of the proposed method. We use the GeForce6600GT instead of the GeForce7800GTX. The differences for each GPU are shown in Table 2 1. The results obtained are shown in Figs.10 and 11. Figure 10 is a graph for the 1-CPU only case and the 1-GPU only case with changing matrix sizes of 1024 and 2048. Figure 11 is a graph in parallel execution. The environment in this experiment is the same as for the GeForce7800GTX.

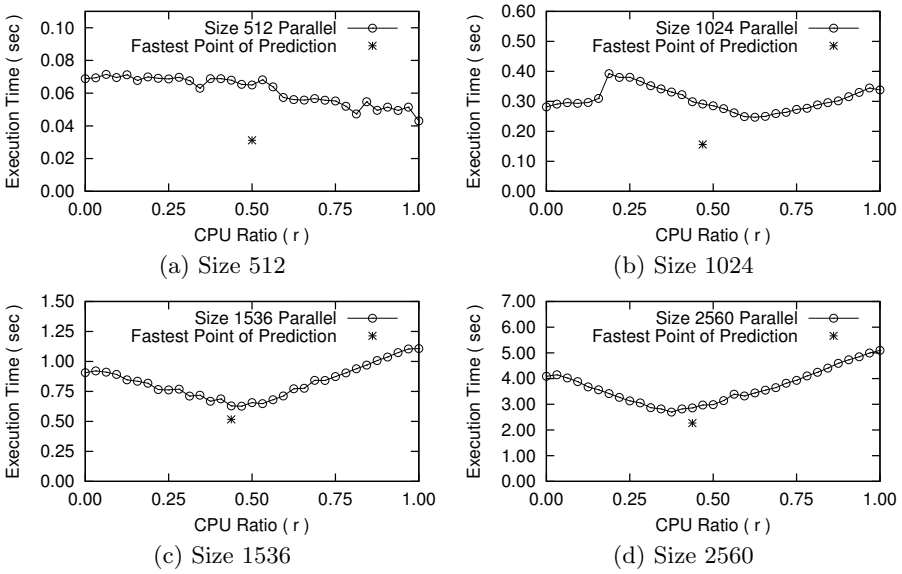
Our method didn't work well when the computation size was small, but the execution time was decreased by parallel processing when the computation size was large enough. The difference between the optimal assignment ratio of computation in CPU and GPU was small. The tendencies were almost same as the case of the GeForce7800GTX. The highest rate of speedup is shown for



(a) Size 512 (b) Size 1024

(c) Size 1536 (d) Size 2560

Fig. 8. 1-CPU only execution time and 1-GPU only execution time (GeForce7800GTX)



(a) Size 512 (b) Size 1024

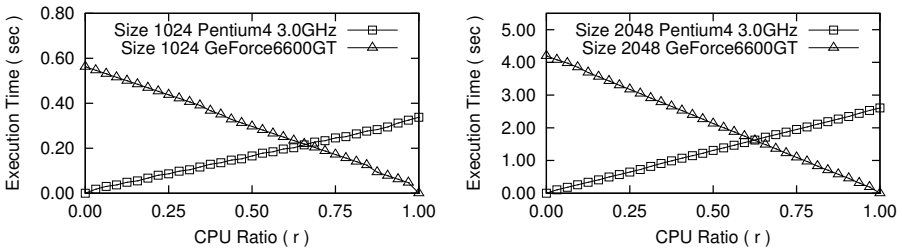
(c) Size 1536 (d) Size 2560

Fig. 9. Execution time measured on the implemented on parallel system with a CPU and a GPU, and fastest point of prediction (GeForce7800GTX)

the size 2048, and the ratio of the execution time for the 1-CPU only case was 70.5%. These results show that our method is useful when computation size is large. and doesn't work well when computation size is too small.

Table 2. Comparison of GPUs

GPU	GeForce7800GTX	GeForce6600GT
Graphics Bus	PCI-Express	PCI-Express
VRAM	256MB	128MB
GPU's core clock	430MHz	300MHz
GPU's memory clock	1.20GHz	1.00GHz
amount of vertex shader unit	8	3
amount of pixel(fragment) shader unit	24	8



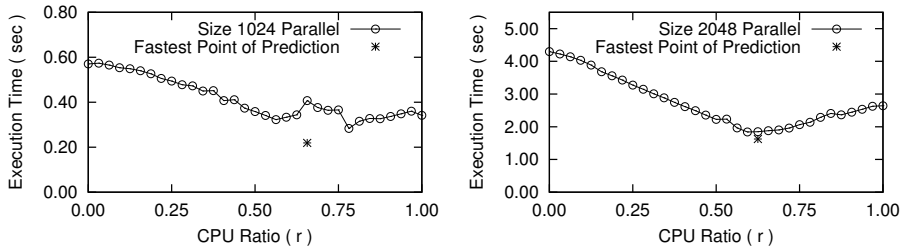
(a) Size 1024 (b) Size 2048

Fig. 10. CPU only execution time and GPU only execution time (GeForce6600GT)

6 Conclusion and Future Work

In this paper, we proposed a method for dividing a large target computation into partial computations and executing them in parallel using 1-CPU and 1-GPU. In addition, we proposed a load balancing method for minimizing the execution time. Using the method we proposed and proved by experiment for experiment. the execution time was reduced to 44.1% for the CPU and 59.5% of that for the GPU. In addition, we demonstrated that the proposed method could be used to predict the optimal assignment ratio to the CPU and GPU according to each execution time.

Future research work is required in the following problem areas. First, we must check how useful our method is in heterogeneous environments. Therefore, it is necessary to evaluate performance in various environments. Secondly,



(a) Size 1024 (b) Size 2048

Fig. 11. Execution time measured on the implemented on parallel system with a CPU and a GPU, and fastest point of prediction (GeForce6600GT)

it is necessary to evaluate the arithmetic precision of the computation using our method. In particular, we have to evaluate the differences in precision for calculated results between the CPU and GPU. Thirdly, a library of parallel programming must be developed for CPU and GPU research. We have to write CPU and GPU programs independently when we want to execute parallel programs using CPUs and GPUs. However, it is desirable that users are not concerned about whether they use CPUs or GPUs.

We are considering developing such an automatic performance tuning library. Various applications are speeded up when a library automatically assigns computations to CPUs and GPUs using our load balancing method in parallel processing. For example, if we make a library with an interface of BLAS, it can automatically assign computations to the CPU and GPU, and many applications using BLAS can be speeded up easily.

The method we proposed can be applied for more complex environments having multiple CPUs and GPUs. Utilizing such a multiple processor environment will become a new trend in GPU technology for the benefit of many CPUs and GPUs. In such environments, new approaches for realizing optimal load balancing are required to achieve the maximal speed up in the high-performance computing field.

References

1. gpgpu.org: General-Purpose computation on GPUs(GPGPU), http://gpgpu.org/
2. Thompson, C.J., Hahn, S., Oskin, M.: Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis. In: Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture. (2002) 306–317
3. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A Survey of General-Purpose Computation on Graphics Hardware. In: Eurographics 2005, State of the Art Reports. (2005) 21–51
4. Higham, N.J.: Exploiting Fast Matrix Multiplication Within the Level 3 BLAS. ACM Transactions on Mathematical Software **16** (1990) 352–368

5. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated Empirical Optimization of Software and the ATLAS Project. Parallel Computing **27** (2001) 3–35
6. John Montrym, H.M.: THE GEFORCE 6800. IEEE MICRO 2005, Vol.25, No.2 (2005)
7. Fernando, R.: GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Addison-Wesley Pub (Sd) (2004)
8. SHINOMOTO, Y., MIWA, S., SHIMADA, H., MORI, S.I., NAKASHIMA, Y., TOMITA, S.: Consideration for Speculative Rendering in PVR. In: IPSJ SIG Technical Reports. 2005-ARC-164 (2005) 145–150
9. T.Amada, M.Imura, Y.Yasumuro, Y.Manabe, K.Chihara: Partivle-Based Fluid Simulation on GPU. In: ACM Workshop on General-Purpose Computing on Graphics Processors. (2004)
10. Ádám Moravánszky: Dense Matrix Algebra on the GPU, ShaderX2 (2003)
11. Krüger, J., Westermann, R.: Linear Algebra Operators for GPU Implementation of Numerical Algorithms. In: Proceedings of ACM SIGGRAPH 2003. (2003) 908–916
12. Moreland, K., Angel, E.: The FFT on a GPU. In: Proc. SIGGRAPH/EUROGRAPHICS Workshop Graphics Hardware. (2003) 112–119
13. Hillesland, K., Lastra, A.: GPU floating-point paranoia. In: Proceedings of GP2. (2004)
14. Larsen, E., McAllister, D.: Fast matrix multiplies using graphics hardware. In: Proceedings of the 2001 ACM/IEEE conference on Supercomputing. (2001)
15. K.Fatahalian, J.Sugerman, P.Hanrahan: Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. In: Graphics Hardware 2004. (2004)
16. Jesse D. Hall, Nathan A. Carr, J.C.H.: Cache and Bandwidth Aware Matrix Multiplication on the GPU . Technical report, University of Illinois Dept. of Computer Science (2003)
17. Jiang, C., Snir, M.: Automatic Tuning Matrix Multiplication Performance on Graphics Hardware. In: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05). (2005) 185–196
18. Blackford, L.S., Hammarling, S., Cleary, A., Petitet, A., Whaley, R.C., Demmel, J., Dhillon, I., Ren, H., Stanley, K., Dongarra, J.: Practical experience in the numerical dangers of heterogeneous computing. ACM Transactions on Mathematical Software (TOMS) **23** (1997) 133–147
19. Microsoft: DirectX and High-Level Shader Language(HLSL), http://msdn.microsoft.com/directx/