This is the accepted manuscript of a paper published in
Proc. 2nd Int'l Symp. on Computing and Networking (CANDAR'14), pp.419-425

Automatic Code Tuning for Improving GPU Resource Utilization

Ryo TAKESHIMA* and Tomoaki TSUMURA*
*Nagoya Institute of Technology
Gokiso, Showa, Nagoya, Japan
Email: camp@matlab.nitech.ac.jp

***Abstract*—Utilizing a GPU to perform general purpose computation is called GPGPU. The high theoretical performance of GPU draws attention to GPGPU. CUDA supplies a platform for the developers of GPU applications. In CUDA programming model, massive threads are allocated to GPU's calculation units. Besides, CUDA has various kinds of memories on GPU. These memories have different features of access latency, capacity, and so on. Therefore, to produce high-performance GPU programs, developers should consider how to allocate the massive threads to cores and which memory should be used for storing data. Hence, developers should have deep understanding of the GPU architecture and CUDA APIs. To address this problem, we propose an auto tuning framework for GPU programs, and explain an implementation of a preprocessor for the framework, in this paper.**

I. Introduction

Graphics Processing Unit (GPU) is a specialized processor for image/video processing. GPU can deliver high-performance with low power consumption when computing in parallel. The high-performance of GPU attracts attention, and GPGPU, which means a general purpose computation on GPU, is now in demand. Accordingly, GPU is now widely used as an acceleration core. Therefore, GPU programming for using GPU as an acceleration core will be in more demand. For such GPU programming, some platforms have been developed. The most popular platform among them is *CUDA (Compute Unified Device Architecture)*[1].

Using CUDA allows developers to write GPU programs easily. However, tuning GPU programs is difficult, because developers should have deep understanding of GPU architecture and CUDA APIs to tune GPU programs for high-performance. In this paper, we propose an auto tuning framework, which tunes GPU programs for high-performance without requiring developers to mind GPU architecture and CUDA APIs. This framework can automatically tune GPU programs aiming to improve utilization of GPU resource. Furthermore, to conceal a complicated memory configuration of GPU from developers, this framework also can generate codes for transferring data to suitable device memories in GPU automatically.

II. Research Backgrounds

In this section, we will give overviews of CUDA and related work for helping developers to produce GPU programs.

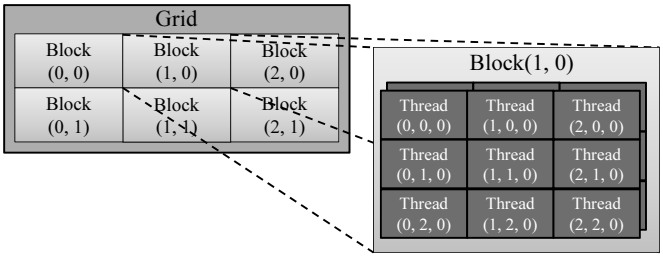


Fig. 1. Hierarchical thread management

A. CUDA

NVIDIA has provided a parallel computing platform CUDA for GPU programming. Here, we will explain the CUDA programming model, its memory configuration, and its execution model.

1) Programming Model: GPU generally has wide memory bandwidth and high processing, and CUDA-enabled GPUs have dozens of processors. The processors are called *Streaming Multiprocessors (SMs)*. Each SM has dozens of CUDA Cores, Load/Store Units, and Special Function Units (SFUs). CUDA Core is an execution core for integer and floating-point operations. Load/Store Unit is a unit for load/store operations. SFU is a unit for complicated operations such as double-precision arithmetic.

In CUDA programing model, GPU can achieve high performance by executing massively parallel threads simultaneously using SMs. These massive threads are managed hierarchically as shown in Fig. 1. A set of threads is called a *Block*, and a set of blocks is called a *Grid*. As shown in Fig. 1, Blocks in a Grid are managed two-dimensionally, and threads in a Block are managed three-dimensionally. The thread hierarchy allows developers to handle massive threads easily. A definition of the dimensions of the Grid and the Block is called an *Execution Configuration*, and the dimensions of the Block, that is, the number of the threads per Block is called *Block size*.

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. In the device code, each routine which should be executed on GPU is defined as a *kernel function*, and each kernel function call must specify an execution configuration.

2) Memory Configuration: A CUDA-enabled GPU has several types of memories; Global Memory, Texture Memory, Constant Memory, Register, Local Memory, and Shared Memory. These are known generally as *device memories*. In addition, a CUDA-enabled GPU has also several types of caches; Global Cache (L2 Cache), Constant Cache, ReadOnly Cache, and L1 Cache. Device memories and these caches have different access latencies and capacities. Moreover, APIs used for accessing data stored in them and for transferring data to them are different according to device memories. Therefore, to produce high-performance GPU programs, developers should consider which memory should be used for storing data, and should mind APIs which they should use. In other words, developers should consider the complicated memory configuration of GPU and the correct APIs, when writing CUDA programs.

3) Execution Model: As described in section II-A1, a CUDA-enabled GPU can achieve high-performance by executing massively parallel threads simultaneously. In CUDA, 32 threads in a set called *Warp* are executed simultaneously. Each Warp is allocated to an SM, and all Warps which are allocated to the SM are executed in out-of-order. However, an SM cannot execute the Warp which is allocated to itself, in the case when data, which are needed to execute the Warp, are not available, or when the execution unit, which is needed to execute the Warp, is not available. In such a case, the SM will be idle. Therefore, it is important to allocate as many Warps as possible to each SM for drawing out the GPU performance.

The number of Warps which can be allocated to one SM is limited. Nevertheless, the number of Warps which are actually allocated to one SM at run time can vary corresponding to the number of Registers used in a kernel function, an amount of Shared Memory used in a kernel function, and the number of Registers which the SM has. The Warps which are actually allocated to each SM at run time are called *Active Warps*. The more Active Warps are allocated to each SM, the more threads can be executed simultaneously. Therefore, the number of Active Warps is an important index for measuring the performance of a GPU program. In CUDA, this index is quantified as *Occupancy* defined in section 10.1 of [1], and it is calculated by dividing the number of Active Warps by the maximum number of Warps which can be allocated to each SM. In general, the performance of a GPU program is in proportion to the Occupancy. However, the number of Active Warps can intricately vary corresponding to the type of kernel function or the hardware configuration of GPU. Therefore, it is difficult to find out an Execution Configuration with which the number of Active Warps is maximized. In other words, to set an Execution Configuration which can achieve the highest Occupancy is difficult.

B. C-to-CUDA Compilers

To produce GPU programs, developers should deeply understand GPU architecture and CUDA APIs. For allowing developers to produce GPU programs easily, some C-to-CUDA compilers which translate a C programs into a CUDA program have been proposed[2], [3], [4]. For example, hiCUDA[5], [6], Bones[7], and Par4All[8] are well-known C-to-CUDA compilers. When developers use hiCUDA compiler, they have to insert some pragmas into a C program to give hiCUDA compiler some parameters which are necessary to translate. By using these parameters, hiCUDA compiler can translate C programs into CUDA programs. Nevertheless, developers should consider an Execution Configuration.

Similarly, developers have to insert some pragmas into a C program to give Bones compiler some parameters which are necessary to translation. In Bones compiler, various elementary algorithms are categorized. The categorized algorithms are called Classes. Developers give Classes, each of which corresponds to a code executed on the GPU, and the number of threads created by each kernel function to Bones compiler. By using these parameters, Bones compiler can translate C programs into CUDA programs. However, Bones compiler assigns an Execution Configuration to each kernel function without considering the number of Registers and the amount of Shared Memory used in the kernel function. That is, Bones compiler assigns the Execution Configuration considering only given Classes. Therefore, Bones compiler may assign an inappropriate Execution Configuration to a kernel function.

In contrast to these compilers, Par4All compiler can translate C programs into CUDA programs, without requiring the developers to use directives. However, Par4All compiler can translate only a loop statement only if the compiler can find that the iterations of the loop have no dependency with each other.

Therefore, Par4All compiler can translate only a part of a C program into a CUDA program. That is, a processing which is expected to be executed on GPU by developers cannot be always executed on GPU.

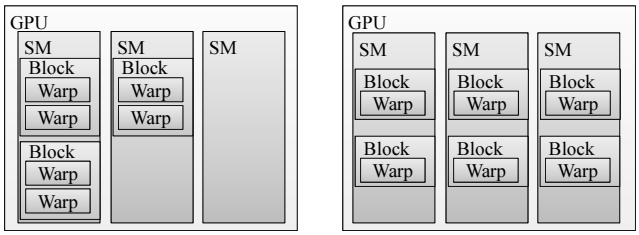
III. Design of Auto-Tuning Framework

In this paper, we propose an auto tuning framework for CUDA programs. This framework produces high-performance CUDA programs without requiring developers to have deep understanding of CUDA APIs and GPU architecture. In this section, we describe the specifications of this framework.

A. Improvement of Execution Configuration

To assign an appropriate Execution Configuration to a kernel function, it is important to consider the number of threads which are created in the kernel function. In this paper, we set the number of Block dimension and thread dimension in the Execution Configuration to one dimension, because how many dimensions in an Execution Configuration are set does not affect the performance of CUDA programs. Now, we consider how the performance of a kernel function is affected by the number of threads which are created in the kernel function and the Execution Configuration assigned to the kernel function. Here, we focus on how Blocks are allocated to SMs.

In the case when the number of threads created in a kernel function is small, the performance will depend on the Block size. Now, we explain this reason using Fig. 2. This figure



(a) When Block size is large (b) When Block size is small

Fig. 2. Examples of allocating Blocks to SMs

shows how Blocks are allocated to SMs, in the case when the number of threads created in a kernel function is small. For simplicity, we assume that the number of Blocks which can be allocated to each SM is at most two. In the case when a Block can contain many Warps, or Block size is large, a few Blocks are created, as shown in Fig. 2(a). In CUDA, as shown in this figure, Blocks are allocated as many as possible to only one SM. Then, the rest of the Blocks are allocated to other SMs. As a result, the Blocks will be allocated to only a few SMs, and the other SMs can be idle. Therefore, the total utilization of the SMs will be low. On the other hand, a Block can contain only a few Warps, or Block size is small, many Blocks are created, as shown in Fig. 2(b). As a result, Blocks are allocated to many SMs. Therefore, the total utilization of the SMs will be high, because few or no SMs will be in the idle state. Hence, in the case when the number of threads created in the kernel function is small, Blocks should be allocated to as many SMs as possible for an appropriate Execution Configuration.

On the other hand, in the case when the number of threads created in the kernel function is large, Blocks will be allocated to all SMs because the number of created Blocks will be enough large regardless of their Block size. In this case, the total utilization of the SMs can be high by utilizing each SM as efficient as possible, because no SM will be idle. In other words, in the case when massive threads are created in a kernel function, an Execution Configuration which can bring the highest Occupancy should be assigned to the kernel function.

Considering these factors, we propose a framework which automatically assigns an appropriate Execution Configuration to a kernel function. On this framework, in the case when the number of created threads is small, an Execution Configuration which can bring no idle SM is assigned to the kernel function. In order to assign such an Execution Configuration to the kernel function, the Block size with which Blocks will be allocated to all SMs is calculated. On the other hand, in the case the number of created threads is large, an Execution Configuration which can bring the highest Occupancy is assigned to the kernel function. In order to assign such an Execution Configuration to the kernel function, the Block size which can bring the highest Occupancy is determined by calculating Occupancy in various Block sizes, because Occupancy varies corresponding to the Block size. If several Block sizes can achieve the highest Occupancy, the Block size, which includes the most threads, is selected. After the block size is determined, the number of Blocks is calculated by dividing the number of threads, which are created in the kernel function, by the Block size. Thus, the number of Blocks and the Block size, both of which are parameters of an Execution Configuration, are determined.

However, the kernel functions whose Execution Configuration can be optimized have a restriction. For example, developers will generally implement template matching programs by allocating a thread to a pixel, and fitting Block size in the total number of pixels in the template image. In such an implementation, each Block compares the template image and a part of the target image, and the Block size has a close relation to the algorithm of the template matching program. Therefore, the program may output a wrong result if the Block size is changed. In such a program, the Execution Configuration should not be changed, because the meaning of the program will be also changed. It is difficult to automatically detect a kernel function whose Block size has close relation to the algorithm of the kernel function. Hence, a developer should specify each kernel function whose Execution Configuration should be optimized, when using this framework.

TABLE I
THE RULES FOR DECIDING WHICH MEMORY SHOULD BE USED

Shared Memory	Among data sets which will be modified, a data set which is the most frequently accessed is located in this memory.
Constant Memory	Among data sets which will not be modified, a data set which is the most frequently accessed is located in this memory.
ReadOnly Cache	Data sets which are used in some kernel functions are located in this memory.
Global Memory	The other data are located in this memory.

B. Automatic Data Transfer

As mentioned in section II-A1, GPU has various memories and caches. Since these memories and caches have different features, a developer should consider the difference of them and store each data set to a suitable memory. To conceal the complicated memory configuration of GPU from developers, the proposed framework can generate codes for automatically transferring each data set to a suitable memory. On the proposed framework, suitable APIs for transferring data to device memories can be inserted into the target program. On this framework, *Constant Memory*, *Shared Memory*, and *Read-Only Cache* are used for data allocation, because the access latencies of these memories are small. In this paper, we place a restriction that data which can be stored in these memories are only data sets in arrays, because which device memories are allocated to scalar variables does not affect much the performance of the kernel function. Now, we show the rules for deciding which memory should be used for storing each data set in TABLE I. These rules are defined for Kepler2 GPU[9]. For the high-performance kernel function, the utilization of Shared Memory and ReadOnly Cache should be maximized, because these memories can be accessed faster than Constant Memory. However, these memories have a restriction that data

on these memories are discarded when returning from a kernel function. Therefore, data sets which are used only in a kernel function can be located in these memories. Among such data sets, some data sets which will be modified can be located in Shared Memory, because Shared Memory is writable and readable. On the other hand, the other data sets which will not be modified can be located in ReadOnly Cache, because ReadOnly Cache is only readable. However, Shared Memory and ReadOnly Cache have small capacity. Hence, only a data set which is the most frequently accessed should be stored in them. In contrast, data on Constant Memory are kept until it is freed explicitly or the program execution is finished. Therefore, in the case when a data set is used across some kernel functions, once the data set is transferred to Constant Memory, the data set need not to be transferred multiple times. Specifically, such a data set should be transferred to Constant Memory, before the kernel function in which the data set are first used is executed. However, Constant Memory is only readable. Therefore, among data sets which are used in some kernel functions, data sets which will not be modified through the program execution should be located in this memory, and the other data should be located in Global Memory.

C. Description Format

To use our auto tuning framework, developers have to insert pragmas into a target program. The reason why we adopt a pragma is for high-extensibility and parsing simplicity. Developers should insert a pragma just before each kernel function call whose Execution Configuration should be optimized. The format of the pragma is as follows;

#pragma AUTO-CTA num_threads

AUTO-CTA is the keyword for using this framework, and *num_threads* is the parameter for indicating the total number of threads created in the kernel function. The parameter *num_threads* is indispensable for calculating the appropriate Execution Configuration. As mentioned in section II-B, this parameter is also necessary when using hiCUDA compiler and Bones compiler.

IV. Preprocessor for Auto Tuning

We implemented the auto tuning framework as a preprocessor. In this section, we will explain the design of the preprocessor.

A. Processing Flow

The processing flow of the preprocessor consists of four steps; parsing step, API insertion step, parameter collection step, and translation step. In the first step, the preprocessor parses a target program, and analyzes Host code and each kernel function. Meanwhile, it determines whether each data set should be transferred to device memories, then it selects a suitable memory in which the data set should be stored. In the second step, the preprocessor inserts some API calls into the target program for transferring the data to device memories. In the third step, the preprocessor collects parameters which are necessary to determine the appropriate Execution Configuration. In the last step, it generates the code for assigning the appropriate Execution Configuration to each kernel function which is specified by the pragma. In the following section, we will describe each step in detail.

B. Parsing Step

In this step, the preprocessor parses the target program, and analyzes Host code and each kernel function. The preprocessor manages analyzed kernel functions with a table called *kernel table*. The preprocessor also uses tables called *argument tables* for managing the arguments of kernel functions. One argument table is prepared for every kernel function.

Scanning Host code, the preprocessor detects and analyzes kernel function calls. When the preprocessor detects a function call with an Execution Configuration specified, it can recognize the function as a kernel function. Meanwhile, the set of the function name and its argument variables is registered into kernel-table. If the pragma described in section III-C is inserted just before the kernel function, the variable in the pragma for indicating the total number of threads is also registered into kernel-table. For determining which memory should be used for storing a data set of each formal argument, the preprocessor begins to analyze each kernel function body after scanning Host code.

First, the argument variables and their types are registered into argument-table. Then, the preprocessor counts how many times each argument variable is read in the kernel function body, and registers the result into argument-table, for giving priority to each formal argument. The reason for giving priority to each formal argument is that not all data sets can be stored in only one memory due to the memory capacity. While counting read accesses on each argument variable, the preprocessor also confirms whether the argument variable is overwritten or not, and registers the result into argument-table. This result is used for determining whether ReadOnly Cache or Constant Memory can be used for storing the data set in the argument variable. Therefore, for efficiently utilizing device memories, the data set of the formal argument which is the most frequently accessed is preferentially stored in the memory whose access latency is small. Finally, the preprocessor determines which memory should be used for storing data in each formal argument based on the rule as shown in TABLE I.

C. API insertion Step

In this step, the preprocessor inserts API calls into Host code for transferring the data, which should be stored in device memories, to a suitable device memory. Then, the preprocessor replaces memory read/write statements, which developers use in kernel functions for reading data from device memories and writing data into device memories, with read/write API calls which are suitable for device memories. In the following, we will explain how the preprocessor inserts the API calls and replaces memory read/write statements in detail.

First, the preprocessor scans Host code in order to determine where the API calls should be inserted. At this time, the preprocessor records the point where a datum in each scalar variable and a data set in each array variable, which are registered in kernel-table, will be overwritten. If a datum or a data set is overwritten multiple times, the latest point before the datum or the data set is passed to the kernel function is selected. Moreover, the preprocessor records variable names, if the data in the scalar variables or the data sets in the array variables will be referred in Host code after executing the kernel function. This is because the API calls for transferring such data to the main memory should be inserted into the target program. Next, the preprocessor inserts the suitable API calls for device memories at the recorded points. Specifically, the preprocessor inserts `cudamalloc()` for device memory allocation and `cudaMemcpy()` for data transfer to device memories. Then, the preprocessor inserts `cudaMemcpy()` for data transfer to main memory at the points after the kernel function call which takes the data as arguments. The API calls are for writing back the data in scalar variables or the data sets in array variables which are recorded by scanning the Host code.

Finally, the preprocessor replaces memory read/write statements in the kernel function body with suitable API calls for reading data from device memories and writing data into device memories. This is because the APIs are different according to device memories as mentioned in section II-A2.

D. Parameter Collection Step

In this step, the preprocessor collects various parameters which are necessary to calculate an Execution Configuration. The parameters are as follows;

- the number of registers used in each kernel function body
- the amount of Shared Memory used in each kernel function body
- the maximum number of the Warps which can be allocated to each SM
- the amount of Shared Memory which each SM has
- GPU generation
- the number of registers which each SM has
- allocation unit size of a register
- allocation unit size of Shared Memory
- the number of Warps which are simultaneously allocated

In the following, we will explain how the preprocessor collects these parameters in detail.

First, the preprocessor compiles the target program for collecting some parameters. The parameters are the number of registers and the amount of Shared Memory used in each kernel function body.

Next, the preprocessor produces a program which includes an API call `cudaGetDeviceProp()`, and collects other parameters by executing the program. The parameters are the maximum number of Warps which can be allocated to each SM, the amount of Shared Memory which each SM has, and GPU generation.

```
1  void calc_Occupancy(int num_T, int *cta){
2    if(num_T < Threshold_T){
3      for(Warps=1; SMs <= SMs_max; Warps++){
4        /* Calculating the Block size with which Blocks
5           are allocated to as many SMs as possible */
6      }
7    }
8    else{
9      for(Warps=1; Warps*32 <= MAX_B; Warps++){
10       /* Calculating the Block size
11          for achieving the highest Occupancy. */
12     }
13   }
14   calculating the number of Blocks;
15   /* store the number of Blocks and the Block size
16      into the array cta.*/
17 }
```

Fig. 3. The function for calculating Occupancy

```
1 threads = x * y;
2 #pragma AUTO−CTA threads
3 kernel_func<<<Blocks, BlockSize>>>(var1, var2);
```

(a) Before replaced

```
1 threads = x * y;
2 int cta[2];
3 calc_Occupancy(threads, cta);
4 kernel_func<<<cta[0], cta[1]>>>(var1, var2);
```

(b) After replaced

Fig. 4. Examples of translating the code

Finally, the preprocessor searches the prepared database for the other parameters by using the GPU generation as a key. This is because the parameters are different according to GPU generations and can not be collected by calling some CUDA APIs.

E. Translation Step

In this step, the preprocessor generates the code for assigning the appropriate Execution Configuration to each kernel function. For assigning it, we prepare a function named `calc_Occupancy` which calculates the appropriate Execution Configuration based on the Occupancy. This function takes the number of threads created in the kernel function as its argument, and the pointer to an array variable which is used for returning the appropriate Execution Configuration(the number of Blocks and Block size). In the following, we will explain this function in detail.

Now, we show a pseudo code of the definition of `calc_Occupancy` in Fig. 3. First, `calc_Occupancy` determines whether the number of threads created in the kernel function is less than the threshold or not (line 2), because how to determine the Execution Configuration is different from the number of threads created in the kernel function in our proposed framework. If the number of threads which are created in the kernel function is small, `calc_Occupancy` determines the Block size with which Blocks will be allocated to as many SMs as possible (line from 3 to 6). On the other hand, if the number of threads which are created in the kernel function is large, `calc_Occupancy` determines the Block size with which SMs will be utilized maximum (line from 9

TABLE II
EVALUATION ENVIRONMENT

GPU	1GTX TITAN
GPU Generation	Kepler2
Core Frequency	876 MHz
Memory Frequency	3004 MHz
Number of SMs	14 Cores
CPU	Core i7-4770
Frequency	3.40 GHz

to 12). That is, this function determines the Block size for achieving the highest Occupancy. Then, the number of Blocks is calculated by dividing the number of threads passed as the argument by the calculated Block size (line 14).

Now, let us see how the preprocessor translates codes for assigning the appropriate Execution Configuration to each kernel function. Fig. 4:(a) and Fig. 4:(b) show a part of Host code before and after the translation respectively. First, the preprocessor replaces the pragma ((a):line 2) with a declaration of an array for storing the appropriate Execution Configuration and with a function call ((b):line 2,3). Then, the preprocessor assigns the appropriate Execution Configuration to the kernel function ((b):line 4).

V. EVALUATION

We have evaluated the proposed preprocessor by comparing execution time of kernel functions whose Execution Configuration is fixed and that of kernel functions tuned by the preprocessor. This section describes the evaluation result.

A. Evaluation Environment

The evaluation environment is shown in the TABLE II. We used Kepler2 GPU for evaluation. Workloads are four benchmark programs in Rodinia Benchmark suite[10]. Specifically, we used cfd, gaussian srad, bfs, and kmeans, which are compiled with CUDA version 5.5.0 with '-O3'. The threshold value which is used in `calc_Occupancy` is defined as 1024 in this evaluation.

B. Evaluation Result

Fig. 5 shows the results of the execution time of each kernel function in the benchmark programs. We have evaluated following 6 programs.

- (R_{64}) The program in which Block size of each Execution Configuration is fixed to 64.
- (R_{128}) The program in which Block size of each Execution Configuration is fixed to 128.
- (R_{256}) The program in which Block size of each Execution Configuration is fixed to 256.
- (R_{512}) The program in which Block size of each Execution Configuration is fixed to 512.
- (P1) The program in which the optimized Execution Configuration calculated by the preprocessor is used.
- (P2) In addition to (P1), data allocation is automated by the preprocessor.

Each bar is normalized to the execution time with the Execution Configuration which is originally used in the benchmark programs. Among the benchmark programs, kmeans can not run when Block size is fixed to 128 or 512. Therefore, we leave out the execution results of kmeans with the Block size 128 and 512 in Fig. 5.

First, the execution time of each kernel function in (P1) is close to the shortest execution time among (R_{64}), (R_{128}), (R_{256}), and (R_{512}), with the exception of six kernel functions; cuda_compute_flux, cuda_time_step, cuda_compute_step_factor, cuda_initialize_variables, Kernel1, and Kernel2. Though, the exceptions perform no worse than with the original Execution Configuration, as shown in Fig. 5. As shown with cuda_compute_flux, cuda_time_step, cuda_compute_step_factor, Kernel1, and Kernel2, the execution time of the kernel functions can drastically be prolonged, when developers assign an inappropriate Execution Configuration to kernel functions. Even if a developer unwittingly assigned an inappropriate Execution Configuration to such the kernel functions, the preprocessor can optimize the Execution Configuration and the performance deterioration can be avoided.

Next, the execution time of each kernel function in (P2) is shorter than the execution time in (P1) with cuda_compute_flux, prepare, Kernel1, and invert_mapping. In particular, the execution time of cuda_compute_flux and invert_mapping are much reduced. To examine this reason, we counted how many times each data set, which is located in device memories except Global Memory, is accessed. As a result, we found that such the data sets are frequently accessed in these kernel functions because a part of each data set is also used in a loop. Therefore, the execution time is reduced by locating such data in a memory whose access latency is small. However, the execution time of cuda_compute_step_factor, srad, compress, and kmeansPoint are prolonged a little. This is because the data, which are used in these kernel functions and located in device memories except Global Memory, are not frequently accessed. As a result, the overhead for locating such data in a memory with small access latency is larger than the amount of the execution time which can be reduced.

In conclusion, the proposed preprocessor generally can assign an appropriate Execution Configuration to each kernel function, and locate data in suitable device memories. The execution time of each kernel function in the benchmark programs can be reduced by 7.78% in maximum and by 1.67% in average by optimizing its Execution Configuration with the proposed preprocessor. The execution time of each kernel function in the benchmark programs can be reduced by 20.4% in maximum and by 9.16% in average by automating data allocation along with the Execution Configuration optimization.

VI. CONCLUSIONS

GPU has high theoretical performance. However, to exploit the GPU performance, developers need to have deep understanding of the GPU architecture and CUDA APIs. In this paper, we proposed an auto tuning framework for GPU programs, and we implemented a preprocessor for the framework. This preprocessor can automatically inserts the code for assigning an appropriate execution configuration to each kernel function

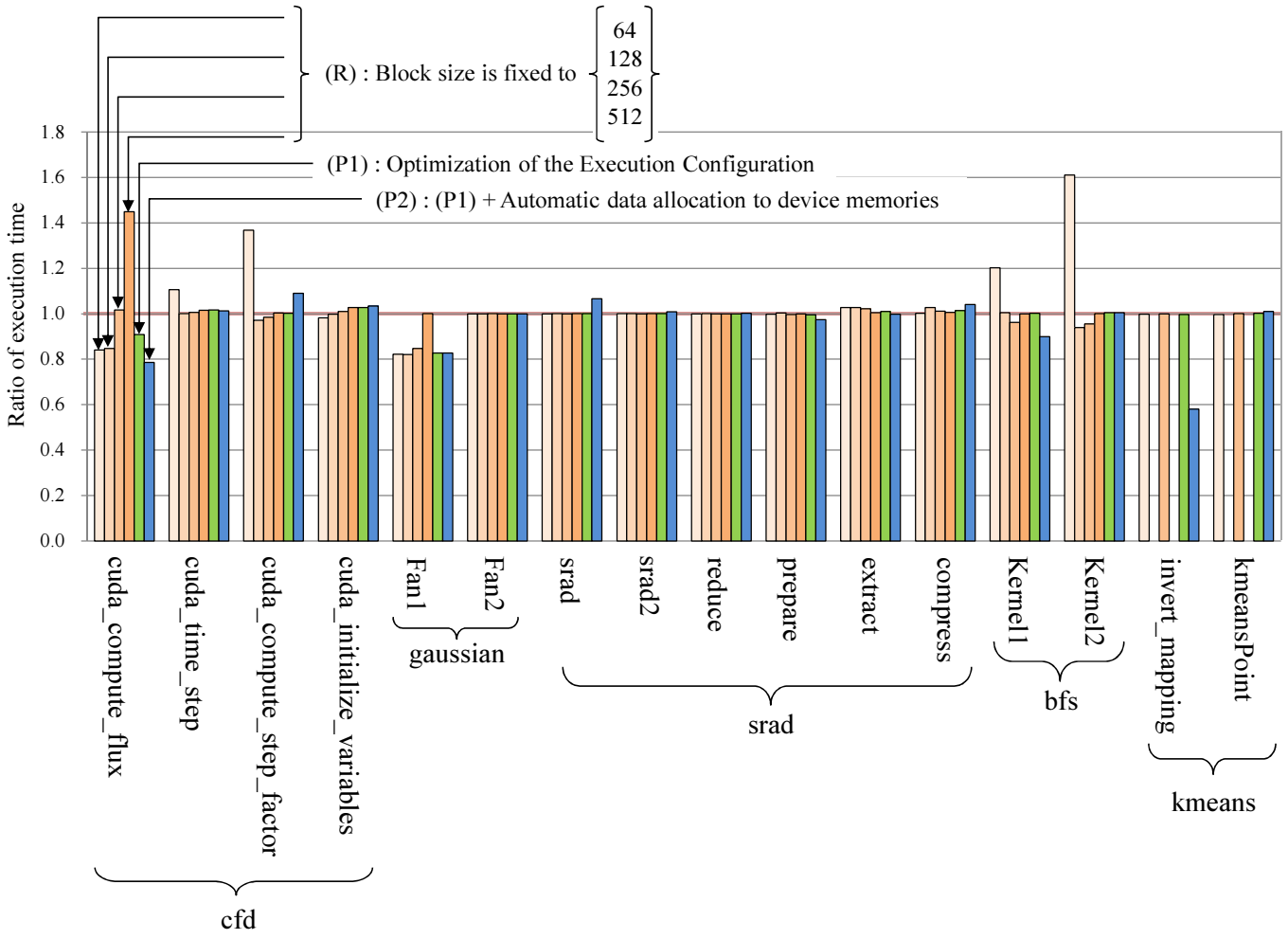


Fig. 5. The evaluation result with Kepler2 GPU

specified by a developer. By using the preprocessor, developers can generate high-performance CUDA programs.

One of our future works is to extend the preprocessor. For example, a compiler implemented by Yang [11] translates CUDA programs for improving the efficiency of memory accesses. We will extend the preprocessor for drawing the higher performance of CUDA programs than that of CUDA programs tuned by the preprocessor by improving the efficiency of memory accesses like the compiler.

REFERENCES

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

[2] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA Code Generation for Affine Programs," in *Proc. 19th Int'l Conf. on Compiler Construction (CC)*, Mar. 2010, pp. 244–263.

[3] M. Wolfe, "Implementing the PGI Accelerator Model," in *Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU'10)*. ACM, 2010, pp. 43–50.

[4] J. Enmyren and C. W. Kessler, "SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems," in *Proc. of 4th Int'l Workshop on High-level Parallel Programming and Applications (HLPP'10)*. ACM, 2010, pp. 5–14.

[5] T. D. Han and T. S. Abdelrahman, "hiCUDA: High-Level GPGPU Programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 78–90, Jan. 2011.

[6] ——, "hiCUDA: A High-level Directive-based Language for GPU Programming," in *Proc. of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2)*. ACM, 2009, pp. 52–61.

[7] C. Nugteren and H. Corporaal, "Introducing 'Bones': A Parallelizing Source-to-source Compiler Based on Algorithmic Skeletons," in *Proc. 5th Annual Workshop on General Purpose Processing with Graphics Processing Units (GPGPU-5)*, March 2012, pp. 1–10.

[8] N. Ventroux, T. Sassolas, A. Guerre, B. Creusillet, and R. Keryell, "SESAM/Par4All: A Tool for Joint Exploration of MPSoC Architectures and Dynamic Dataflow Code Generation," in *Proc. 12th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO'12)*, Jan. 2012, pp. 9–16.

[9] NVIDIA Corp., "NVIDIA'S Next Generation CUDA Compute Architecture: Kepler GK110," Whitepaper, 2012.

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

[11] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU Compiler for Memory Optimization and Parallelism Management," in *Proc. 10th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2010, pp. 86–97.