gPot: Intelligent Compiler for GPGPU using Combinatorial Optimization Techniques

Yuta TOMATSU, Tomoyuki HIROYASU, Masato YOSHIMI, Mitsunori MIKI
Graduate Student of School of Ewngineering, Faculty of Department of Life and Medical Sciences, Faculty of Department of Science and Engineering
Doshisha Univ., Kyoto
ytomatsu@mikilab.doshisha.ac.jp

Key words : **GPU, Optimization, High Performance Computing**

1. INTRODUCTION

Since improving accelerating operation frequency of general purpose needs huge cost of consuming power, general purpose processors have tried to improve their performance by parallel processing on multiple processing cores since 2002. At the same time, many approaches to apply conventional dedicated multimedia processing devices to general purpose computing. Multimedia processing devices generally have many numbers of cores and if these cores can be used for calculation, it is very easy to improve computational performance using multimedia processing devices. Graphical Processing Unit (GPU) is one of the most famous dedicated multimedia processors in a personal computer [3][5][9]. GPU can be regarded as a type of many-core processor which has hundreds of processing cores. One of the reasons why GPU has many numbers of cores is that the real-time rendering for 3D model requires vast amount of quite simple arithmetic including massively parallelism [5].

Large scale PC cluster Nebulae in China ranked second at Top500 [7], the supercomputer ranking project at June, 2010. The performance of 1.271 PFlops is marked by the hybrid system of Intel's Xeon processors and NVIDIA's GPUs called Tesla. Moreover, utilization of GPU as a low-cost and high-performance accelerator is expanding including TSUBAME, a campus grid in Tokyo Institute of Technology and GPU cluster in Nagasaki University [10][11].

The information and technologies executing General Purpose computing on GPU (GPGPU) is opened to public and development environment is provided by primary GPU vendors, such as NVIDIA and ATI. While promoting improvement of the environment, difficulty of implementing software codes for GPGPU remains as a one of serious problems. High and unique programming skill is required to exploit performance of the hybrid system consist of CPU and GPU. When program developers have conventional sequential codes and they would alter their codes into parallel version, they have to change implementing codes for GPU and exploiting parallelism in the code one by one. Several researches to reduce developers' liability are progressing. For example, there are a framework to judge operation and executing device through specifying codes which can be executed as stream operation [12], and a compiler which generates binaries executed on GPU and CPU automatically by injecting directives to codes including parallelism [4]. Even when these parallel tools help program developer to create parallel programs automatically, defects of GPU programming still exists. Since network performance between CPU and cores of GPU is different and cache size of GPU is small, some parts of program should be performed in CPU and the others in GPU. Thus, when program is performed in parallel using CPU and GPU, the optimized parallelism should be obtained. This is very difficult task.

To solve this problem, we propose a system called GPU Parallel Optimization Tool (gPot), which accelerates general C-code used GPU partially. gPot optimizes computing time by circulating searching code regions which should be executed on GPU reducing the developing cost. This paper also discusses tuning technique for gPot based on quantitative performance derived by preliminary evaluation of Genetic algorithm. The structure of this paper is as follows: First of all, Section 2 describes the overview of GPU computing which this study focused on. Section 3 introduces a proposal procedure called gPot to optimize program codes for GPU. Test benches to evaluate the affectivity of gPot are explained in Section 4, and the results are discussed in Section 5. Finally this preliminary study is concluded in Section 6.

2. OVERVIEW OF GPU COMPUTING

2.1 Graphical Processing Unit (GPU)

GPU stands for Graphical Processing Unit, is well known hardware as an accelerator for 3D or 2D graphics applications. Fig.1 shows the structure of general architecture for NVIDIA's GPU consists of following components:

- Texture Processor Cluster(TPC)
- Thread Execution Manager
- Device Memory
- L2 Cache

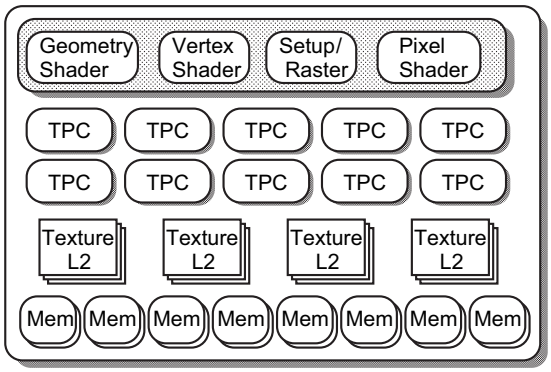

Fig.1 Outlines of GT200 series architecture

As shown in Fig.1, Texture Processor Clusters (TPCs) are arithmetic processing units, which have some internal computation cores. GTX280 has 10 TPCs each of which provides Texture Filtering Unit(TF) and some Streaming Multiprocessor(SM). Each SM consists of:

- 8 Streaming Processors(SPs):
  single floating-point processing pipelines
- 2 Special Function Units(SFUs):
  processing pipelines for transcendent function
- A shared memory:
  an on-chip memory shared by 8 SPs

2.2 CUDA Programming and PGI Accelerator

CUDA stands for Compute Unified Device Architecture provided by NVIDIA is typical development environment for GPU. CUDA is accessible to software developers though variants of industry standard programming languages: C language [2].

Program developers using GPU accelerator must have various skills of GPU programming: controls to send data between Device and Host, invoking the kernel function to process on Device, C programming skill of high level. To reduce developers' liability, there is a compiler which generates binaries executed on GPU and CPU automatically by injecting directives to codes including parallelism. Here, we introduce the PGI Accelerator.

PGI analyzes holistic program structure and data automatically, and separates codes for GPU and CPU based on directives injected to codes including parallelism [4].

Fig.2 shows the example of how to use PGI directive. When we set "#pragma acc region" on a loop code including parallelism and compile this source-code with PGI, PGI generates optimal binaries executed on GPU.

```
1 #pragma acc region
2 for( i=0;i<N;i++){
3   for( j=0;j<N;j++ ){
4     for( k=0;k<N;k++ ){
5       c[i*N+j]=c[i*N+j]+a[i*N+k]*b[k*N+j];
6 }}}
```

Fig.2 The example of how to use PGI directive

3. gPot: PROPOSAL OF AN ACCELERATING APPROACH BY GPU

3.1 Proposal method

gPot is a system to produce optimized parallel binary on the calculation environment using CPU and GPU. Program developers prepare their C codes and gPot optimize codes region processing on GPU by

evaluating a variation of execution times and produce binaries shown in Fig.3.

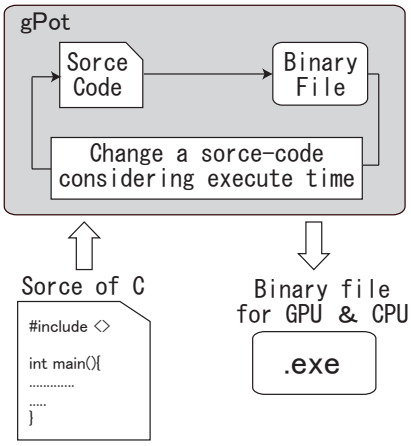

Fig.3 Overvier of gPot

3.2 Algorithm of gPot

gPot optimizes codes area processing on GPU as following sequences:

1. gPot loads source code.
2. Code analyzer finds the candidates of blocks which can be performed in Parallel.
3. Optimizer chooses the blocks which is suitable for performing to run on CPU and GPU

Current version of gPot only supports C language. Code analyzer only extracts "for" sentence from code, and makes a list of "for" sentence. In the future work, other parallel way will be performed. As optimizer, Genetic Algorithm (GA) is utilized. GA is one of strong optimization methods and GA simulates mechanisms of heredity and evolution of creatures. GA operation is a process generating child-codes by "Select", "Crossover", and "Mutation" of Genetic Algorithm (GA). As the number of combinatorial pattern of "for" getting larger according to the number of "for" sentences in an input program, gPot finds an optimum pattern of combination by GA. gPot intends the list of "for" for bit-vector as GA's genetic information. Fig.4 shows an example of injecting PGI directives. The genetic information of gPot is "[1,0,1]" when gPot inject directives to "for" of 1 and 5 line's in Fig.4 program.

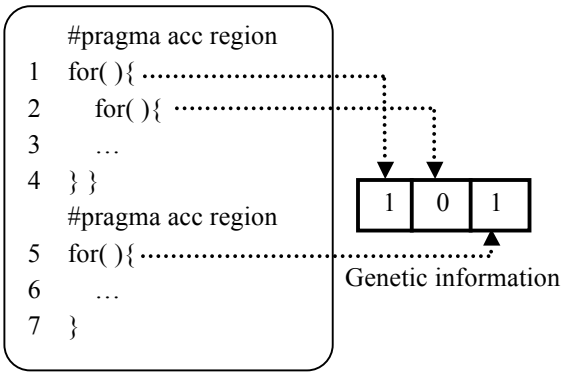

Fig.4 Example of genetic information when gPot injects PGI directives

3.3 gPot implementation

gPot is developed by CUDA run on Host and Device: Host is CPU, and Device is GPU. In CUDA, we need to invoke Device's process as the kernel function from Host. When we invoke this function, we assign the number of threads which process on Device using Grid and Block defined on the basis of CUDA [3]. Fig.5 shows an example of invoke the kernel function. In actuality, we need to describe instruction word allocating memory and sending data at 9 lines in Fig.5 because we must send data to Device's memory.

```
1 /* function processing on GPU */
2 __global__ void vecAdd(float *a,float *b,float *c){
3       int tid=threadIdx.x;      /* Getting thread ID */
4       c[tid]=a[tid]+b[tid];
5 }
6
7 /* main function on CPU */
8 int main(void){
9       …
10      vecAdd <<< 1, 256 >>> (dA,dB,dC);
11      …
12}
```

Fig.5 Example of invoke the kernel function

4. TEST SUITE

To evaluate gPot, a test program including 28 loops is examined computational time with various parameters in it.

4.1 Evaluation environment

The evaluation environment with GeForce 8400 GS and AMD Opteron P1210 shown in Table.1 is used for evaluation.

Table.1 Specification of evaluation platform

The number of CPU core	2
Core clock of CPU(GHz)	1.0
Memory size (GM)	2.5
The number of SM/SP in GPU	1 / 8
Core clock of GPU(MHz)	450

4.2 Test program adapted to gPot

Test program consists of 28 loop sentences shown in Fig.8 from Fig.10. Fig.7 shows a pseudo code of the test program. In Fig.8 from Fig.10, INDI, N, and GENIC are parameters of the test program.

In this evaluation, Fig.6 shows a genetic information which depends on loop structure of the test program. In genetic information, "1" means setting PGI directive on loop in relation to this potion.

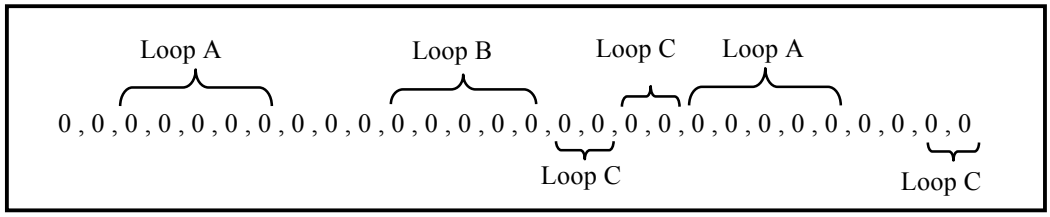

Fig.6 Genetic information in this evaluation

```
Loop A

FOR 世代=1 to 500
    Loop B
    Loop C
    Loop C
    Loop A
    Loop C
```

Fig.7 Simple code of test program

```
1  FOR i=0 to INDI
2      FOR j=0 to GENIC
3      FOR j=0 to 10
4          FOR k=0 to GENIC/N
5      FOR j=0 to 10
```

Fig.8 Program of loop A

```
1  FOR i=0 to INDI
2      FOR j=0 to 4
3          FOR k=i to j
4      FOR j=0 to 4
5      FOR j=0 to GENIC
```

Fig.9 Program of loop B

```
1  FOR i=0 to INDI
2      FOR j=0 to GENIC
```

Fig.10 Program of loop C

Table.2 Parameters of the test program

N	10
INDI	400
GENIC	100

Table.3 Pattern 1 of gPot's parameters

Mutational rate	0.1, 0.5, 0.8
Population size	30

Table.4 Pattern 2 of gPot's parameters

Mutational rate	0.1
Population size	20,30,40

4.3 Turning parameter of gPot

gPot has three parameters; a mutation rate, a population size, and a maximum step size. A population size shows *P* mentioned in section 3.1. The location of injected PGI directive is randomized according to mutation rate.

In this section, target program with parameters shown in Table.2 is evaluated by gPot with parameters; mutation rate, population size and maximum step are set as 0, 0 and 30, respectively.

Fig.11 and Fig.12 show a history of execution time of elite-code with parameters shown in Table.3 and Table.4.

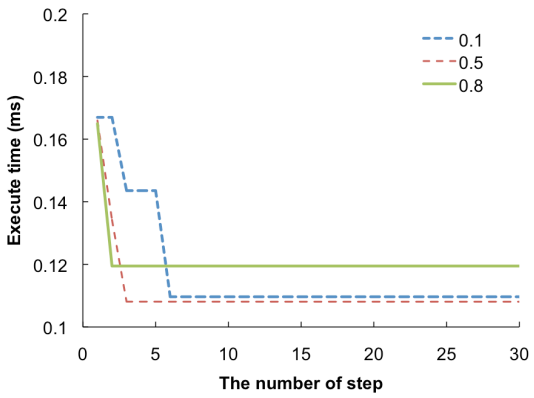

Fig.11 History of execute time with parameters of pattern 1

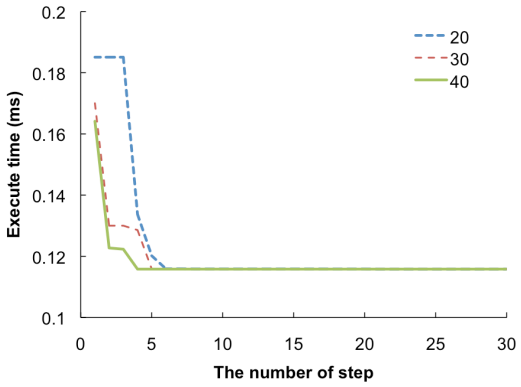

Fig.12 History of execute time with parameters of pattern 2

4.4 Change parameter of the test program

Here, we introduce an environment where we examined advancement of elite-file's execution time when we change parameters of the test program. Table.5 shows parameters of the test program, and Table.6 shows parameters of gPot.

Fig.13 shows an advancement of elite-file's execution time with each parameters of the test program when we change its parameters. Legends in Fig.13 show GENIC, and an abscissa axis in Fig.13 shows INDI.

Table.5 Parameters of the test program

N	10
INDI	20, 40, 60, 80, 100
GENIC	50, 100
Max trial number	500

Table.6 Parameters of gPot

Mutational rate	0.1
Population size	30
Max number of step	20

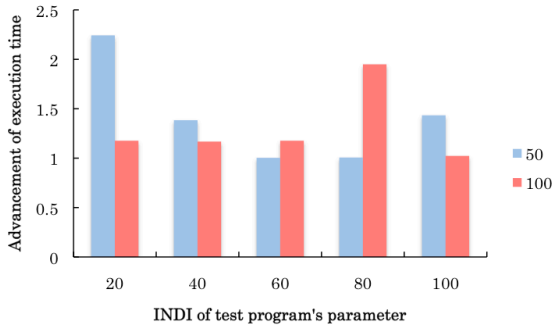

Fig.13 Advancement of execution time with each parameters of test program

5. DISCUSSION

5.1 Discussion of gPot parameters

As a mutation rate in gPot getting smaller, gPot requires more number of steps until the convergence. Fig.11 shows the execution time when mutation rates are 0.8 and 0.5[ms] is later than 0.1 and 0.5, respectively. Table.7 shows genetic information in elite-code with mutation rate of 0.1, 0.5, and 0.8. A elite-code with mutation rate of 0.8 parallelizes loop sentence of third line in Fig.9. A cause of larger computational time is the number of circulation as this loop circulates up to 10 times, while its external loop is repeated 400 times. It indicates that an extracting advantage from parallelization is more serious than time to data communication CPU and GPU to call kernel function. As a mutation rate of 0.8 changes quite a lot of the genetic information, gPot is not able to preserve appropriate location of loop. This is almost same with random selecting. In this case, a mutation rate of 0.5 shows the best performance.

Table.7 Genetic information of elite-file

Mutational	Genetic length
0.1	01100000001000010101000000000
0.5	00100000001000000101000000001
0.8	01100000001000000100010000000

As shown in Fig.12, A number of step to convergence makes small according to a population size in gPot is getting larger. It caused that the number of evaluation in a step is larger compared to execution with smaller size of population. In this case, the

smaller size of population is preferable such as 20. It is because the number of steps to convergence required almost same among various size of population.

5.2 Influence of parameters in the test program

Fig.13 shows that the fastest elite-code has parameters whose INDI and GENIC are 20 and 50, respectively. On the other hand, the slowest one has parameters whose INDI and GENIC are 60 and 50. Their genetic information are shown in Table.8. "A" in Table.8. is the fastest elite-code, and "B" is the latest one. Fluctuations of execution time of "A" and "B" are shown in Fig.14.

Table.8 Genetic information

Elite-file	Genetic information
A	01100010001000000010100000000
B	00000000000000000010000000000

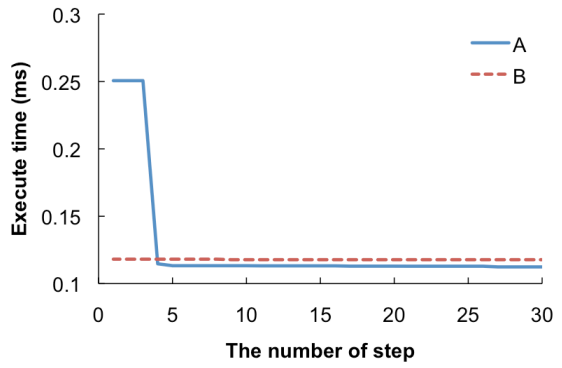

Fig.14 Execution time's history of A and B elite-files

As shown in Fig.14, genetic information of "A" is similar to the mutation rate of 0.1 and 0.5 in Table.7, while "B" parallelizes only one loop sentence. This result shows that "B" did not operate "Selection" in GA operation appropriately. Fig.14 also shows that history of execution time in "B" unchanged from the beginning. As a result, recent version of gPot has been sensitive for an execution time of elite-code at first step.

6. CONCLUSTION

We propose a system called GPU Parallel Optimization Tool (gPot), which accelerates general C-code using GPU partially. A test program including 28 loops is examined with various parameters of test program and gPot as a preliminary evaluation. We found that gPot has a possibility to accelerate the general C-code. At the same time, several types of problems of gPot are made cleared. In the future work, we will work on solving these problems.

REFERENCES

[1] V. Volkov and J. W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebla", SC'08, 2008.

[2] NVIDIA, "CUDA Technical Traning Volume I", 2008.

[3] NVIDIA, "CUDA Programming Guide 2.3", 2009

[4] The Portland Group, "PGI Fortran & C Accelerator Programing Model", ver 12, Mar. 2010

[5] Fujimoto.N, "Dense Matrix-Vector Multiplication on the CUDA Architecture", Parallel Processing Let- ters, Vol. 18, No. 4, pp. 511–530, 2008

[6] Akihiro Shitara, "Implementation and Evaluation of Self-organizing Map Algorithm on a Graphic Processor", Parallel and Distributed Computing and Systems, track:668-027, 2009

[7] "Top 500", http://www.top500.org (accesed Jul 21, 2010)

[8] Paul R. Dixon, "Harnessing graphics processors for the fast computation of acoustic likelihoods in speech recognition", Computer Speech & Language, vol. 23, no. 4, pp. 510–526, Oct. 2009

[9] Patrick Cardinal, "GPU Accelerated Acoustic Likelihood Computations," in Proc. of INTERSPEECH, 2008

[10] Satoshi Matsuoka, "GPU accelerated computing–from hype to mainstream, the rebirth of vector computing", Journal of Physics: Conference Series, vol. 180, no. 1, 2009

[11] Ali Cevahir, "High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning", Computer Science - Research and Development, vol. 25, no. 1-2, pp. 83-91, Apr. 2009

[12] Buck, I., et al., "Brook for GPUs : Stream Computing on Graphics Hardware", ACM Transactions on Graphics, vol. 23, no. 3, pp. 777-786, Aug. 2004