This paper is author's private version of the paper published as follows:
Proc. 3rd Int'l Conf. on Networking and Computing (ICNC'12), pp.208-214
Copyright (C) 2012 IEEE
DOI: 10.1109/ICNC.2012.39

An Automatic Host and Device Memory Allocation Method for OpenMPC

Hiroaki UCHIYAMA*, Tomoaki TSUMURA* and Hiroshi MATSUO*

*Nagoya Institute of Technology
Gokiso, Showa, Nagoya, Japan
Email: camp@matlab.nitech.ac.jp

***Abstract*—The CUDA programming model provides better abstraction for GPU programming. However, it is still hard to write programs with CUDA because both some specific techniques and knowledge about GPU architecture are required. Hence, many programming frameworks for CUDA have been developed. OpenMPC is one of them based on OpenMP. OpenMPC is an easy-to-write framework for programmers familiar with traditional OpenMP, but still requires programmers to use the special directives for utilizing fast device memories. To solve this problem, this paper proposes a method for allocating appropriate device memories automatically. This paper also proposes a method for automatically allocating page locked memory for the data which are transferred between host and device. The evaluation results with several programs show that proposed methods can reduce 52% execution time in maximum.**

***Index Terms*—GPGPU, CUDA, OpenMPC, memory allocation**

I. Introduction

Graphics Processing Unit (GPU) is a specialized processor for image/video processing. It generally has wide memory bandwidth and high processing performance. The GPGPU, which means general purpose computation on GPU, is now in demand. Hence, for writing the general purpose programs which are executed on GPU, Compute Unified Device Architecture (CUDA)[1] is developed by NVIDIA.

The CUDA model achieves speedup by mapping a large number of threads to the cores on GPU. The GPU cores can execute the same instructions in parallel on their own threads. However, to write programs with CUDA, programmers should explicitly describe thread allocation and data transfer between CPU and GPU. Therefore, deep knowledge about GPU architecture and CUDA are required. To solve this problem, many programming frameworks for CUDA have been developed. They can reduce the burden on programmers by freeing them from the special directives of CUDA.

In this paper, we focus on one of the frameworks, **OpenMPC**[2]. OpenMPC is based on OpenMP[3]. Therefore, programmers who are familiar with traditional OpenMP can use OpenMPC with little effort. However, to write efficient CUDA programs, OpenMPC still requires programmers to use special directives for utilizing fast device memories. In addition, the data transfer between CPU and GPU is not optimized enough on OpenMPC, because there occur redundant copies on host memory.

To solve these problems, this paper proposes two methods. The one is for automatically allocating appropriate device memories for data. The other is automatically allocating page locked memory for the data which are transferred between CPU and GPU. With these methods, we aim to speed-up programs and to reduce the complexity of CUDA programming.

II. Programming Environments for GPU

In this section, we will give an overview of the CUDA programming model and some programming frameworks for CUDA. In addition, we describe OpenMPC the target of our study.

A. CUDA

Now, NVIDIA has provided CUDA for GPU programming. CUDA has been developed as an integrated development environment for GPUs. A program written with CUDA consists of a host code and a device code. The host code is executed on CPU, and the device code is executed on GPU. The host code is a sequential code for CPU, and it has some kernel function calls in it. The kernel functions are defined in the device code and executed in parallel on GPU.

A brief architecture of a GPU shipped by NVIDIA is shown in Fig. 1. Although GPUs are different each other according to their generations or families, they have a common architecture. One GPU has dozens of Streaming Multiprocessors (SM), and each SM has eight Streaming Processors (SP). GPU can achieve high performance by executing massively parallel threads simultaneously using SPs. In the CUDA framework, a GPU can execute $65535 \times 65535 \times 512$ threads across all SPs. These threads are grouped hierarchically, as shown in Fig. 2. A set of threads is called a *Block*, and a set of blocks is called a *Grid*. A definition of the number of the threads per Block and the number of the blocks per Grid is called as an *execution configuration*. This can be specified at each call of kernel functions by using the CUDA language extensions. Each thread has a unique ID, and an address which is accessed by the thread can be specified by using the built-in variable for the ID.

The CUDA memory model is shown in Fig. 3. In the CUDA memory model, the memory of CPU is called *host*

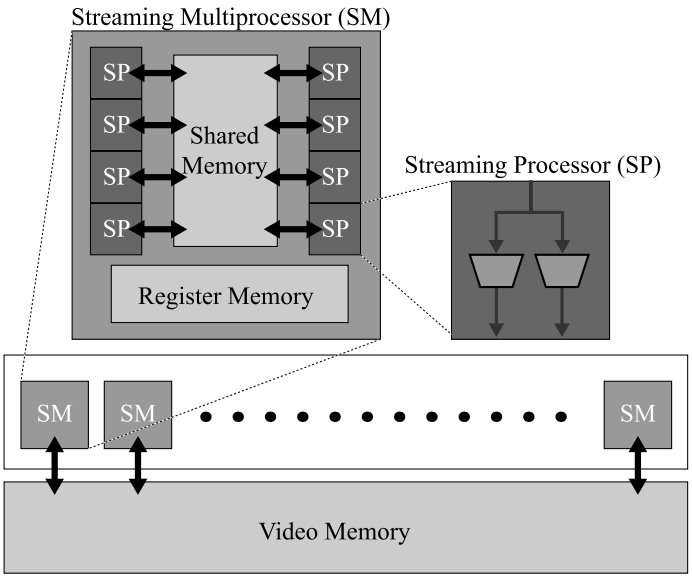

Fig. 1. Brief architecture of GPU

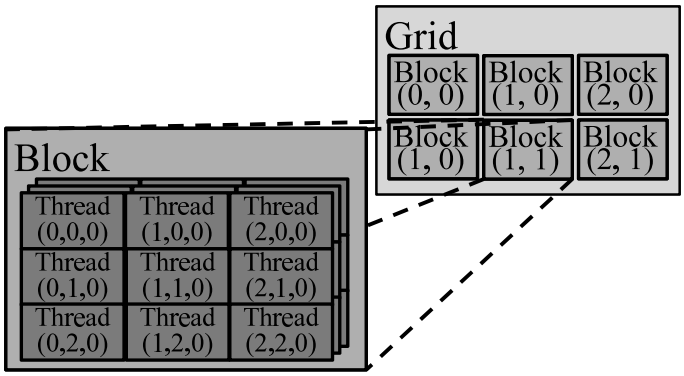

Fig. 2. Hierarchical thread management

memory. On the other hand, the memories on GPU are called *device memories*. The CUDA model assumes that each of CPU and GPU has a separate and unique address space. For sharing data between CPU and GPU, CUDA provides some APIs for explicit GPU memory management, including functions for transferring data between CPU and GPU. There are several types of device memories; register memory, local memory, shared memory, global memory, texture memory and constant memory. Their characteristics such as the physical location, whether they have caches or not, and so on, are different. Therefore, if programmers can use more appropriate device memory considering the characteristics, the program will run faster. However, some knowledge about different device memories are required for achieving good performance.

B. Related Work

When writing programs with CUDA, it is necessary to indicate explicit data transfer between host memory and device

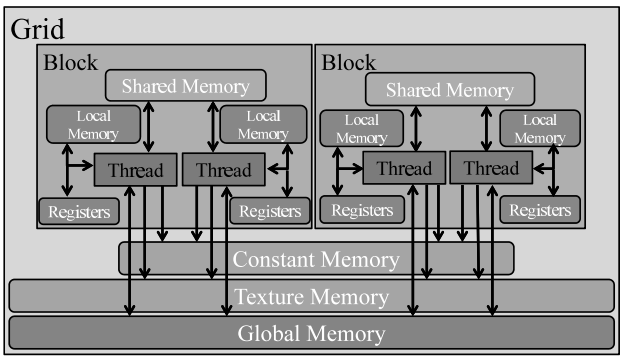

Fig. 3. The CUDA memory model

memories, and to specify an execution configuration. Moreover, to write efficient GPU programs, some knowledge of GPU architecture is required. However, it is rather difficult for average programmers.

Hence, some frameworks for CUDA programming have been proposed. Baskaran et.al.[4] has proposed a compiler framework for optimizing CUDA programs. It can make global memory accesses effective. However, the compiler framework can optimize affine loop nests only.

CUDA-lite[5] is a translator for CUDA programs. With CUDA-lite, programmers can easily get the programs which are optimized about memory accesses and kernel configurations. However, it still requires programmers to specify the data transfers and to define kernel functions.

Another framework called hi-CUDA[6], [7] is also proposed to provide abstraction of CUDA. It provides some dedicated pragmas for specifying which data should be transferred between CPU and GPU, or for specifying which computation block should run on GPU. The hi-CUDA compiler then generates CUDA programs automatically by parsing and interpreting the pragmas. However, it is still necessary for programmers to learn the syntax of hi-CUDA pragmas, and then, to use them for specifying data transfers and kernel computations.

C. OpenMPC

For programmers' convenience, many frameworks for CUDA have been proposed as shown in II-B. However, some specific knowledge of GPU architecture and the parallel programming model are still required.

Hence, a new framework OpenMPC has been proposed. In OpenMPC programs, OpenMP pragmas are used for indicating parallelization. OpenMP is a parallel programming framework, which has been widely used for computation on CPUs. With OpenMP, programmers can indicate a computation block, which they want to parallelize, in the source code by using pragmas. The OpenMP compiler automatically generates parallelized codes by parsing and interpreting the pragmas. On the other hand, OpenMPC can achieve speedup by defining the indicated block as a kernel function, and then executing the kernel function in massively parallel on GPU.

In addition, OpenMPC compiler automatically detects the variables which are used in the indicated blocks. The compiler then generates data transfer codes automatically. Therefore, programmers do not have to consider whether a variable is transferred or not. Moreover, an execution configuration is specified automatically by considering the number of the loop iteration in the block. As a result, programmers can write CUDA programs without knowledge of GPU architecture or parallel programming model.

However, OpenMPC has some problems. The one is that OpenMPC cannot use appropriate device memories automatically. In the CUDA programming model, programmers can use several types of device memories, and the way to use device memories is important for achieving enough performance with GPUs. However, without specifications given by programmers, OpenMPC uses global memory which is the slowest in device memories. In order to use device memories other than the global memory, it is necessary to explicitly specify which memory should be used, by using pragmas.

In addition, data transfer between host memory and device memories is inefficient with OpenMPC. With CUDA, the data transferred between host memory and device memories should be located in host page locked memory, where the data is not swapped out by operating systems. Therefore, the data in pageable memory should be copied to the page locked memory before being transferred. To prevent this copy, CUDA provides the functions which directly allocate page locked memory for data. However, OpenMPC has no interface for these functions, and the data should be always copied.

III. Automatic Use of Appropriate Memories

In this section, we will propose a method for automatically allocating appropriate device memories for data. Also, we will propose another method for automatically allocating page locked memory for data which are transferred between CPU and GPU.

A. Automatic Allocation of Appropriate Device Memories

Device memories have some different characteristics each other. Hence, it is important to properly use the memories. However, programmers should understand the GPU architecture deeply for using appropriate device memories. Therefore, we propose a method for automatically allocating device memories which can be accessed fast. This method allows programmers to generate efficient code without specifying which memories should be used.

TABLE I shows the characteristics of device memories which can be accessed faster than global memory. The details are shown below.

shared memory
: Shared memory is accessible only from all the threads in the associated block. It is the only writable memory among the three kinds of memories. Shared memory is suitable for storing Block-local temporary data, since 16KB of shared memory is assigned to each SM. However, when accessing the data on shared memory, synchronization in the Block is required.

texture memory
: Texture memory is accessible from all threads in the associated Grid. Both read and write are available from CPU but only read is available from GPU. The data stored in texture memory are called *texture* and accessed as a 1-3 dimensional array. The size depends on products, but maximum width of a texture in texture memory is 2^{27}. The texture should be accessed through the special functions which are provided by the CUDA library. In addition, the data type which can be stored in texture memory must be signed int, unsigned int, or float.

constant memory
: Constant memory is accessible from all threads in the associated Grid as well as texture memory. The permission for read and write is also same as texture memory. The size of the constant memory is only 64KB.

TABLE I
Characteristics of Device memories

name	cache	permission	access	size
Global Memory	n/a	R/W	slow	depends on products
Shared Memory	-	R/W	fast	16KB/SM
Texture Memory	available	R	fast	depends on products
Constant Memory	available	R	fast	64KB

In this paper, we propose a method for automatically using texture memory and constant memory among these three memories. In our method, the data for which texture memory or constant memory can be allocated are automatically detected, and programmers can use appropriate memories without considering their characteristics.

B. Automatic Allocation of Page Locked Memory

The data, which are transferred between host memory and device memories, must be located in page locked memory for not being paged out by the operating system. Therefore, the data which are located in pageable memory must be copied to page locked memory before being transferred. However, if the data had been located in page locked memory in advance, the data need not to be copied into page locked memory. The functions for allocating page locked memory for data are provided by CUDA library. However, allocating too much page locked memory causes a performance degradation, because page locked memory is not paged out by the operating system. Therefore, we propose a method for detecting the data which are transferred between host memory and device memory, and allocating page locked memory only for such data.

The data flow between host memory and device memory is shown in Fig. 4. With OpenMPC, the data which will be transferred is located in pageable memory in host memory. Therefore, (1) the data are copied into page locked memory first, and then, (2) transferred to device memory. When CPU receives the computation results from GPU, (3) they are

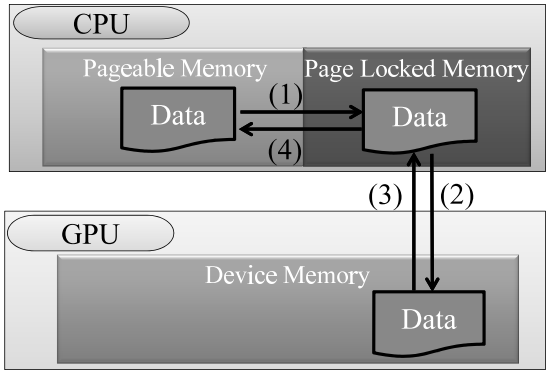

Fig. 4. Data transfer between host memory and device memory

transferred from device memory to page locked memory, and then, (4) they are copied again from page locked memory to pageable memory. Whenever the data are transferred between host memory and device memory, this copy overhead occurs.

On the other hand, some expressions for allocating page locked memory for data with CUDA library function are generated automatically in the proposal method. Therefore, redundant copies such as (1) and (4) in Fig. 4 is omitted, and the data can be transferred directly and fast.

IV. Improvement of the OpenMPC Compiler

We have improved the OpenMPC compiler to implement the proposal methods. In this section, we describe the compile flow of the improved compiler, and how to implement the proposal methods.

A. Flow of the Compilation Process

OpenMPC provides a compiler which translates OpenMP programs into CUDA programs, and the special pragmas which are used for optimizing the output CUDA program.

Fig. 5 shows the processing flow of the improved compiler. We added two steps, which are filled with dark grey in Fig. 5, for the proposal methods in the compiler. The existing OpenMPC compiler receives OpenMP program as its input and parses it. Then, the compiler generates an Intermediate Representation (IR) tree. After that, the compiler generates a CUDA program from the IR tree.

On the other hand, in the proposal method, the compiler further parses the IR tree, and detects transferred variables and kernel function calls. Then, the compiler generates a CUDA program which can use appropriate memories.

B. Allocating Fast Device Memories

As shown in TABLE I, constant memory and texture memory are read-only from GPU. Therefore, only the data which are not overwritten in kernel functions can be located in these memories. In other words, for allocating these memories for a datum, the associated variable must satisfy both of the two conditions:

- not on the left side of any assignment operator in any statement

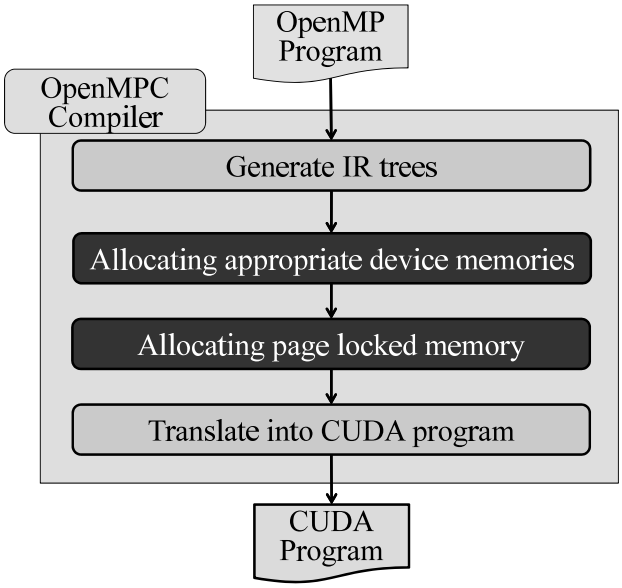

Fig. 5. The compile flow with proposal methods

```
1   void kernel_1(int *a,int *b,int *c , int *d) {
2       //compute the index of array
3       int i = threadId.x + blockId.x * blockDim.x;
4
5       a[i] = b[i] + c[i];// assignment
6       c[i] += ++ d[i]; // assignment and increment
7   }
```

Fig. 6. Detection of the presence of assignment or increment or decrement

- not on the left side of any increment/decrement operator in any statement

In addition, a variable which is used as an argument for other function call in the kernel function cannot be located in such device memories. Because such a variable may be overwritten through the execution of the function body. Structure variables also cannot be located in fast device memories, because their data size cannot be estimated.

Now, we explain the method for determining whether a variable is used as a target of assignment, increment or decrement, or not. Fig. 6 shows a sample program, and Fig. 7 shows the IR tree which is generated from the fifth line of the program shown in Fig. 6. Fig. 8 shows the IR tree which is generated from the sixth line in Fig. 6.

Fig. 7 shows that the variable *a* is the left child of the assignment operator. Therefore, the variable *a* is excluded from candidates for variables, for which fast device memories can be allocated. However, the variable *b* and *c* are still candidates, because it is obvious from the right child of the tree that they are not overwritten.

Next, Fig. 8 shows that the variable *c* is the left child of the compound assignment operator. Therefore, the variable *c* is now excluded from the candidates. Also, the variable *d* is excluded from the candidates, because Fig. 8 shows that the variable *d* is the left child of an increment operator.

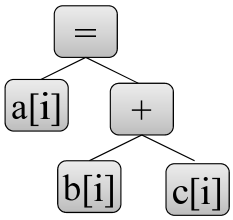

Fig. 7. Assignment statement

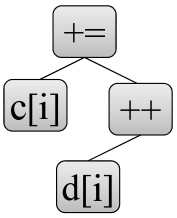

Fig. 8. Complex assignment and increment

```
1   int y[M]; // defined as an array
2   int Function_0(){
3     int x[N]; // defined an array
4     int *gpu_x;
5     int *gpu_y;
6      :
7
8
9      :
10    // xfer to device memory
11    cudaMemcpy(gpu_x,x);
12    cudaMemcpy(gpu_y,y);
13     :
14
15
16    return 0;
17  }
18
19  int main(){
20     :
21
22
23    Function_0();
24
25
26    return 0;
27  }
```

Before translated

```
1   int *y; // defined as a pointer
2   int Function_0(){
3     int *x; // defined as a pointer
4     int *gpu_x;
5     int *gpu_y;
6      :
7     // allocation
8     cudaMalloHost(&x);
9      :
10    // xfer to device memory
11    cudaMemcpy(gpu_x,x);
12    cudaMemcpy(gpu_y,y);
13     :
14    // free
15    cudaFreeHost(x);
16    return 0;
17  }
18
19  int main(){
20     :
21    // allocate
22    cudaMallocHost(&y);
23    Function_0();
24    // free
25    cudaFreeHost(y);
26    return 0;
27  }
```

After translated

Fig. 9. An example of translation to allocate page locked memory for data

Consequently, only the variable *b* is not overwritten, and the data which are stored in the array variable *b* can be located in fast device memories in this kernel function.

Whether constant memory or texture memory should be allocated for the data is decided by the type of the associated variable. The size of constant memory is only 64KB. Hence, for a variable which is not an array, constant memory is allocated. If the total size of constant memory allocation becomes over 64KB, the compiler does not allocate constant memory for any more data. On the other hand, texture memory is allocated for array variables, because texture memory can be considered to has no upper limit of its size. However, the type of variable for which texture memory can be allocated should be either 8bit, 16bit, 32bit integer or 32bit float.

C. Allocating Page Locked Memory

To transfer data efficiently between host memory and device memories, we propose a method for directly allocating page locked memory for the transferred data, using CUDA library functions. When should page locked memory be allocated and freed will vary depending on whether the associated variable is local or global.

Fig. 9 shows an example code. The variable *x* which is defined as a local variable of *Function_0()* is used as an argument of *cudaMemcpy()*. Therefore, the variable *x* will be transferred between host memory and device memory. Then, the proposal method considers that page locked memory should be allocated for the variable *x*. Consequently, in the result code of the translation, the variable *x* is defined as a pointer variable, and page locked memory is allocated for the variable by using the CUDA library function *cudaMallocHost()* (line 8). Subsequently, it is freed at the end of *Function_0()* (line 15).

On the other hand, the variable *y* is defined as an global variable, and it may be accessed in various functions. Hence, if page locked memory is allocated for the variable *y* and freed in *Function_0()*, the program may run incorrectly. Therefore, when a variable is global, page lock memory should be allocated for the variable and freed not in the function, in which the associated datum is transferred, but in the main function. Consequently, the codes for allocating page lock memory for the variable *y* and freeing it are inserted in the main function in the result of translation (line 22, 25).

TABLE II
EVALUATION ENVIRONMENT #1 (GEFORCE GTX280)

OS	Fedora15
CPU	Core2Quad
Frequency	2.83GHz
Memory	3GB
GPU	GeForce GTX280
Number of multiprocessors	30
Number of cores (SP)	240 (30 × 8)
CUDA version	4.0
Compute capability	1.3
Compiler	gcc 4.6.1
Compile options	-O3
OpenMPC Compiler version	0.31

TABLE III
EVALUATION ENVIRONMENT #2 (TESLA C1060)

OS	Fedora15
CPU	Phenom II X4
Frequency	3.4GHz
Memory	8GB
GPU	Tesla C1060
Number of multiprocessors	30
Number of cores (SP)	240 (30 × 8)
CUDA version	4.0
Compute capability	1.3
Compiler	gcc 4.6.1
Compile options	-O3
OpenMPC Compiler version	0.31

V. EVALUATION

In this section, we discuss the performance of the automatic memory allocation methods proposed in this paper.

A. Evaluation Environment

The evaluation environments are shown in TABLE II and TABLE III. We used GeForce GTX 280 and Tesla C1060 for

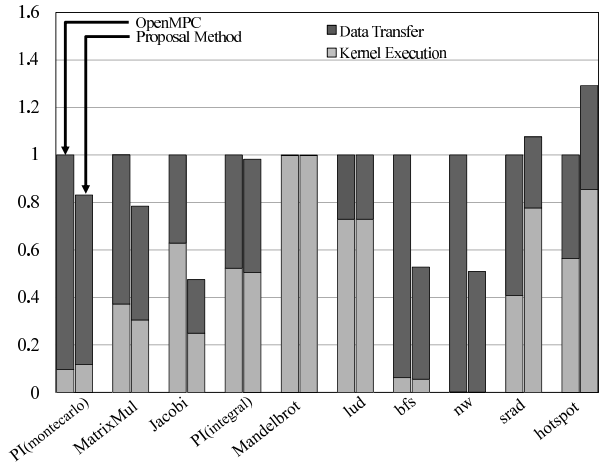

Fig. 10. Ratio of execution time with GeForce GTX280

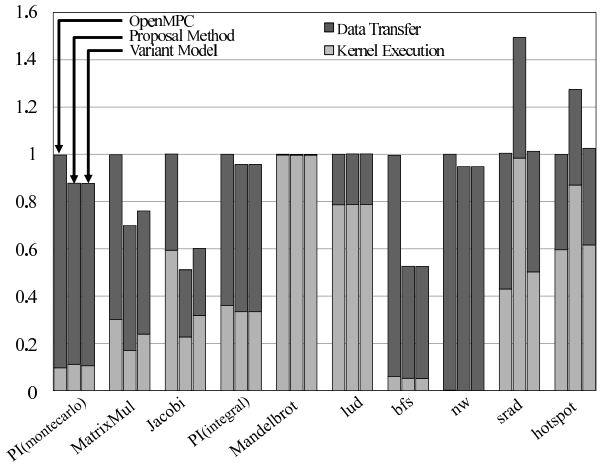

Fig. 11. Ratio of execution time with Tesla C1060

evaluation. Each of them has 30 SMs, and each SM has eight SPs.

Three sets of benchmark programs are used as workloads. The first is the set of the programs which we have originally designed. They are PI(montecarlo) which calculates π using the Monte Carlo method, and MatrixMul which calculates matrix product. The second is the set of the programs from OmpSCR[8]. They are PI(integral) which calculates π using the integral methods, Mandelbrot which estimates the area of the Mandelbrot set, and Jacobi which solves a finite difference discretization of Helmholtz equation. The third set is of the five programs from Rodinia Benchmark suites[9], [10], which are used for evaluation in [2].

As the input parameters for these workloads, the number of iterations of PI(montecarlo) is defined as 100,000, and the size of the matrix used in MatrixMul is 128 $\times$ 128. Default input values are used for PI(Integral), Mandelbrot, and the programs from Rodinia Benchmark suites. However, the size of the matrix for Jacobi is defined as 200 $\times$ 200, because the default input size 5,000 $\times$ 5,000 is too large and Jacobi can not run correctly with the input size.

B. Evaluation Results

Fig. 10 and Fig. 11 show the execution time of the workloads. We divide total execution time into two parts, kernel execution and data transfer. The result of each workload in environment #1 (GeForce GTX280) shown in Fig. 10 is represented by two bars. The left bars show the results of OpenMPC, and the right ones show the results of the proposal method. On the other hand, the result of each workload in environment #2 (Tesla C1060) shown in Fig. 11 is represented by three bars. The left bars show the results of OpenMPC, and the center ones show the results of the proposal method. The right bars show the results of a variant of the proposed model. In the variant model, we have set a limit that texture memory can be allocated for only one array variable, for efficient use of texture caches. The size of texture cache is 6–8KB, and allocating too much texture memory may cause frequent cache misses.

Each bar is normalized to the execution time of OpenMPC. The result shows that the proposal method achieves performance gain with more than half of the benchmark programs. However, the performance of srad and hotspot is deteriorated. To examine the cause of the performance deterioration with these two workloads, we confirmed their assembly codes. Through the examination, it was found that calculating indices for accessing to texture data costs much more time than OpenMPC.

Each of these two programs has a loop block in its kernel function, and accesses to an array variable in the loop block iteratively. With the existing OpenMPC, the array variables are located in global memory and the access to the variables is optimized by the CUDA compiler nvcc. The redundant access is reduced, and reloading values from the same address where is accessed before is avoided. On the other hand, the array variables are located in texture memory with the proposal method, and nvcc seems to be not able to optimize the redundant access to texture memory. Therefore, indices for accessing to texture data will be calculated and a value which has been loaded from an address will be loaded again from the same address. Hence, this causes the performance deterioration with srad and hotspot. However, the variant model can avoid the performance deterioration by using only one texture in each kernel function.

On the other hand, the proposal method can reduce execution time with more than half of the benchmark programs, and achieves maximum speed-up with Jacobi. The reason of the large performance gain with Jacobi is that Jacobi requires many data to be processed, and most of them are only read in the program. Incidentally, the variant model achieves maximum speed-up with bfs, because the limit in the variant model restrains the performance gain of Jacobi.

As a whole, in environment #1, the proposal method can reduce 15% execution time in average, and 52% in maximum. In

environment 2, the proposal methods can reduce 7% execution time in average, and 49% in maximum, and the variant model can reduce 18% in average, and 47% in maximum.

VI. Conclusion

In this paper, we proposed a method for improving OpenMPC to automatically allocate appropriate device memories. This paper also proposed a method for automatically allocating page locked memory for data which are transferred between host and device. Through the evaluation with several benchmark programs, it is found that the proposal methods can reduce 15% execution time in average, and 52 % execution time in maximum.

One of our future works is allocating fast device memories for more appropriate data. Now, the proposal methods are applied to all data which are detected as locatable in fast device memories. However, if the compiler allocates fast device memories for only the data which are accessed frequently, more programs might be faster.

References

[1] NVIDIA Corp., *NVIDIA CUDA Programming Guide*, 2nd ed., Jun. 2008.

[2] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP for Efficient Programming and Tuning on GPUs," *Int'l Journal of Computational Science and Engineering*, 2012.

[3] L. Dagum and R. Menon, "OpenMP: an Industry Standard API for Shared-Memory Programming," *IEEE Computational Science and Engineering*, vol. 5, 1998.

[4] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, A. Rountev, and P.Sadayappan, "A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs," in *ICS'08: Proc. 22nd Annual Intl. Conf. on Supercomputing*. ACM, 2008, pp. 225–234.

[5] S.-Z. Ueng, M. Lathara, S. Baghsorkhi, and W. mei Hwu, "CUDA-lite: Reducing GPU Programming Complexity," in *Proc. 21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC 2008)*, 2008, pp. 1–15.

[6] T. D. Han and T. S. Abdelrahman, "hiCUDA: a high-level directive-based language for GPU programming," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, 2009, pp. 52–61.

[7] ——, "hiCUDA: High-Level GPGPU Programming," *IEEE Trans. on Parallel and Distributed Systems*, vol. 22, pp. 78–90, 2011.

[8] A. Dorta, C. Rodriguez, and F. de Sande, "The OpenMP Source Code Repository," in *Proc. 13th Euromicro Conf. on Parallel, Distributed and Network-Based Processing (PDP2005)*, Feb. 2005, pp. 244–250.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proc. 2009 IEEE Int'l Symp. on Workload Characterization (IISWC '09)*, ser. IISWC'2010. IEEE, 2009, pp. 44–54.

[10] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *Proc. 2010 IEEE Int'l Symp. on Workload Characterization (IISWC'10)*, ser. IISWC '10. IEEE, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2010.5650274