**TOHOKU UNIVERSITY**
Graduate School of Information Sciences

# Automatic Performance Tuning Methods for Heterogeneous Computing Systems

**(複合型計算システムにおける性能自動チューニング手法に関する研究)**

A dissertation submitted for the degree of
Doctor of Philosophy (Information Sciences)

Department of Computer and Mathematical Sciences

by

**Katsuto SATO**

January 16, 2012

Automatic Performance Tuning Methods for Heterogeneous Computing Systems

Katsuto Sato

Abstract

Conventional computing systems, called *homogeneous computing systems*, consist of general purpose processors (CPUs) with the same hardware architecture. Currently, the size of a homogeneous computing system is often limited by the power budget available for the system. Since drastic improvement of power efficiencies is practically hard for CPUs, it is difficult for such a system to significantly increase the system size under assumption of the same power budget. Therefore, the power budget restricts the performance improvement, and it becomes a big program.

*Heterogeneous computing systems* consist of some different processors that have different architectures, such as general purpose processors (CPUs) and accelerators. A heterogeneous computing system is considered a promising system architecture to achieve high-performance and energy-efficient computing. As accelerators have high floating operation rates and high memory bandwidths, heterogeneous computing systems can achieve high power efficiency. Therefore, heterogeneous computing systems are widely utilized in recent large-scale high performance computing systems. TSUBAME 2.0 [1] and Roadrunner [2] are well-known as examples of large-scale heterogeneous computing systems. As these systems have many accelerators, a programmer can use multiple accelerators to improve a performance of a program. In these systems, appropriate task assignment is important to efficiently use accelerators of the systems: processor selection and load balancing.

As the sustained performance on an accelerator depends on computations of tasks, programmers need to assign only suitable tasks to accelerators. Assigning an unsuitable task causes serious performance degradation. Moreover, appropriate task assignment is determined at runtime because it often depends on input data and available accelerators. Hence, runtime processor selection is important to exploit the high computing potential of heterogeneous computing systems.

Load-balancing among multiple accelerators is required to efficiently use a heterogeneous computing system that has many accelerators. Basically, programmers must assign tasks to a processor with careful consideration of suitability between a task and a processor. However, since appropriate task assignment completely depends on the configuration of accelerators in a heterogeneous computing system, it is difficult to assign tasks into accelerators in advance. Inappropriate task assignment may cause serious performance degradation in the system.

A Graphics Processing Unit (GPU) is one of accelerators that have high floating-point operation rates and high memory bandwidths. As a standard PC with a GPU can be seen as a widely-available heterogeneous computing system, standard programmers can use a GPU for computation, which is called *GPU computing*. For GPU computing, programmers can write a *kernel* function to define *a task* that is executed on GPUs by using programming frameworks such as CUDA [3] or OpenCL [4].

As accelerators such as GPUs have architecture-specific features useful to improve performance, architecture-specific performance tunings and optimizations such as execution parameter setting and efficient memory accesses are necessary to achieve high performance. However, these tunings and optimizations are difficult and labor-intensive even for expert programmers. Thus, the automatic tunings and optimizations are required to improve sustained performance of accelerators.

To alleviate these difficulties, programming frameworks or system software should be equipped with the functionality of the performance tunings. To this end, automation of performance tunings

is strongly required. In this dissertation, automatic performance tuning methods including runtime processor selection, program optimization, and load balancing among accelerators are proposed to overcome the above difficulties.

In Chapter 2, the SPRAT framework consisting of a domain-specific programming language and its runtime environment is proposed to automatically select the appropriate processor at runtime. The SPRAT compiler translates a program written in the SPRAT language to a code written in C++ for a CPU and a code written in CUDA for GPUs, respectively. The SPRAT runtime environment automatically selects an appropriate processor for each task based on the performance prediction. In the performance prediction of SPRAT, a linear prediction model is built for each program. In addition, runtime processor selection based on energy efficiency is performed by using the prediction model and the parameters of power consumption. The effects of runtime processor selection for performance-aware and energy-aware computing are evaluated. From the evaluation results, it is demonstrated that the SPRAT framework enables even a non-expert programmer to benefit from the use of GPUs without risks of performance degradation.

In Chapter 3, to improve the sustained performance, two performance tuning strategies are proposed; some optimization methods to improve sustained memory bandwidth and a tuning method of execution parameters. In GPU computing, improving sustained memory bandwidth is important to achieve high performance. Hence, Chapter 3 first proposes one tuning strategy including two optimization methods of memory accesses. *Reusable data prefetching* finds highly-reusable data blocks and places those blocks on an on-chip memory to shorten the memory access latency. *Adjusting access patterns* removes the inefficient memory access patterns in the program by using an on-chip memory as a read buffer. Furthermore, Chapter 3 proposes the other tuning strategy to automatically find the optimal configuration of execution parameters based on profiling data. The proposed tuning method runs a program with several configurations and measures its sustained performances for profiling. Evaluation results indicate that the two performance tuning strategies are effective to improve sustained performance of CUDA programs.

In Chapter 4, an online task scheduling method is proposed to realize automatic load balancing among multiple accelerators. Automatic load balancing by online task scheduling is effective to extract the potential of multiple accelerators. For the task scheduling, accurate performance prediction and dependency analysis are necessary. Hence, Chapter 4 first proposes a performance prediction method and a dependency analysis method. Then, an online scheduling method is proposed. The proposed performance prediction method uses not only past execution times but also the argument values passed to the tasks. In the dependency analysis, unnecessary data dependencies and synchronization points are removed to make many more parallel tasks. The online task scheduling method is performed based on the *minimum completion time* (MCT) algorithm. In this method, a task in a program is assigned to an appropriate accelerator that can early complete the task. Evaluation results show that the proposed prediction method achieves higher accuracy than conventional methods. The online task scheduling method can automatically and finely balance the loads between different accelerators, and efficiently use their performances.

In conclusion, this dissertation establishes three approaches that enable programmers to fully exploit the computing potential of heterogeneous computing systems. The proposed methods can automate performance tuning and optimizations, and realize a programming framework with automatic performance tuning mechanisms. This programing framework enables programmers to describe programs without labor-intensive performance tunings and risks of performance degradation.

# Acknowledgments

I have obtained a lot of supports and warm leading from many people of Tohoku University. I would like to gratefully acknowledge all of them.

First of all, I would like to express sincere gratitude to my research supervisor, Associate Professor Hiroyuki Takizawa who has conducted me to this research field and has given abundant leading. I would also like to express cordial acknowledgment to Professor Hiroaki Kobayashi who has supported and offered invaluable assistance. They have given me great help and have encouraged me in my research activity. I would like to thank Professor Kazuhiro Nakahashi and Professor Takafumi Aoki for their thoughtful review of my dissertation and valuable comments. I wish to express my appreciation to Assistant Professor Ryusuke Egawa for invaluable discussion and his considerable advices.

I would like to thank Associate Professor Hideaki Goto for his support and helpful comments. I wish to express special thanks to Dr. Kazuhiko Komatsu and Dr. Yoshitomo Murata who give me important assistance and invaluable suggestions in my research. I would also like to express my appreciation to my irreplaceable colleagues; Mr. Yusuke Funaya, Mr. Yoshiei Sato, and Mr. Masayuki Sato. We have encouraged each other and discussed many themes in various research topics. Special thanks go to the member of the heterogeneous computing project team, Mr. Kentaro Koyama, Mr. Yusuke Arai, and Mr. Makoto Sugawara for the meaningful discussions.

Finally, I express the deep acknowledgement to my family. I would like to thank my parents for their affectionate support. I would also like to thank my sister for warm encouragement.

January, 16, 2012
Sato, Katsuto

# Contents

**Abstract** **i**

**Acknowledgments** **iii**

**1 Introduction** **1**

1.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Graphics Processing Units as Accelerators . . . . . . . . . . . . . . . . . . 6

1.3 Objective of the Dissertation . . . . . . . . . . . . . . . . . . . . . . . 8

1.4 Organization of the Dissertation . . . . . . . . . . . . . . . . . . . . . . 10

**2 A Domain-specific Language with Runtime Processor Selection** **11**

2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.2.1 Programming Environments for Heterogeneous Computing Systems 14

2.2.2 Performance Model and Prediction . . . . . . . . . . . . . . . . . 16

2.3 A Domain-specific Language with Runtime Processor Selection . . . . . . 19

2.3.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

2.3.2 Stream Programming language with Runtime Auto-Tuning (SPRAT) 19

2.3.3 Performance Prediction and Processor Selection . . . . . . . . . . 23

2.4 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

2.4.1 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . 29

2.4.2 Evaluation of Performance-aware Processor Selection . . . . . . . 29

2.4.3 Evaluation of Energy-aware Processor Selection . . . . . . . . . . 31

2.5 Concluding Remarks . . . . . . . . . . . . . . . . . . . . . . . . . . 37

**3 Automatic Performance Tuning for the Domain-specific Language 41**
3.1 Introduction . . . 41
3.2 Related Work . . . 43
3.2.1 Performance Tuning in CUDA . . . 43
3.2.2 Optimization Tools for GPU computing . . . 46
3.3 Optimizing Methods Based on Architectural Features . . . 48
3.3.1 Optimization Methods for Memory Accesses . . . 48
3.3.2 Automatic Performance Tuning of the CTA configuration . . . 56
3.4 Evaluation . . . 63
3.4.1 Benchmarks for Evaluations . . . 63
3.4.2 Evaluation of Optimization Methods for Memory Accesses . . . 63
3.4.3 Evaluation of the CTA Configuration Tuning . . . 68
3.5 Concluding Remarks . . . 77

**4 Online Task Scheduling in Standard Programming Environments 79**
4.1 Introduction . . . 79
4.2 Related Work . . . 82
4.2.1 OpenCL . . . 82
4.2.2 Performance Prediction Methods . . . 84
4.2.3 Methods for Dependency Analysis . . . 86
4.2.4 Methods for Task Scheduling . . . 87
4.3 Online Task Scheduling Based on Performance Predictions . . . 90
4.3.1 History-based Performance Prediction with Profile Data Classification 90
4.3.2 Data and Event Dependency Analysis Methods . . . 95
4.3.3 Online Task Scheduling Based on the MCT Algorithm . . . 100
4.4 Evaluations . . . 103
4.4.1 Evaluation of the Performance Prediction . . . 103
4.4.2 Evaluation of the Performance Improvement by Online Task Scheduling . . . 109
4.5 Concluding Remarks . . . 117

**5 Conclusions 119**

**Bibliography 123**

# List of Tables

2.1 Specifications of GPUs used for evaluation. . . . . . . . . . . . . . . . . . . 29
2.2 Power consumption of each system configuration. . . . . . . . . . . . . . . . 31

3.1 The memory hierarchy in CUDA. . . . . . . . . . . . . . . . . . . . . . . . 44
3.2 The bandwidths ratios between coalesced and uncoalesced memory accesses for several GPUs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
3.3 Experimental setup in Section 3.4.2. . . . . . . . . . . . . . . . . . . . . . . 65
3.4 Experimental setup in Section 3.4.3. . . . . . . . . . . . . . . . . . . . . . . 69

4.1 The types of data dependencies among tasks. . . . . . . . . . . . . . . . . . 95
4.2 Experimental setup to evaluate the proposed prediction method. . . . . . . 103
4.3 Experimental setup to evaluate the proposed online task scheduling method. 109

# List of Figures

1.1 Peak double-precision floating-point operation rates of several processors. . 2
1.2 Peak memory bandwidths of several processors. . . . . . . . . . . . . . . . 2
1.3 The difference in usage of hardware resources for CPUs and accelerators. . 3
1.4 An example of heterogeneous computing systems. . . . . . . . . . . . . . . 4
1.5 Performance tunings in programming frameworks. . . . . . . . . . . . . . 9

2.1 The hardware architecture assumed in CUDA. . . . . . . . . . . . . . . . . 15
2.2 The compiling flow of a SPRAT program. . . . . . . . . . . . . . . . . . . 20
2.3 The coordinate in gather access. . . . . . . . . . . . . . . . . . . . . . . . 22
2.4 Speedup ratio of the CFD simulation. . . . . . . . . . . . . . . . . . . . . 30
2.5 Speedup ratio of the LU decomposition. . . . . . . . . . . . . . . . . . . . 33
2.6 Sustained performance of C2Q and GF88GTX. . . . . . . . . . . . . . . . 35
2.7 Energy efficiency of C2Q and GF88GTX. . . . . . . . . . . . . . . . . . . 35
2.8 Sustained performance of C2Q and GFGTX28. . . . . . . . . . . . . . . . 36
2.9 Energy efficiency of C2Q and GFGTX28. . . . . . . . . . . . . . . . . . . 36

3.1 The thread hierarchy in CUDA. . . . . . . . . . . . . . . . . . . . . . . . 43
3.2 Warp switching execution in CUDA. . . . . . . . . . . . . . . . . . . . . . 45
3.3 The memory access pattern of a `gather` stream by an absolute index. . . . 49
3.4 The memory access pattern of a `gather` stream by a relative index. . . . . 49
3.5 The memory access pattern of `in` and `out` streams. . . . . . . . . . . . . 49
3.6 Estimation for the histogram of memory access counts. . . . . . . . . . . . 51
3.7 Examples of a global memory access in CUDA. . . . . . . . . . . . . . . . 52
3.8 The overview of the proposed method to adjust the access pattern. . . . . . 53
3.9 Relationship between sustained bandwidths and CTA configurations on the copy kernel. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

3.10 Relationship between sustained computing performances and CTA configurations on the saxpy kernel. . . . . . . . . . . . . . . . . . . . . . . . . . 58
3.11 The method of building a prediction model by profiling. . . . . . . . . . . . 62
3.12 The policy of selecting the optimal CTA configuration. . . . . . . . . . . . 62
3.13 Evaluation results with the Himeno benchmark. . . . . . . . . . . . . . . . 66
3.14 Evaluation results with the LU decomposition. (GeForce 8800 GTX) . . . . 67
3.15 Evaluation results with the LU decomposition. (GeForce GTX 280) . . . . 68
3.16 The performance model of each CTA configuration on the Himeno benchmark. 70
3.17 The predicted execution time of each CTA configuration on the Himeno benchmark. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
3.18 The actual execution time of each CTA configuration on the Himeno benchmark. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
3.19 The ranking of actual execution time on the Himeno benchmark. . . . . . . 72
3.20 The correlation between the predicted and the actual rankings of CTA configurations on the Himeno benchmark. . . . . . . . . . . . . . . . . . . . 72
3.21 The performance model of each CTA configuration on the LU decomposition. 73
3.22 The predicted execution time of each CTA configuration on the LU decomposition. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
3.23 The actual execution time of each CTA configuration on the LU decomposition. 74
3.24 The ranking of actual execution time on the LU decomposition. . . . . . . . 75
3.25 The correlation between the predicted and the actual rankings of CTA configurations on the LU decomposition. . . . . . . . . . . . . . . . . . . . 75

4.1 The problem of (a) appropriate task assignment and (b) the proposed solution. 80
4.2 The platform model of OpenCL. . . . . . . . . . . . . . . . . . . . . . . . 82
4.3 The thread hierarchy of OpenCL. . . . . . . . . . . . . . . . . . . . . . . 83
4.4 The queuing model of OpenCL. . . . . . . . . . . . . . . . . . . . . . . . 84
4.5 Prediction procedure of the proposed method. . . . . . . . . . . . . . . . . 91
4.6 Examples of data dependencies among tasks. . . . . . . . . . . . . . . . . 96
4.7 Examples of (a) a memory access graph and (b) a data dependency graph for matrix multiplication. . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
4.8 Examples of event dependency graphs for matrix multiplication. (a) The graph of unoptimized program. (b) The graph of optimized program. . . . 100
4.9 The conventional and proposed execution models in OpenCL. . . . . . . . 101

4.10 The procedure of the proposed online scheduling method based on the MCT algorithm. . . . 102
4.11 Usage of runtime parameters. . . . 104
4.12 Percentage of kernels within tolerable ARPE. . . . 105
4.13 Comparison results of ARPE in the cases where the proposed method outperforms the others. . . . 107
4.14 Comparison results of ARPE in the cases where the accuracy of the proposed method is less than those of the others. . . . 108
4.15 The evaluation results of the MonteCarloAsian. . . . 111
4.16 The operating rates of GPUs and a scheduling thread. (a) the *Single* implementation. (b) the *Automatic* implementation. . . . 112
4.17 Evaluation results of the POC and BCM benchmarks. . . . 114
4.18 Task assignment in the POC benchmark. (a) The case of balancing loads. (b) The case of using only a GPU. . . . 115

# List of Algorithms

1 Pseudo code of automatic tuning. . . . . . . . . . . . . . . . . . . . . . . . . . 60

# Listings

2.1 A sample code of a reference stream. . . . . . . . . . . . . . . . . . . . . . 21
2.2 An equivalent code for the kernel code. . . . . . . . . . . . . . . . . . . . . . 21
2.3 A sample code of `saxpy` written in the SPRAT language. . . . . . . . . . 23
2.4 The CFD program written in the SPRAT language. . . . . . . . . . . . . . . 38
2.5 The LU decomposition written in the SPRAT language. . . . . . . . . . . . 40
3.1 The sample code using an absolute indexing operator. . . . . . . . . . . . . 50
3.2 The sample code using relative indexing operators. . . . . . . . . . . . . . . 50
3.3 The code of the saxpy kernel. . . . . . . . . . . . . . . . . . . . . . . . . . . 57
3.4 The main kernel of the Himeno benchmark implemented by SPRAT. . . . . 64
3.5 The main kernel of the LU decomposition implemented by SPRAT. . . . . 64

# Chapter 1

# Introduction

## 1.1 Background

A conventional computing system has generally been *homogeneous* and consisted of general-purpose processors, called CPUs, with the same architecture. However, it is difficult for CPUs to drastically improve their power efficiencies. As a result, the size of such a system is limited by the given power budget. Although the most popular way to improve the system performance is to increase the system size, it is difficult to further increase the size of such a system without increasing the power consumption.

Figures 1.1 and 1.2 show the peak performances and the peak memory bandwidths of various processors. In these figures, accelerators such as *Graphics Processing Units(GPUs)* have higher performances and memory bandwidths than typical CPUs. This is because CPUs use a huge amount of hardware resource for large cache memories and complicated control logics to reduce the latency of instruction execution, while GPUs use most hardware resource for computation as shown in Figure 1.3. Although GPUs increase power consumption at a certain level, they usually improve the performance per watt for data-parallel processing due to their high peak performances and memory bandwidths, and thus improve the energy efficiency. Therefore, accelerators have become important components to achieve a higher performance under assumption of a limited power budget.

However, as accelerators are usually assumed to be controlled by CPUs, a computing system needs to have different kinds of processors, CPUs and accelerators. CPUs are used for I/O processing, communications, and complicated program flow controls. On the other hand, accelerators are used for massive data-parallel processing with high power efficiency.

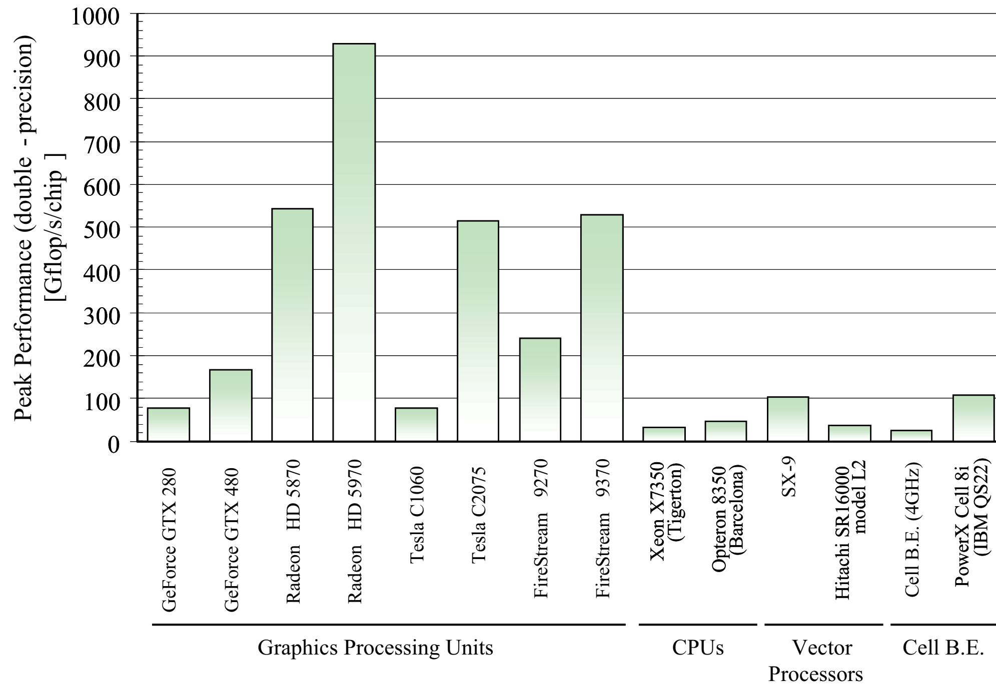



Figure 1.1: Peak double-precision floating-point operation rates of several processors.

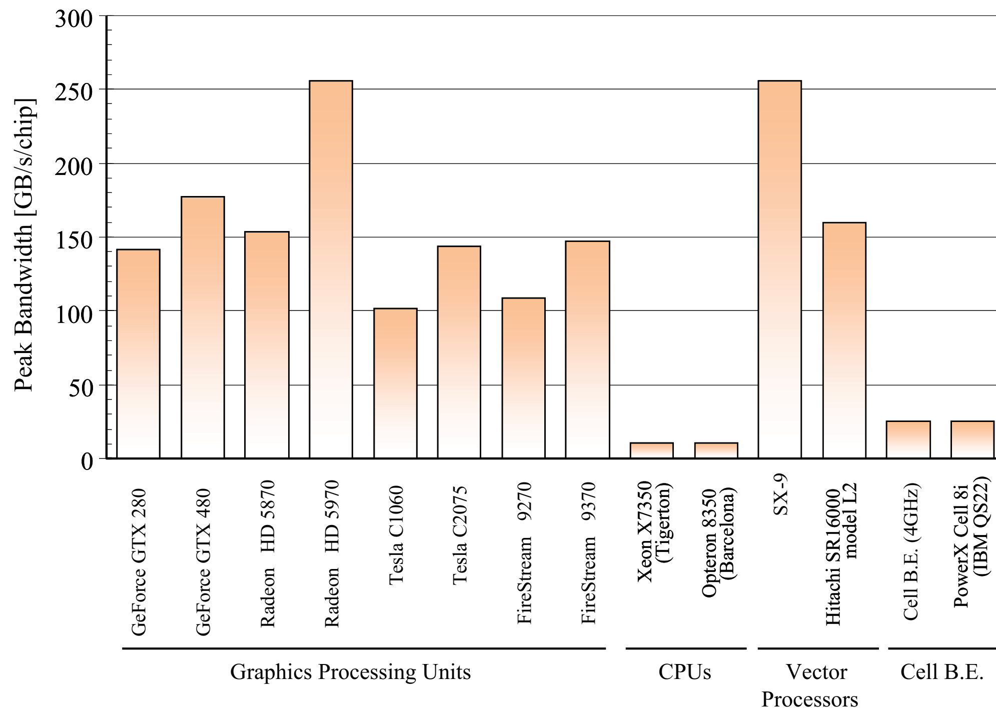



Figure 1.2: Peak memory bandwidths of several processors.

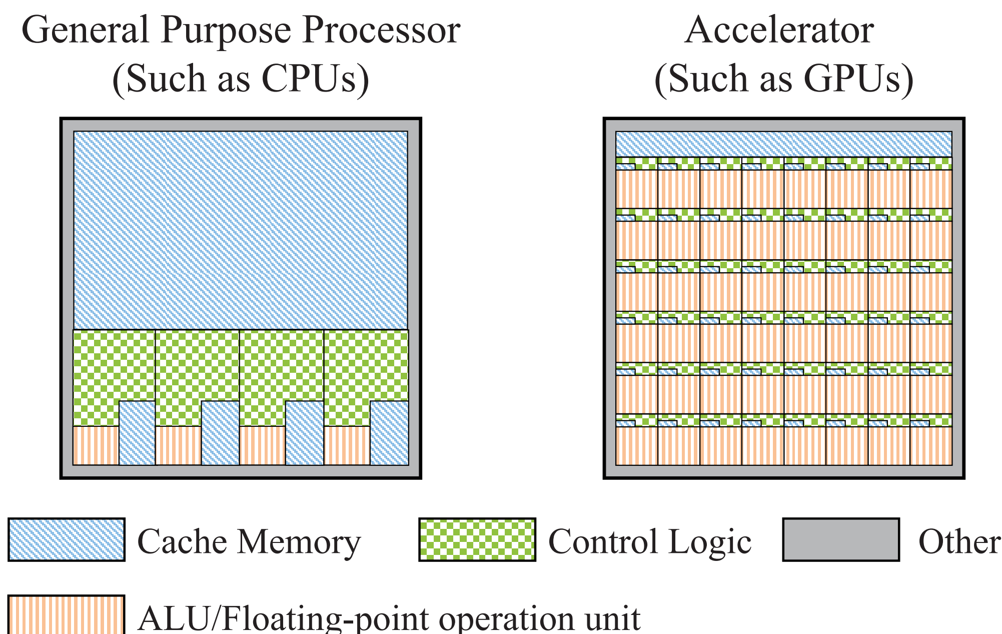



Figure 1.3: The difference in usage of hardware resources for CPUs and accelerators.

Hence, *heterogeneous computing systems* consisting of CPUs and accelerators are promising to achieve both high performance and high power efficiency, i.e. *high energy efficiency*.

In heterogeneous computing systems, to achieve both high performance and high power efficiency, each task must be appropriately assigned to one processor in the system. By the appropriate assignment of the tasks, the execution time of a computation is shortened, and hence the energy consumption is reduced; the energy efficiency is improved. However, inappropriate task assignment causes serious performance degradations. Hence, programmers have to appropriately assign tasks to processors with carefully considering the combination of a given computation and an available processor.

Figure 1.4 shows an example of heterogeneous computing systems. A heterogeneous computing system consists of multiple computing nodes, each of which has at least one CPU and one accelerator. Accelerators have their own memory spaces, called *device memories*, which are independent of main memory spaces managed by a CPU. Therefore, explicit data management is required for collaborative work of CPUs and accelerators. As the overhead of data transfer between a CPU and accelerators is not negligible in many cases, one difficulty in task assignment is to ensure that the performance gain is larger than the performance loss due to the overhead. Hence, it is required to consider the data transfer overhead for

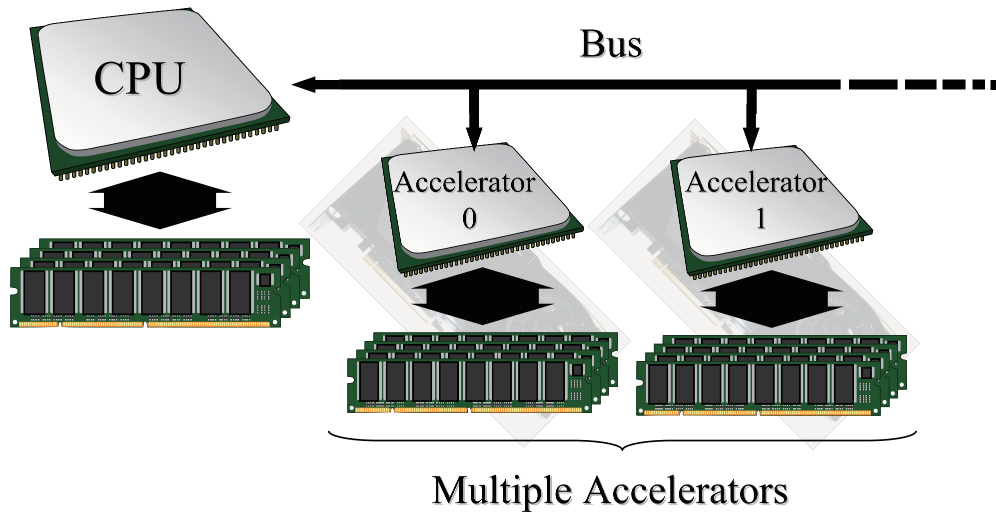



Figure 1.4: An example of heterogeneous computing systems.

appropriate task assignment.

If there are multiple accelerators in a heterogeneous computing system, another difficulty is to decide which accelerator executes a task. If an application has many *parallel tasks* that can be independently executed, load balancing among multiple accelerators is needed to minimize the execution time. As programmers do not always know available accelerators and the overhead of data transfers in advance of execution, dynamic load balancing among multiple accelerators is required to exploit their computing power. However, it is difficult and labor-intensive even for expert programmers to balance the loads among accelerators.

To fully exploit the system performance, it is needed to optimize a program according to the architecture of every accelerator in the system. For a conventional processor such as a CPU, a program has mainly been optimized to improve the ratio of cache hits and to parallelize for use of multiple cores. On the other hand, accelerators have execution parameters that affect the sustained performance, and some of the parameters must be specified at execution. As those parameters strongly depend on the accelerator to be used for execution, knowledge and programming experience of the accelerator are required for programmers to optimally determine the parameters. Hence, it is also difficult and labor-intensive to find the optimal parameter setting.

As described above, there are many difficulties to achieve high sustained performance because of complicated and labor-intensive performance tunings. Hence, programming

frameworks or system software must deal with some performance tunings to alleviate difficulties of performance tunings. To handle these performance tunings by programming frameworks, it is required to establish automatic performance tuning methods that carry out appropriate task assignment and tuning of execution parameters. Therefore, this dissertation discusses strategies to automate labor-intensive performance tunings.

## 1.2 Graphics Processing Units as Accelerators

Among several accelerators, GPUs draw attention as accessible accelerators that have high floating-point operation rates and high memory bandwidths. A GPU is originally designed to accelerate computation in CG rendering, called a *graphics computation*. Those computations can be offloaded to a GPU so as to alleviate loads on the CPU. The performance of GPUs has been improved to generate higher-quality images. Moreover, their programmability is also extended to support more advanced rendering techniques.

To improve the programmability, a *programmable shader* has been introduced to GPUs. A programmable shader enables a programmer to describe and execute a user-defined tiny program to implement more advanced effects of computer graphics. Programmers can access the programmable shaders via graphics *application programming interfaces* (APIs) and special shader languages such as Cg [5], HLSL [6], and GLSL [7]. With shader programming efforts to trick GPUs, even a non-graphics computation can exploit their computing power. These trials are called *general purpose computation on GPUs* (GPGPU).

In 2007, NVIDIA announced *Compute Unified Device Architecture* (CUDA)[3]. Unlike the previous shader programming methods, CUDA does not require graphics API to use GPUs for computation and is the first programming framework for *GPU computing*. CUDA provides an extended C language, called C for CUDA, and several algorithms can be freely implemented with a few constraints, which is generally so-called *programming flexibility*. Shader programming languages have strong constraints on the programming models, and programmers must describe programs with consideration for these constraints. However, CUDA eliminates most of the constraints and enables a programmer to develop a program in a similar fashion to the conventional C language. So far, there have been many reports to demonstrate that CUDA can accelerate various kinds of non-graphics applications [8]. In CUDA, a computation-intensive part of a program is offloaded to a GPU for acceleration, and the other is executed on a CPU. The offloadable part of a program is called a *CUDA kernel*. As a GPU works well for data-parallel processing, data-parallel processing parts of a program are usually rewritten as CUDA kernels.

Programmers can implement several algorithms for GPUs by using the CUDA language. However, to achieve a high performance, a programmer has to learn not only the CUDA language but also the architectural features of GPUs. The architectures of GPUs are drastically changed according to their generations and grades. It is strongly needed to optimize a program for a particular GPU architecture. Without architecture-aware optimizations,

the sustained performance would be very low. Moreover, as a program written in the CUDA language, called a *CUDA program*, can be directly executed only on NVIDIA's GPUs, it is impossible to switch one kind of processor to another of processor during the execution of a program.

In 2009, Khronos group proposed OpenCL that is a standard programming interface for various accelerators including GPUs [4]. A program written in OpenCL, called an *OpenCL program*, can be executed on different accelerators without any modification of the program. However, performance tunings such as optimizing task assignment and tuning of execution parameters are still required, and the sustained performance significantly depends on the performance tunings.

As mentioned above, CUDA and OpenCL enable programmers to describe a program that can work with GPUs. However, complicated and labor-intensive performance tunings are still required to effectively use GPUs. Especially, if the optimal tuning parameter depends on runtime factors, it is inherently impossible to know them in advance of execution.

## 1.3 Objective of the Dissertation

In programming heterogeneous computing systems, there are many difficulties in performance tunings. To overcome the difficulties, a programming framework should automate some performance tunings and enables programmers to write programs without consideration of processors that are used to execute tasks. Figure 1.5 shows the overview of the programming framework proposed in this dissertation to overcome the difficulties. As shown in this figure, automatic tuning mechanisms work in compiler and runtime environment layers, and then automation of performance tunings can free programmers from labor-intensive performance tunings. Hence, the objective of this dissertation is to establish automatic performance tuning methods used in the programming framework that allows to develop a program without considering the underlying processors in a system. To achieve this objective, the programming framework has to appropriately assign tasks to processors, determine optimal execution parameters for a particular processor, and appropriately balance loads among accelerators. Therefore, this dissertation proposes the following three methods to automate these performance tunings and clarifies the effects of these methods through quantitative evaluations.

One proposed method is a domain-specific programming language to automate appropriate processor selection. In this language, programmers define *kernels* in a program that can be executed either on a CPU or on an accelerator. In a programming framework, the proposed method works in both compiler and runtime environment layers, as shown in Figure 1.5. In the compiler layer, the proposed method abstracts differences of processors to realize runtime processor selection. In the runtime environment layer, the proposed method automatically selects an appropriate processor for each kernel based on performance prediction. Thereby, the proposed language enables programmers to write a program without considering which processor is selected to execute kernels. Moreover, this runtime environment can select an appropriate processor not only for performance-aware computing but also for energy-aware computing that can optimize power efficiency of heterogeneous computing systems.

Another proposed method is use the features of the proposed programming language to achieve some automatic performance tunings. This proposed method works in the compiler layer shown in Figure 1.5 of the programming framework and applies some optimizations and tuning for automatically-generated programs at compiling. As a result, programmers do not need to examine various optimization techniques to improve the sustained performance

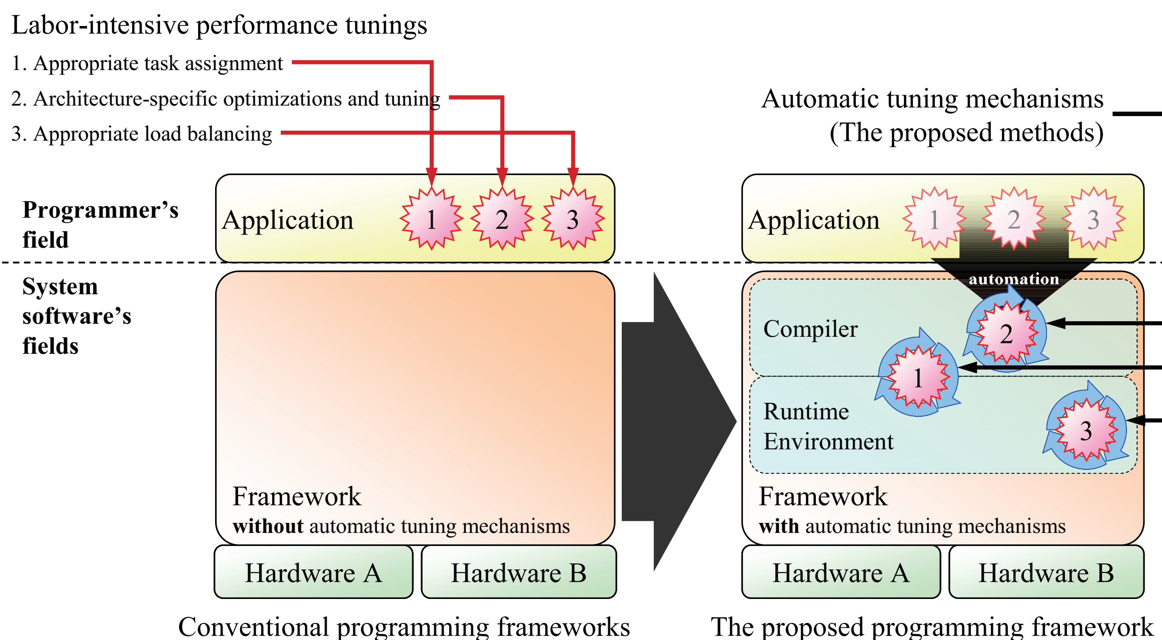



Figure 1.5: Performance tunings in programming frameworks.

of GPUs. This dissertation also achieves automatic tuning of execution parameters required for efficient use of GPUs.

The last proposed method is an online task scheduling method. If a heterogeneous computing system has multiple accelerators and if an application has the tasks that can be executed in parallel, called *parallel tasks*, load balancing is needed to simultaneously use those accelerators for parallel processing of the parallel tasks. For a parallel task whose execution time depends on its input data and varies dynamically, it is impossible to statically balance the loads among accelerators. Therefore, this dissertation proposes an online task scheduling method based on performance prediction to automate load balancing. The proposed scheduling method works in the runtime environment layer shown in Figure 1.5 to transparently realize automatic load balancing for an OpenCL program, and each parallel task in the OpenCL program is automatically assigned to an accelerator that can early complete the task. Then, this dissertation shows that the proposed scheduling method enables to efficiently use multiple accelerators.

## 1.4 Organization of the Dissertation

This dissertation is organized as follows.

Chapter 1 describes the background and the objective of this dissertation. In Chapter 1, the difficulties in programming for heterogeneous computing systems are pointed out: complicated and labor-intensive performance tunings are required to efficiently exploit accelerators of the system. Chapter 1 also describes the objective of this dissertation that is to establish automatic performance tuning methods used in a programming framework. The programming framework automatically performs the optimizations, tuning, and task assignment to alleviate the difficulties in programming heterogeneous computing systems. Thus, this dissertation proposes three useful methods to automate these performance tunings.

Chapter 2 proposes a domain-specific programming language for runtime processor selection. In Chapter 2, it is demonstrated that a processor is appropriately selected based on the performance prediction method in the following three chapters.

Chapter 3 describes automatic performance tuning methods for the domain-specific programming language proposed in Chapter 2. A program for a GPU must be sufficiently optimized for the architecture to fully exploit the computing capability of the accelerator. Hence, Chapter 3 proposes automatic optimizing methods to effectively use memory hierarchy in GPUs, and an execution parameter tuning method that determines the number of threads assigned to a processing core automatically.

Chapter 4 describes an online task scheduling method for a standard programming language, OpenCL, to automatically adjust loads among accelerators. Unlike the proposed domain-specific language discussed in Chapter 2, the latest standard programming languages such as OpenCL have high programming flexibility with few constraints. In these languages, there are difficulties in performance prediction and use of multiple accelerators. To automate load balancing among multiple accelerators, Chapter 4 proposes three methods: a highly-accurate performance prediction method, an analysis method to detect dependencies between parallel tasks, and an online task scheduling method.

Chapter 5 describes conclusions of this dissertation.

# Chapter 2

# A Domain-specific Language with Runtime Processor Selection

## 2.1 Introduction

To exploit the high computing performance of a heterogeneous computing system, programmers must appropriately use processors including CPUs and accelerators in the system. As accelerators such as GPUs especially have strengths and weaknesses about computation due to their application specific architectures, the sustained performance for a computation task strongly depends on the combination of the task and the available accelerator. Generally, accelerators work well only for tasks including massive data parallelism without complicated control flows. There is a case that the performance for a program is degraded by executing the program on accelerators. Hence, programmers must properly manage different kinds of processors to achieve a high performance.

The programming for heterogeneous computing systems of CPUs and GPUs, so-called *GPU programming*, is generally labor-intensive and error-prone. One of difficulties in GPU programming is how to determine which processor, either a CPU or a GPU, should be used for a given task. Even if a task is suitable for GPUs, finding an appropriate processor for the task is not easy. It is often difficult for a programmer and a compiler to ensure that a certain GPU in one PC can execute the task faster than its CPU. This is because the difference in sustained performance between a CPU and a GPU depends on individual tasks and information available only at runtime; such as the size of data processed by the task and the loop length in the task. As a result, the sustained performances of a CPU and a GPU drastically

change at runtime and are unpredictable even for expert programmers.

For energy-aware computing, GPUs can reduce the energy consumption of executing a program if GPUs considerably decrease the execution time. The energy efficiency also changes according to the sustained performance. Therefore, an appropriate processor selection mechanism is required to achieve high performance and/or energy-aware computing on a heterogeneous computing system.

CUDA [3] is currently the most popular programming language for GPU computing with NVIDIA GPUs. In CUDA, programmers define *tasks* as special functions, called CUDA kernels that are offloadable parts of a program to GPUs. A CUDA program needs at least one NVIDIA's GPU for execution, and other processors cannot execute a CUDA program. Hence, programmers must decide which processor executes a task and cannot change it at runtime. However, it is difficult to determine an appropriate processor for a task in advance of execution, because the appropriate processor may change at runtime. In some cases, it is needed to develop two programs of the same task for both a CPU and a GPU to select one of them at runtime, even though considerable efforts are needed to develop two versions for the task.

To alleviate burdens in programming, therefore, a programming language with runtime processor selection is useful. This language enables a programmer to describe a program without considering the processor to execute tasks because the program is automatically translated into a code for each processor. Then, each task is automatically assigned to an appropriate processor for performance-aware or energy-aware computing. Moreover, programmers can avoid unexpected performance degradation by inappropriate processor selection. Therefore, performance tuning by automatic processor selection is effective to alleviate one of difficulties in GPU programming.

In this chapter, a programming framework is proposed to realize runtime processor selection in a heterogeneous computing system. This framework consists of a domain-specific language and its runtime system. This chapter assumes a commodity personal computer (PC) as a heterogeneous computing system that has a CPU and a GPU. The design objective of the proposed programming language is as follows.

- To easily describe data-parallel processing that can be efficiently executed by accelerators such as GPUs.

- To easily predict an execution time for each accelerator.

- To select an appropriate processor based on performance prediction.

A lightweight runtime system is built to enable automatic processor selection considering the runtime information. Moreover, this chapter presents a metric to find the situation where one processor obviously outperforms another one. Then, this chapter shows that runtime processor selection can also improve energy efficiency.

## 2.2 Related Work

### 2.2.1 Programming Environments for Heterogeneous Computing Systems

One research topic in software development for heterogeneous computing systems is how to abstract different kinds of processors such as CPUs and GPUs. A GPU generally prefers massive SIMD data parallelism. Hence, a GPU is often used for *stream processing*, which is modeled as a computation-intensive *kernel* for processing a long *data stream*. The stream processing can hide its memory access latency by making memory accesses highly predictable and overlapping data fetches with computations. Importance of the overlapping is growing more and more, due to the so-called memory wall problem [9]. The stream processing is suitable not only for GPUs but also for many other processors such as general-purpose processors [10] and heterogeneous multicore processors [11]; it will be a key technology to achieve high sustained performance with current and future computing systems.

Many researchers have demonstrated that GPUs can be seen as general-purpose stream processors [12]. Several high-level programming languages have been proposed to alleviate the programming efforts required to use the computing power of GPUs for stream processing applications [13, 14, 15, 16, 17]. However, they do not consider runtime processor selection for executing a given task in terms of the sustained performance and the energy efficiency.

Brook for GPU (BrookGPU) [13] is the first abstraction of GPUs for GPGPU programmers and is a popular programming tool that extends the standard C programming language to explicitly describe stream processing applications. BrookGPU provides a high-level programming language and its runtime backends to facilitate the development of GPGPU applications. Each of the runtime backends corresponds to a runtime environment supported by BrookGPU: CPU, OpenGL, and DirectX9. Using the Brook language, a programmer can explicitly write a kernel code. The Brook compiler is a source-to-source compiler that translates a Brook code into a standard C++ code. At the translation, a kernel code is translated into multiple codes respectively corresponding to runtime backends. When the executable file is launched, it first checks the environmental variable, `BRT_RUNTIME`, to decide the runtime backend. Finally, all the kernels in an application program are executed using one of available processors called a *computing engine*.

There are also many programming tools that realize higher-level abstraction to facilitate GPU programming. RapidMind [14] and PeakStream [15] are both commercial software

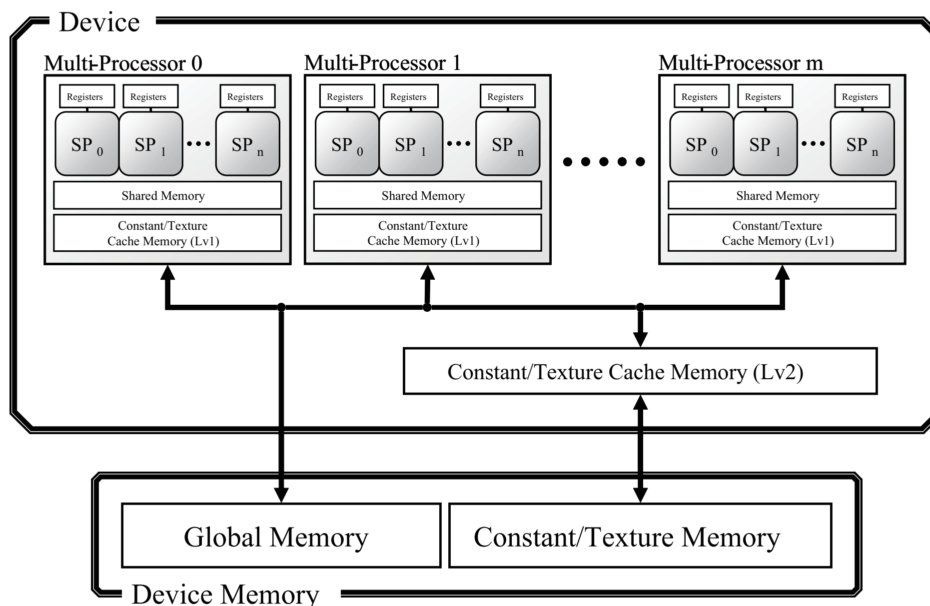



Figure 2.1: The hardware architecture assumed in CUDA.

products to describe and execute multi-platform programs in a stream processing manner. In 2008, PGI has announced a commercial C/C++ compiler that defines additional compiler directives for generating CUDA codes from a standard C/C++ code [18]. HMPP also provides compiler directives to reduce the difficulties in the GPU programming [19]. However, they usually assume that a programmer or an application user statically determine the computing engine of every kernel in advance of the execution, even though appropriate processor selection clearly depends on various runtime behaviors and system configurations.

CUDA abstracts underlying hardware, and a programmer can describe a single program for NVIDIA GPUs even if the hardware configuration such as the number of processors in a GPU is different. Figure 2.1 shows the overview of the hardware architecture assumed in CUDA. In the execution model of CUDA, many threads are organized as arrays of threads and executed in parallel. A GPU is called a *device*, and one device has several *Multi-Processors (MPs)* that independently execute an array of threads. One MP consists of multiple *Stream Processors (SPs)*, each of which executes one thread. SPs in a MP cooperatively execute threads in an array in a Single-Instruction Multiple-Data (SIMD) [20] manner. If the branch targets of threads in an array are different, all the SPs in a MP execute both the paths.

Although Brook, CUDA, and the above libraries provide a high-level abstraction of

GPU programming, they do not consider that the difference in sustained performance between a CPU and a GPU depends on the information available only at runtime, such as the data sizes and GPUs installed in the system. Therefore, appropriate processor selection clearly depends on various runtime behaviors and system configurations [21]. It is often difficult even for expert programmers to determine the appropriate processor for a computation, because a programmer does not always know the performance difference when programming. As a result of inappropriate processor selection, use of GPUs may lead to the performance degradation rather than acceleration.

### 2.2.2 Performance Model and Prediction

It is possible to improve performance by changing the processor to execute the kernel if the execution time of a kernel on one processor is obviously longer than that on another processor. Based on this idea, the kernel should be executed by the latter processor next time. Therefore, to select an appropriate processor, it is necessary to estimate the performances of two processors for each kernel.

There are some studies on performance evaluation and modeling of GPU computing applications. Since the GPU architectures are not fully disclosed, the GPU performance for various applications has been experimentally examined [22, 23, 24].

Govindaraju et al. investigated the details on a GPU memory hierarchy. Their memory model shows that the GPU's memory bandwidth depends on memory access patterns, and effective use of the two-dimensional cache memory can maximize the GPU performance [25]. Harrison et al. have assessed the execution time required for the data transfer between the main memory and the device memory [26]. As the time for the data transfer often becomes dominant especially in the case that a kernel has a low-arithmetic intensity, performance evaluation of the data transfer using different APIs is important to predict the total execution time of a GPGPU application. These experimental studies show that the performance prediction of GPGPU applications must consider styles of the implementations.

Ito et al. have proposed a model to estimate the execution time of a GPGPU application [27]. They assume that the GPU performance is always limited by its memory bandwidth, and hence the execution time of a kernel is in proportion to the size of data transferred between the GPU cores and the device memory. However, the actual memory bandwidth obviously depends on the memory access patterns. Therefore, He et al. have separately modeled sequential access performance and random access performance [28].

Buck et al. have presented BrookGPU with a simple performance model to analyze the performances of a CPU and a GPU [13]:

$$T_G = n(T_R + K_G), \tag{2.1}$$

$$T_C = nK_C, \tag{2.2}$$

where $T_G$ and $T_C$ are execution times of a GPU and a CPU, respectively, $T_R$ is the time associated with downloading and reading back a single stream element, $K_G$ and $K_C$ are the times required to execute a kernel on a single element, and $n$ is the number of elements in a data stream. It is obvious that the GPU will outperform the CPU only when

$$T_R < K_C - K_G. \tag{2.3}$$

This means, only if the performance gain by using a GPU exceeds the data transfer overhead, the GPU can outperform the CPU.

Transco et al. have reported a comprehensive experimental study of the performance comparison between a CPU and a GPU [29]. They investigated the execution times of BrookGPU kernels, changing kernel parameters such as the computation intensity, the data size and the data format. Their results clearly indicate that the GPU's superiority in performance depends on several parameters determined at runtime.

As the BrookGPU language can describe only stream kernels, it is relatively easy to model the performance of each kernel. However, it is difficult to automatically generate the performance model of arbitrary CUDA codes [30] because of its high programming flexibility. Therefore, one idea to achieve runtime processor selection is to limit the programming flexibility so that a runtime system can easily predict the performance. Since GPUs basically are suitable for stream processing, the flexibility-limited programming language can still describe many significant GPU applications even if it can describe only kernels of stream processing.

In addition to performance improvement, energy-aware computing also has become more and more important not only in mobile systems but also in *high-performance computing* (HPC) systems. Göddeke et al. have reported that use of even low-end and out of date GPUs leads to improvements in both performance- and power-related metrics of a GPU-accelerated cluster system for FEM applications [31]. Use of GPUs generally increases the power consumption, however, it does not always increase the performance. As a result,

it may lead to the increase in energy consumption for the applications, which cannot be efficiently executed by GPUs. Accordingly, runtime performance prediction is needed to achieve appropriate processor selection in terms of energy efficiency; GPUs should be used only if its energy consumption of executing a program is substantially small because of the performance gain by GPUs.

## 2.3 A Domain-specific Language with Runtime Processor Selection

### 2.3.1 Overview

In this section, a programming framework named *Stream Programming with Runtime Auto-Tuning* (SPRAT) [32] is proposed to realize runtime processor selection. This framework consists of a domain-specific language to describe data-parallel processing, called the *SPRAT language*, and the SPRAT runtime environment for dynamic selection of an appropriate processor. As with BrookGPU [13], the SPRAT compiler translates a SPRAT code into multiple codes, each of which is corresponding to one processor. The processor for executing each kernel is called a *computing engine* in SPRAT. According to the runtime behaviors that are not available for a programmer and a compiler, the SPRAT runtime environment dynamically switches the computing engine so as to minimize the execution time or the energy consumption; a GPU is used as a computing engine only if it can accelerate the kernel execution or reduce the energy consumption.

Without any preknowledge, it is difficult to automatically predict the execution time of an arbitrary code. To achieve runtime performance prediction of user-defined functions, hence, the SPRAT language provides special functions, named *kernel functions*, in which a programmer can write only stream processing kernels. As a kernel function is applied to every stream element, the execution time increases linearly with the number of stream elements. When a stream is given, therefore, the SPRAT runtime environment can estimate the execution time of each kernel function with a simple performance prediction model of linear approximation.

### 2.3.2 Stream Programming language with Runtime Auto-Tuning (SPRAT)

The SPRAT language is an extension of the standard C language incorporating some special keywords for description of stream processing tasks. In a stream processing task, a *stream* is a collection of data processed by a *kernel*. A stream is declared with the `stream` keyword and angle-bracket syntax. A kernel function, which operates on individual stream elements, is specified by the `kernel` qualifier. The syntax of the SPRAT language itself is similar to the conventional stream programming languages such as BrookGPU [13].

Figure 2.2 illustrates how a SPRAT code is converted into an executable file. A SPRAT

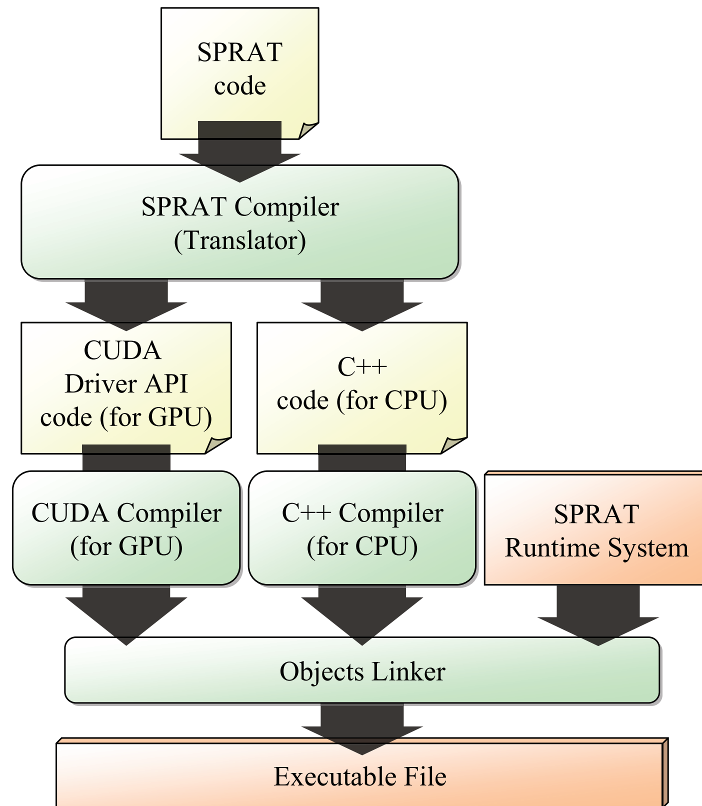



Figure 2.2: The compiling flow of a SPRAT program.

code written by a programmer is first translated into multiple codes: a standard C++ code for a CPU and a CUDA code for a GPU. Then, a programmer can manually optimize the automatically-generated codes if needed. Finally, those codes are respectively compiled and then linked with the SPRAT runtime library to generate an executable file.

The details on streams and kernels are described as follows.

### Streams

In the SPRAT language, a variable declared with the `stream` keyword is used as a container of stream data. The number and the variable type of stream elements are specified in the `stream` variable declaration.

Since stream elements are directly accessible only within a kernel function, there are built-in functions to manage stream data. For example, `streamRead` and `streamWrite` are used for the data transfer between a standard C array and a stream. The former copies

Listing 2.1: A sample code of a reference stream.

```
1 stream<float>& ref = strm[i][j](w,h);
```

Listing 2.2: An equivalent code for the kernel code.

```
1 for(int i=0; i<M; i++)
2    for(int j=0; j<N; j++)
3       z[i*N+j] = a*x[i*N+j]+y[i*N+j];
```

each array element to its corresponding stream element. The latter copies each stream element onto its corresponding array element.

There are four kinds of qualifiers to specify the access attributes of a stream: `in`, `out`, `inout`, and `gather`. A stream specified by the `in` keyword permits sequential read-only accesses. A stream with the `out` keyword permits sequential write-only accesses. A stream specified by `inout` is both readable and writable. An element in a `gather` stream can be read using an array index operator to be mentioned later.

In addition, a stream reference to a part of stream elements can be declared as shown in Listing 2.1. Here, a stream reference `ref` represents the domain of a 2-dimensional stream `strm` whose left-upper corner position and size are specified by `[i][j]` and `(w,h)`, respectively. Note that `ref` and `strm` share the same memory area of `w`×`h` stream elements.

### Kernels

A kernel, which operates on each stream element, is described by a special function specified by the `kernel` keyword. A programmer cannot permute the sequence of stream elements; they may independently be processed in parallel.

A kernel function with the `map` qualifier takes one or more output streams specified by the `out` or `inout` keyword. The kernel execution is a data-parallel task that logically computes all the output stream elements. For example, the `saxpy` function in Listing 2.3 is implicitly translated by the SPRAT compiler to the multiple codes, each of which has the same meaning of the nesting loops shown in Listing 2.2. In this listing, `N` and `M` indicate the width and height of streams.

Every input stream element is corresponding to one of elements in an output stream based on the position in the input stream by default. However, a stream specified by the `gather` qualifier can be accessed using an array index operator. A gather stream element

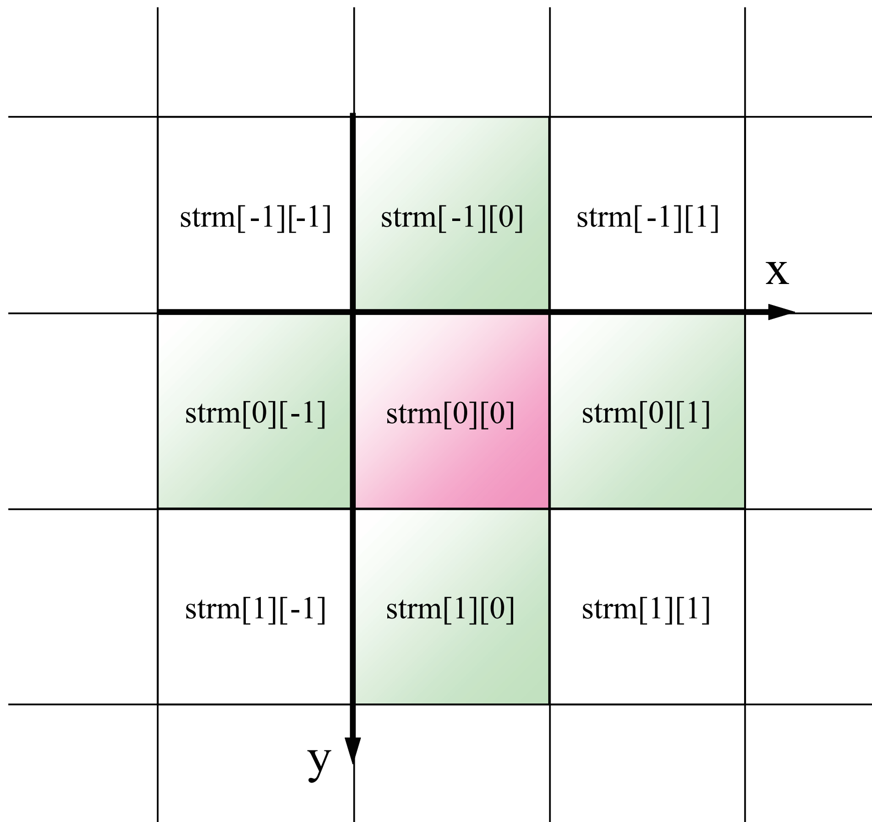



Figure 2.3: The coordinate in gather access.

is accessed using the array index that indicates the relative position from the corresponding output stream element. For example, `strm[0][0]` of a 2-dimensional `gather` stream denotes a stream element whose position is the same as that of the calculated output stream element; `strm[-1][0]`, `strm[1][0]`, `strm[0][-1]`, and `strm[0][1]` indicate the four neighboring elements of `strm[0][0]`, as shown in Figure 2.3. As a stream processing task often accesses the neighboring stream elements, the array index operator indicates the relative position from the corresponding output stream element. If a kernel function must always refer to the `i`-th element of stream `s`, it can be expressed using a special index operator, `s[[i]]`.

Listing 2.3 shows a sample code written in the SPRAT language. In the code, a kernel function `saxpy` is called with a scalar value `pi`, two input streams `sX` and `sY`, and an output stream `sZ`. Here, each of `sX`, `sY`, and `sZ` contains the same number of stream elements.

Listing 2.3: A sample code of saxpy written in the SPRAT language.

```
kernel map saxpy(
      float a,
      in stream<float> x,
      in stream<float> y,
      out stream<float> z)
{
   z = a * x + y;
   return;
}

int main(int argc, char** argv)
{
   stream<float> sX(N,M), sY(N,M), sZ(N,M);
   float x[N*M], y[N*M], z[N*M], pi=3.14f;

   init_array( x, y);
   streamRead(sX, x);
   streamRead(sY, y);
   saxpy(pi, sX, sY, sZ);
   streamWrite(sZ, z);
   print_array(z);
   return 0;
}
```

Stream elements at the same position in different streams are corresponding to each other. Every element of sX is multiplied by pi, and then is added to its corresponding element of sY. The calculation result is written to the corresponding element of sZ. Since a stream element is accessible only within a kernel function, elements of arrays x and y are copied onto sX and sY using streamRead function. Similarly, elements of sZ are copied onto z using streamWrite function.

### 2.3.3 Performance Prediction and Processor Selection

To select an appropriate processor, the SPRAT runtime environment profiles the execution times of kernels, and builds linear prediction models for each processor using profile data. The runtime environment automatically selects the processor that can execute a kernel in the minimum execution time.

#### Performance Modeling of Kernels

If a programmer directly writes a GPU computing code using graphics APIs and/or CUDA, styles of the implementations should be considered in performance prediction because they considerably affect the performance [24, 25, 26]. As SPRAT sacrifices the flexibility of programming styles to some extent, a kernel code written in the SPRAT language is automatically translated into multiple codes corresponding to a CPU and a GPU. Therefore, by

taking into account the translation rules, it is needed to consider only one style of the implementation for performance prediction, which becomes tractable in performance prediction.

In a stream processing task, an identical kernel computation is performed to produce each element in an output stream. Hence, the execution time of a kernel almost linearly increases with the number of output stream elements. To restrain the runtime overhead, a simple model is preferable for runtime performance prediction. Therefore, the SPRAT framework uses a simple linear performance model to estimate the execution time of a kernel from the number of elements in an output stream:

$$T_p(k_i) = \frac{D(k)}{B_p(k_i)} + S_p(k_i), \tag{2.4}$$

where $T_p(k_i)$ is the execution time required by the processor $p$ to execute kernel $k_i$, $D(k_i)$ is the number of output stream elements, $B_p(k_i)$ is the sustained throughput of the processor $p$ for kernel $k_i$, and $S_p(k_i)$ is the startup time required by the processor $p$ to launch kernel $k_i$.

Similarly, the data transfer time also increases with the size of the transferred stream data [23]:

$$T_{p\to q} = \frac{D_{p\to q}}{B_{p\to q}} + S_{p\to q}, \tag{2.5}$$

where $T_{p\to q}$, $D_{p\to q}$, $B_{p\to q}$, and $S_{p\to q}$ are the total time, the data size, the sustained bandwidth, and the startup time of the data transfer from the processor $p$ to the processor $q$, respectively.

In general, the linear performance model presented above can precisely estimate the execution time of a kernel on a GPU. However, if the execution times are profiled for only the streams whose sizes are small, it may overestimate the CPU performance without considering the effects of cache spilling of large streams. One possible approach is to use additional performance parameters, $B'_{p,k_i}$ and $S'_{p,k_i}$, to estimate the CPU performance when the kernel is called with a large stream whose size $D_{k_i}$ exceeds the cache capacity. Accordingly, the modified performance model estimates the execution time required by CPU to execute kernel $k_i$ by

$$T_{C,k_i} = \begin{cases} \frac{D_{k_i}}{B_{C,k_i}} + S_{C,k_i} & \text{if } D_{k_i} < D_\$ \\ \frac{D_{k_i}}{B'_{C,k_i}} + S'_{C,k_i} & \text{otherwise} \end{cases}, \tag{2.6}$$

where $D_\$$ is the size of the last level cache. The SPRAT runtime environment can get the value of $D_\$$ by using system query API.

### Energy Consumption Modeling of Kernels

In addition to performance parameters, power consumption parameters are also required for energy-aware computing. In this section, it is assumed that the power consumption is independent from kernels and depends only on the processor that executes kernels. Then, power consumption parameters can be regarded as system-specific parameters and need to be measured once for a system. The validity of this assumption is experimentally discussed later in Section 2.4.

Let $P_p$ be the wattage when using the processor $p$ as a computing engine. Similarly, $P_{p\to q}$ denotes the wattage for the data transfer from processor $p$ to processor $q$. This allows even a simple watt meter to measure the wattages such as $P_p$ by running a test kernel that executes `saxpy` on the GPU and the CPU for a long time; this does not need an expensive measuring instrument. $P_{p\to q}$ is also measured by transferring data from the memory of processor $p$ to the memory of processor $q$ for a long time.

Using these coefficients such as $P_p$ and $P_{p\to q}$, power consumption of executing kernels and data transfer can be indicated by

$$E_p(k_i) = P_p \cdot T_p(k_i) = P_p \left\{ \frac{D(k_i)}{B_p(k_i)} + S_p(k_i) \right\}, \tag{2.7}$$

and

$$E_{p\to q} = P_{p\to q} \cdot T_{p\to q} = P_{p\to q} \left\{ \frac{D_{p\to q}}{B_{p\to q}} + S_{p\to q} \right\}, \tag{2.8}$$

where $E_p(k_i)$ and $E_{p\to q}$ are the amount of energy consumption of executing the kernel $k_i$ on the processor $p$ and data transfer $p$ to $q$, respectively. Moreover, to accurately estimate the performance of a CPU with cache memory, the modified prediction model estimates the energy consumption of the CPU executing kernel $k_i$ by

$$E_C(k_i) = P_C(k_i) \cdot T_C(k_i) = \begin{cases} P_C(k_i) \left( \dfrac{D_{k_i}}{B_{C,k_i}} + S_{C,k_i} \right) & \text{if } D_{k_i} < D_\$ \\ P_C(k_i) \left( \dfrac{D_{k_i}}{B'_{C,k_i}} + S'_{C,k_i} \right) & \text{otherwise.} \end{cases}, \tag{2.9}$$

### Measuring Coefficients for Prediction

Performance prediction by SPRAT requires all the parameters in advance. Among the parameters, the parameters involved in the execution time of the kernel $i$ such as $B(k_i)$ and

$S(k_i)$ must be obtained for individual kernels; the others are system-specific parameters and are measured only once for a computing system. For measuring those performance parameters, an application user needs to run the program several times in advance. For example, the performance parameters are obtained by the first 10 executions; five times for the CPU parameters and the others for the GPU ones. Appropriate data sizes are properly given by the user to improve the accuracy of the prediction. The parameters of executing the kernel are measured and are automatically stored in a parameter database.

**Strategies of Processor Selection**

A CPU can access only the data on the main memory, while a GPU can access only on the device memory. When a computing engine is switched, the stream data to be accessed by the kernel function have to be transferred to the memory space of the new processor. Since the data transfer induces a considerable overhead, the dynamic switching has to be carefully decided by taking into account the trade-off between the data transfer overhead and the performance gain by switching.

Similar to Equation (2.3), one may claim that the appropriate processor can be found by comparing the execution time of a CPU to that of a GPU. That is, in the case where a CPU is currently selected as a computing engine, a computing engine for executing kernel $k_i$ should be switched to a GPU only if the following condition is met.

$$T_C(k_i) > T_G(k_i) + T_{C \to G}, \tag{2.10}$$

where processors $C$ and $G$ denote the CPU and the GPU, respectively.

However, $T_{C \to G}$ is generally larger than $T_C(k_i)$ and $T_G(k_i)$ especially if $k_i$ is a user-defined simple kernel function. Even if a GPU is much faster than a CPU and can reduce the total execution time, the processor will not be switched from the CPU to the GPU. Accordingly, Equation (2.10) does not always result in appropriate processor selection. Suppose that $T_C(k_i)$ is greater than $T_G(k_i)$ but less than $T_G(k_i) + T_{C \to G}$. Then, the GPU can reduce the total execution time by almost $\sum_{i=1}^{N} \{T_C(k_i) - T_G(k_i)\}$, if the processor is switched from the CPU to the GPU and then kernel $k_i$ is called $N$ times. Accordingly, if $N$ satisfies the

following condition, a processor should be switched from the CPU to the GPU.

$$T_{C\to G} < \sum_{i=1}^{N} \left\{ T_C(k_i) - T_G(k_i) \right\}. \tag{2.11}$$

In many cases, however, it is difficult to obtain the actual stream size $D(k_i)$ used for estimating $T_p(k_i)$ and $T_{p\to q}$ in advance of the kernel call.

To solve this problem, the *accumulated time difference* is introduced to SPRAT for appropriate processor selection. In the case where processor $p$ is used as a computing engine, the accumulated time difference of processor $q$ is updated at every kernel call as follows.

$$\Delta T_q := \max\{\Delta T_q + (T_p(k_i) - T_q(k_i)), 0\}. \tag{2.12}$$

Hence, $\Delta T_q$ indicates how much the execution time is shorten if $q$ is used as a computing engine. SPRAT switches a computing engine from $p$ to $q$ if the following condition is met.

$$T_{p\to q} < \Delta T_q. \tag{2.13}$$

When the computing engine is switched from $p$ to $q$, the accumulated time difference of each processor is cleared to 0,

$$\Delta T_p \quad := \quad 0, \tag{2.14}$$

and

$$\Delta T_q \quad := \quad 0. \tag{2.15}$$

In the case of energy-aware computing, the *accumulated energy difference* is also introduced to SPRAT. When the processor $q$ is not used as a computing engine, the accumulated energy difference of processor $q$ is updated at every kernel call as follows.

$$\Delta E_q := \max\{\Delta E_q + (E_p(k_i) - E_q(k_i)), 0\}. \tag{2.16}$$

Hence, $\Delta E_q$ indicates how much energy is saved if $q$ is used as a computing engine and is used instead of $\Delta T_q$ in energy-aware processor selection.

These metrics assume iterative computations and realize a lightweight runtime environment that can select an appropriate processor based on the trend of stream sizes in the past. Thereby, these metrics speculatively customize an application program to common stream sizes in the future kernel calls. Although this needs the extra execution time or energy of $T_{p \to q}$ or $E_{p \to q}$ until switching the computing engine to the other processor, it will be negligible in many cases where a sequence of some kernels is periodically invoked many times.

## 2.4 Evaluation

### 2.4.1 Experimental Setup

This section shows the evaluation results to examine the performance of the proposed run-time processor selection for performance-aware and energy-aware computing. All of the evaluation results are obtained using a Linux PC equipped with Intel Core 2 Quad (C2Q) Q6600 Processor running at 2.4GHz, DDR2 4GB main memory, and one of GPUs listed in Table 2.1. In the table, # SPs indicates the number of streaming processors in the GPU. Mem, CFreq, MFreq, and BW indicate the memory capacity, the core clock frequency, the memory clock frequency, and the peak memory bandwidth, respectively. GFGTX280 means NVIDIA GeForce GTX280. GF88GTX is the abbreviation of NVIDIA GeForce 8800 GTX, and the other GPU models are also abbreviated in the same way.

The Linux kernel version is 2.6.18 (CentOS 5 x86_64) and C++ compiler that compiles the SPRAT runtime environment and codes for CPUs automatically-generated by the SPRAT compiler is gcc-4.2.1 with "-O3" options. The NVIDIA graphics driver version 173.14, the CUDA version 1.1, and CUDA compiler version 1.1 V0.2.1221 are used. At every execution of a SPRAT program, a CPU is initially selected as the computing engine for kernel execution, and then the computing engine is switched according to runtime performance prediction.

### 2.4.2 Evaluation of Performance-aware Processor Selection

To clarify the effectiveness of the SPRAT framework for a programmer who does not have knowledge of GPU programming, the computational fluid dynamics (CFD) simulation code

Table 2.1: Specifications of GPUs used for evaluation.

| Processor Name | Abbreviated Name | # SPs | Mem. [MB] | CFreq. [MHz] | MFreq. [MHz] | BW. [GB/s] |
|---|---|---|---|---|---|---|
| GeForce GTX 280 | GFGTX28 | 240 | 1024 | 1296 | 1107 | 141.7 |
| GeForce 8800 GTX | GF88GTX | 128 | 768 | 1350 | 900 | 86.4 |
| GeForce 8800 GT | GF88GT | 112 | 512 | 1500 | 900 | 57.6 |
| GeForce 8600 GTS | GF86GTS | 32 | 256 | 1450 | 1000 | 32.0 |
| Core 2 Quad Q6600 | C2Q | — | 4096 | 2400 | 800 | 12.8 |

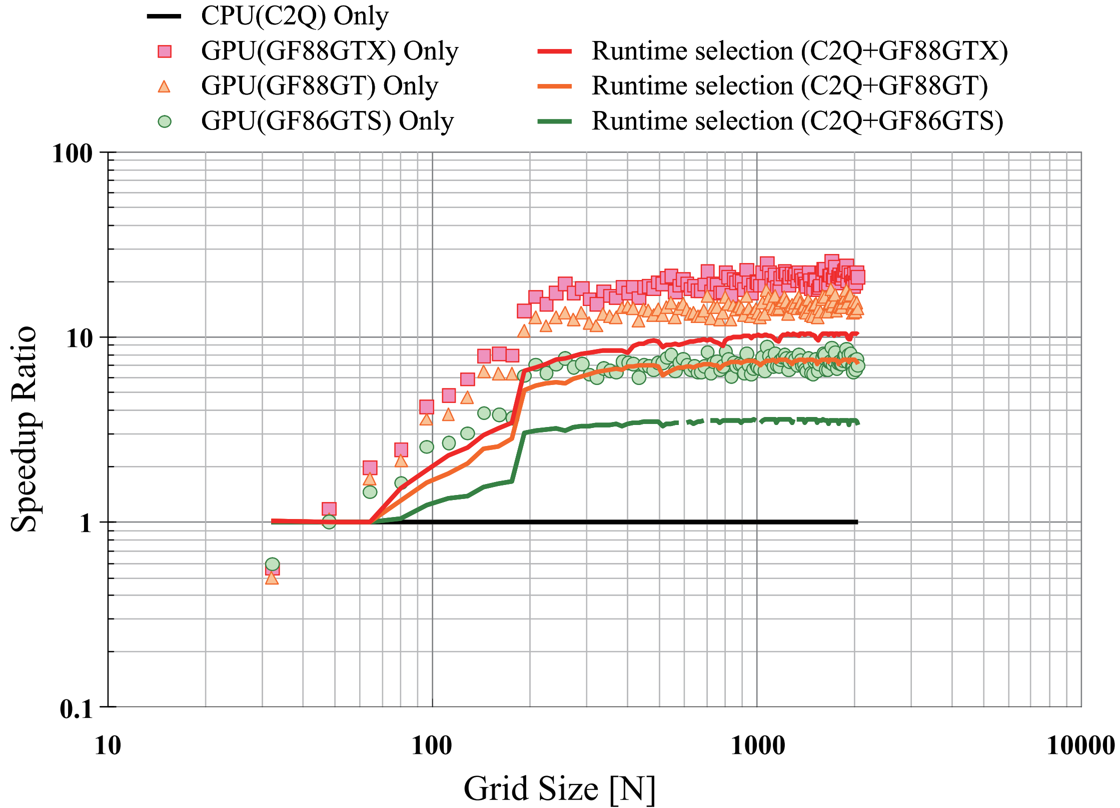



Figure 2.4: Speedup ratio of the CFD simulation.

shown in Listing 2.4 is used for the following evaluation. In this evaluation, the CFD program calculates 100 simulation steps of the 2-dimensional cavity flow using the fractional step method [33]. The Jacobi iteration method is used for the pressure calculation. In the case where the pressure calculation is performed on GPU, the error is transferred from GPU to CPU to check the convergence. In this evaluation, a convergence difference is calculated, but it is not used to exit the loop of pressure calculation.

To measure prediction parameters for each processor, the CFD program is executed for every kernel with the grid sizes of $32 \times 32$, $64 \times 64$, $128 \times 128$, $256 \times 256$, and $512 \times 512$. Moreover, the additional prediction parameters of CPU for large streams are measured to consider the capacity misses with the grid sizes of $1024 \times 1024$, $1536 \times 1536$, $2048 \times 2048$, $2560 \times 2560$ and $3072 \times 3072$.

Figure 2.4 shows the speedup ratio of three GPUs measured with changing the data size. In Figure 2.4, the performance is obtained by counting the number of grid points per second, and then the speedup ratio of each processor to CPU is computed.

For the CFD code, all the GPUs significantly outperform C2Q in large grid sizes. However, in very small grid sizes, GPUs cannot outperform C2Q because there are overheads for executing kernels on GPUs such as data transfers. Moreover, the cache memories of C2Q provide a higher memory bandwidth than those of GPUs. Hence the performance is degraded if the computing engine is fixed to GPUs when the grid size is very small.

If one processor is much faster than the other, SPRAT can easily select the appropriate processor. In the evaluation results shown in Figure 2.4, all the GPUs are almost always used as a computing engine, and C2Q is used only when the grid size is approximately equal to or less than $64 \times 64$. The evaluation results clarify that even a middle-range GPU, GF86GTS, has a possibility to achieve a high performance, if the application is well-suited for GPU computing. In this case, SPRAT enables programmers who do not have knowledge of GPU computing to appropriately exploit the computing power of a middle-range GPU without risks of performance degradation.

In most cases, the performance using runtime processor selection is less than the performance especially when a computing engine is fixed to an appropriate processor. This difference in performance appears when the appropriate processor is a GPU. This is because a CPU is selected as an initial computing engine, and there is a certain overhead of data transfers and kernel launch to switch the computing engine from a CPU to a GPU. However, it would be negligible if the initial overhead becomes small compared to the total execution time.

### 2.4.3 Evaluation of Energy-aware Processor Selection

To realize runtime processor selection for energy-aware computing, the power consumption parameters required to predict energy consumptions are measured in advance. Table 2.2 shows the power consumption of each system configuration under full load. The power consumption is measured with HIOKI HiTESTER 3334 [34] while the `saxpy` function in Listing 2.3 is iteratively running on either a CPU or a GPU.

Table 2.2: Power consumption of each system configuration.

| System Configuration | $P_G$ [W] | $P_C$ [W] | $P_{C \to G}$ [W] | $P_{G \to C}$ [W] |
|---|---|---|---|---|
| C2Q + GF88GTX | 251.4 | 204.9 | 214.6 | 213.2 |
| C2Q + GFGTX28 | 305.3 | 212.7 | 210.3 | 209.2 |

In Table 2.2, it can be seen that $P_C$ increases with the GPU's peak power consumption. Thus, such a high-performance GPU consumes a considerable power even if it does not compute anything. If a GPU cannot efficiently execute an application, hence, the best way to save energy for the application is to detach the GPU from the system. However, it is not practical to change the hardware configuration according to each application. In the cases of GPUs, the GPU's power consumption in the idle state is not negligible. However, it is solved by the GPU that has a more advanced power management capability and can remarkably reduce the power when it is idle. Accordingly, for such a GPU, runtime processor selection will become more effective for saving the energy consumption.

In this evaluation, the effect of runtime processor selection on energy-aware computing is shown. In addition, it is also shown that the intersection point of changing the appropriate processor is different between performance-aware computing and energy-aware computing to indicate that high sustained performance does not always lead to high energy efficiency.

Listing 2.5 shows the SPRAT code of no-pivoting LU decomposition, which iterates `normalize` and `rowop` with decreasing the stream size. This benchmark is purposely implemented to degrade the sustained performance on GPUs to demonstrate that SPRAT can properly select a CPU as a computing engine when a GPU cannot work well. This code has been ported from LU-GPU [35]. As the initial address of the input stream is changed at every kernel call, a GPU cannot perform coalesced memory access [3] in most cases. As a result, the sustained memory bandwidth of a GPU severely degrades when executing the LU decomposition.

The prediction parameters of each processor for every kernel in the LU decomposition are measured with the data sizes of $32 \times 32$, $64 \times 64$, $128 \times 128$, $256 \times 256$, and $512 \times 512$. To consider the capacity misses, the additional performance parameters of the CPU for large streams are measured with the data sizes of $1024 \times 1024$, $1536 \times 1536$, $2048 \times 2048$, $2560 \times 2560$ and $3072 \times 3072$.

Figure 2.5 shows the speedup ratios of C2Q and two GPUs measured with changing the data size. In Figure 2.5, the sustained performance is calculated based on the number of floating-point operations per second, and then the speedup ratio of each processor to C2Q is calculated.

Though the LU decomposition is a typical memory-intensive benchmark, GPUs cannot efficiently execute the LU decomposition code because of non-coalesced memory accesses; GF88GTX increases the power consumption, even though it does not reduce the execution

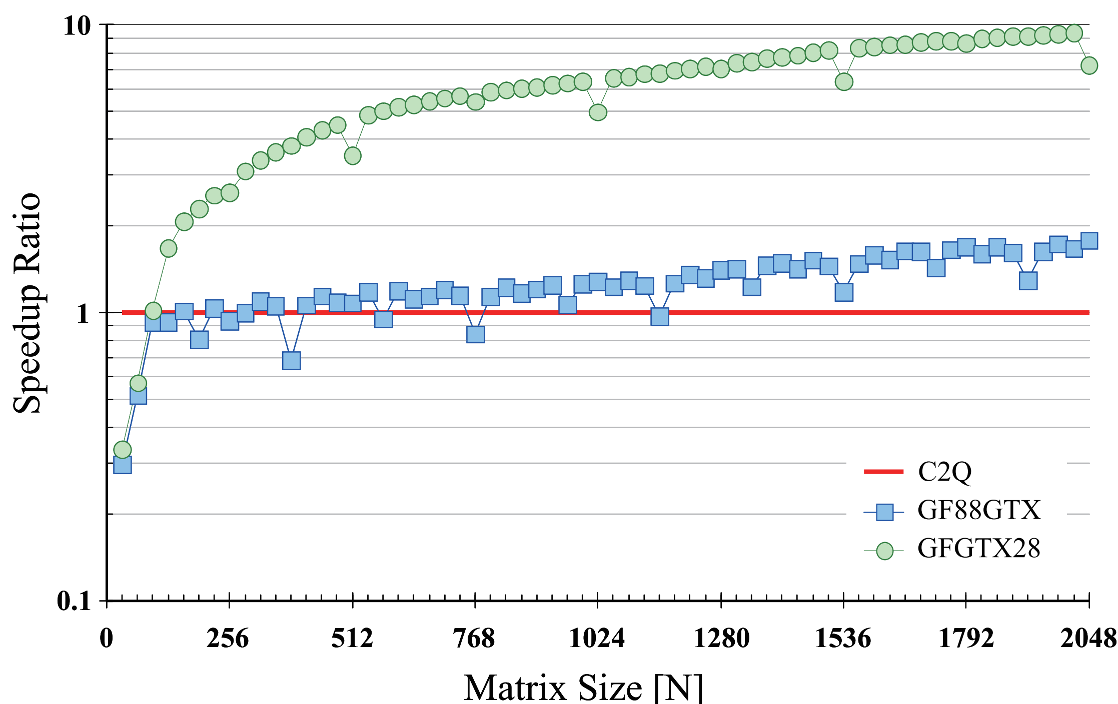



Figure 2.5: Speedup ratio of the LU decomposition.

time. As a result, the use of those GPUs may decrease the energy efficiency. In some matrix sizes, there is an obvious trade-off between the performance and the energy consumption because the execution time is not reduced enough to hide an increase in power consumption induced by using GPUs. Since the performance improvement depends on the data size often determined at runtime, the computing engine should also be dynamically selected at runtime.

Figure 2.6 shows the sustained performances of fixed processors and runtime processor selection for performance-aware and energy-aware policies. In the energy-aware selection, unlike the processor selection based on the performance-aware policy, the proposed method selects C2Q even if GF88GTX is somewhat faster than C2Q. Figure 2.7 shows the numbers of floating-point operations per watt hour [Gflop/s/Wh] of C2Q and GF88GTX. In the cases of small matrix sizes, as a CPU can effectively use cache memory and does not access off-chip memory, the energy efficiency of a CPU is higher than that of GPUs. On the other hand, when the matrix size is sufficiently large, the energy efficiency of the GPU becomes smaller than that of the CPU because GPUs can reduce the execution time that it is enough to cover additional power consumption due to GPUs. In this case, SPRAT uses GF88GTX only if GF88GTX outperforms C2Q in terms of energy efficiency.

On the other hand, Figures 2.8 and 2.9 show the sustained performance and energy efficiency in the system consisting of C2Q and GFGTX28. Even though the LU decomposition causes non-coalesced memory accesses, GFGTX28 is much faster than C2Q because of its cache memory. In this system, GFGTX280 is an appropriate processor in most matrix sizes for both performance-aware and energy-aware selections. However, in very small matrix sizes, there are some cases that the energy efficiency of C2Q is higher than that of GFGTX28 because of the overhead of data transfer. In this case, SPRAT can also select GFGTX28 only if GFGTX28 is superior to C2Q in terms of energy efficiency.

All the above results were obtained by assuming that the power consumption of each processor is constant irrespective of the workload. Thus, these results clearly indicate that SPRAT can properly select the processor by estimating the energy consumption from the preliminarily-obtained parameters.

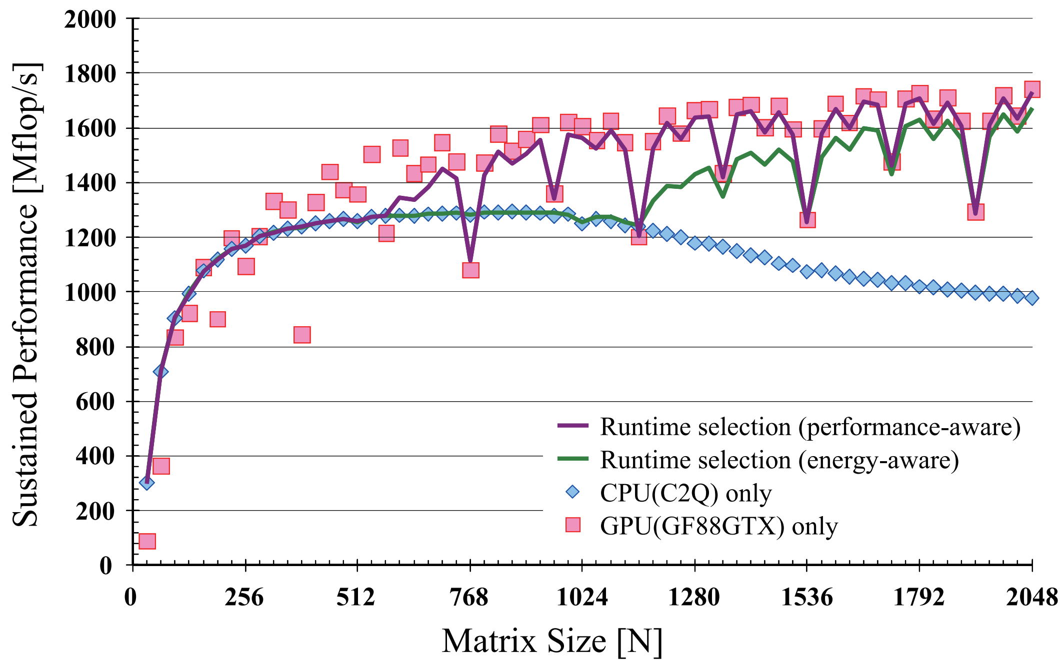



Figure 2.6: Sustained performance of C2Q and GF88GTX.

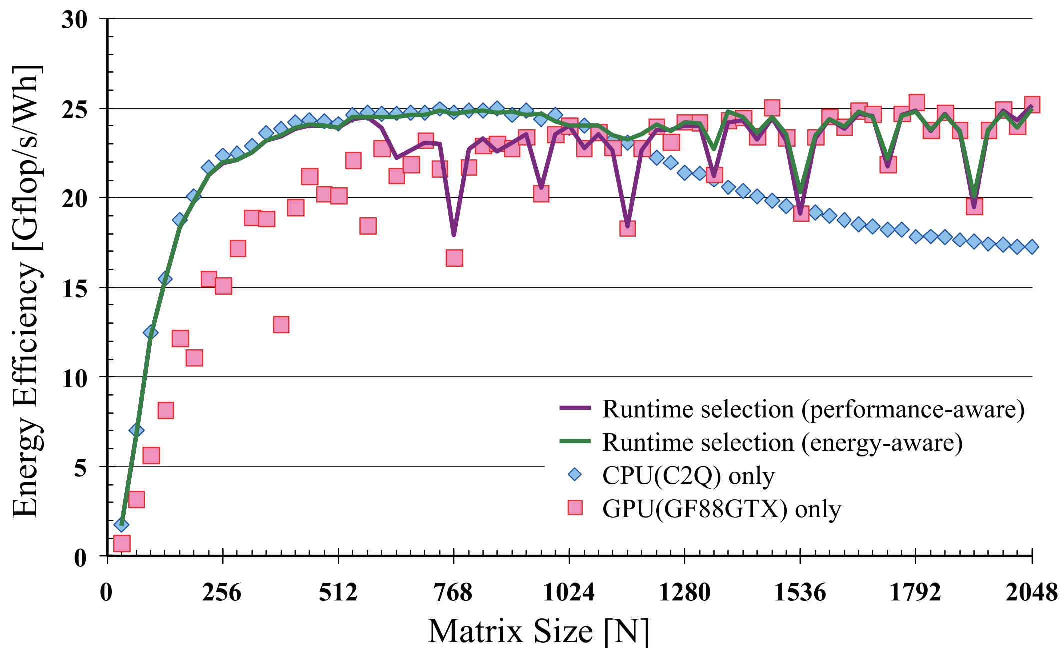



Figure 2.7: Energy efficiency of C2Q and GF88GTX.

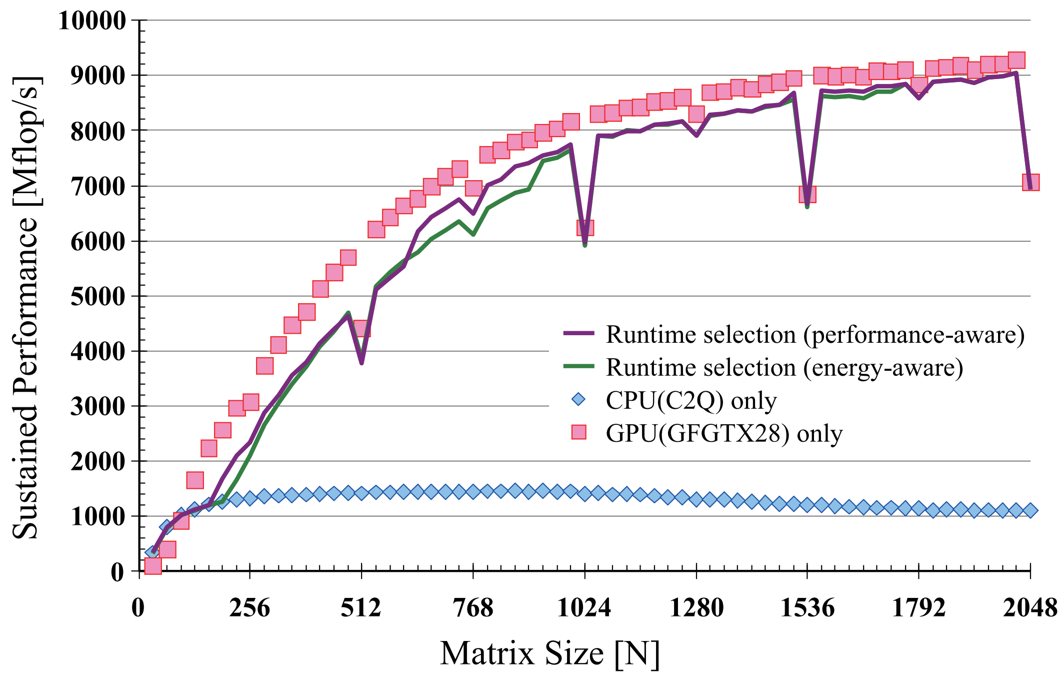



Figure 2.8: Sustained performance of C2Q and GFGTX28.

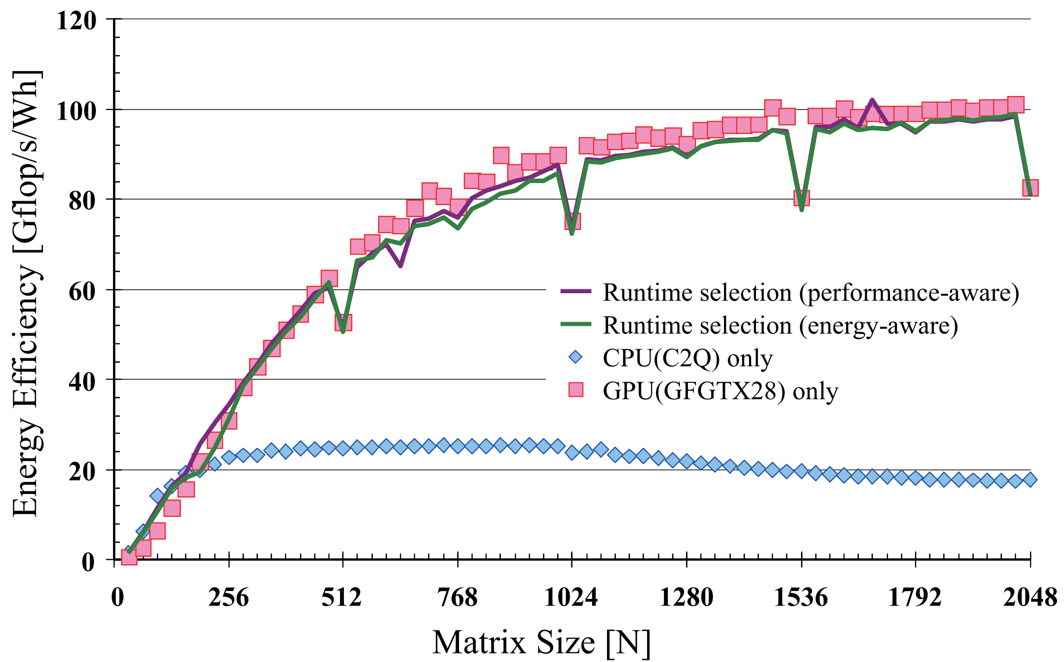



Figure 2.9: Energy efficiency of C2Q and GFGTX28.

## 2.5 Concluding Remarks

In this chapter, a programming framework consisting of a domain-specific programming language and its runtime environment, named SPRAT, is proposed to achieve energy-aware and performance-aware computing in a heterogeneous computing system of a CPU and a GPU. The SPRAT compiler translates a SPRAT code to a C++ code for CPUs and a CUDA code for GPUs, respectively. Then, the proposed mechanism can automatically select the appropriate processor at runtime by taking into account the difference in energy efficiency or sustained performance between a CPU and a GPU.

The evaluation results clearly indicate that the SPRAT framework enables programmers who do not have knowledge of GPU computing to exploit GPUs without risks of performance degradation. Moreover, the proposed processor selection can further increase the overall performance. The evaluation results also demonstrate that SPRAT can use GPUs only if the performance per watt hour of the GPU exceeds that of the CPU, even though SPRAT employs a rough approximation scheme to estimate the energy consumption.

From the evaluations, it is demonstrated that the proposed programming framework can automate runtime processor selection and solve one of the difficulties in programming heterogeneous computing systems. By using this framework, programmers can describe programs without careful processor selection. The proposed method will be applicable for various heterogeneous computing systems because it does not use architecture-specific prediction models. Even if a processor has cache memories, the proposed runtime environment can accurately predict the execution time by using different prediction models corresponding to the size of data. Therefore, if the execution time and the stream size are available, the proposed method can select an appropriate processor based on the linear prediction models.

To achieve a higher performance, the SPRAT compiler has to support some architecture-aware optimizations to generate more efficient kernels for GPUs. Hence, the next chapter deals with automatic optimization methods employed in the SPRAT compiler.

Listing 2.4: The CFD program written in the SPRAT language.

```
1  kernel map calc_u_mid(
2        out stream<float> u_mid,
3        gather stream<float> u,
4        gather stream<float> v,
5        float U_t, float U_b, float dt, float invRe)
6  {
7     float u_c = u[0][0];
8     float u_r = u[0][1];
9     float u_l = (kernel.index_d[0] == 0) ? 0.0f : u[0][-1];
10    float u_t = (kernel.index_d[1] == kernel.size_d[1] - 1)
11                ? (2 * U_t - u_c) : u[1][0];
12    float u_b = (kernel.index_d[1] == 0) ? (2 * U_b - u_c) : u[-1][0];
13    float v_tr = v[0][1];
14    float v_tl = v[0][0];
15    float v_br = (kernel.index_d[1] == 0) ? 0.0f : v[-1][1];
16    float v_bl = (kernel.index_d[1] == 0) ? 0.0f : v[-1][0];
17
18    float A = (((u_r + u_c)*(u_r + u_c)) - ((u_c + u_l)*(u_c + u_l)) +
19              ((u_t + u_c)*(v_tr + v_tl)) - ((u_c + u_b)*(v_br + v_bl)))/4.0f;
20    float B = u_r + u_l + u_t + u_b - 4 * u_c;
21
22    u_mid = u_c + dt * (-A + invRe * B);
23    return;
24 }
25
26 kernel map calc_v_mid(
27       out stream<float> v_mid,
28       gather stream<float> u,
29       gather stream<float> v,
30       float V_r, float V_l, float dt, float invRe)
31 {
32    float u_tr = u[1][0];
33    float u_tl = (kernel.index_d[0] == 0) ? 0.0f : u[1][-1];
34    float u_br = u[0][0];
35    float u_bl = (kernel.index_d[0] == 0) ? 0.0f : u[0][-1];
36    float v_c = v[0][0];
37    float v_r = (kernel.index_d[0] == kernel.size_d[1] - 1)
38                ? (2 * V_r - v_c) : v[0][1];
39    float v_l = (kernel.index_d[0] == 0) ? (2 * V_l - v_c) : v[0][-1];
40    float v_t = v[1][0];
41    float v_b = (kernel.index_d[1] == 0) ? 0.0f : v[-1][0];
42
43    float A = (((v_t + v_c)*(v_t + v_c)) - ((v_c + v_b)*(v_c + v_b)) +
44              ((v_r + v_c)*(u_tr + u_br)) - ((v_c + v_l)*(u_tl + u_bl)))/4.0f;
45    float B = v_r + v_l + v_t + v_b - 4 * v_c;
46
47    v_mid = v_c + dt * (-A + invRe * B);
48    return;
49 }
50
51 kernel map calc_div_u(
52       out stream<float> div_u,
53       gather stream<float> u,
54       gather stream<float> v,
55       float dt)
56 {
57    float u_r = u[0][0];
58    float u_l = (kernel.index_d[0] == 0) ? 0.0f : u[0][-1];
59    float v_t = v[0][0];
60    float v_b = (kernel.index_d[0] == 0) ? 0.0f : v[-1][0];
61
62    div_u = (u_r - u_l + v_t - v_b) / dt;
63    return;
64 }
65
66 kernel map calc_p_next(
67       out stream<float> p_next,
68       gather stream<float> p,
69       in stream<float> div_u)
70 {
71    float p_c = p[0][0];
72    float p_r = (kernel.index_d[0] == kernel.size_d[0] - 1) ? p_c : p[0][1];
73    float p_l = (kernel.index_d[0] == 0) ? p_c : p[0][-1];
74    float p_t = (kernel.index_d[1] == kernel.size_d[1] - 1) ? p_c : p[1][0];
```

```
75     float p_b = (kernel.index_d[1] == 0) ? p_c : p[-1][0];
76
77     p_next = 0.25f * (p_r + p_l + p_t + p_b - div_u);
78     return;
79  }
80
81  kernel map calc_p_error(
82        inout stream<float> p_next,
83        in stream<float> p)
84  {
85     float tmp = p_next - p;
86     p_next = (tmp >= 0.0f) ? tmp : -tmp;
87     return;
88  }
89
90  kernel reduce calc_max_error(
91        out float error,
92        in stream<float> p_error)
93  {
94     error = (p_error.R > p_error.L) ? p_error.R : p_error.L;
95     return;
96  }
97
98  kernel map calc_u_next(
99        out stream<float> u_next,
100       in stream<float> u,
101       gather stream<float> p,
102       float dt)
103 {
104    u_next = u - dt * (p[0][1] - p[0][0]);
105    return;
106 }
107
108 kernel map calc_v_next(
109       out stream<float> v_next,
110       in stream<float> v,
111       gather stream<float> p,
112       float dt)
113 {
114    v_next = v - dt * (p[1][0] - p[0][0]);
115    return;
116 }
117
118 int main()
119 {
120    stream<float> u_cur(SIZE_X, SIZE_Y);
121    stream<float> v_cur(SIZE_X, SIZE_Y);
122    stream<float> p_cur(SIZE_X, SIZE_Y);
123    stream<float> u_mid(SIZE_X, SIZE_Y);
124    stream<float> v_mid(SIZE_X, SIZE_Y);
125    stream<float> p_mid(SIZE_X, SIZE_Y);
126    stream<float> div_u(SIZE_X, SIZE_Y);
127    int outer_loop, inner_loop;
128    // -- (snip) --
129
130    TimerStart();
131    for (outer_loop = 0; outer_loop < 100; outer_loop++) {
132       calc_u_mid(u_mid(0,0,(SIZE_X-1),SIZE_Y), u_cur, v_cur, 0, U, dt, invRe);
133       calc_v_mid(v_mid(0,0,SIZE_X,(SIZE_Y-1)), u_cur, v_cur, 0, 0, dt, invRe);
134       calc_div_u(div_u(0,0,SIZE_X,SIZE_Y), u_mid, v_mid, dt);
135       error = FLT_MAX;
136
137       for(inner_loop = 0; inner_loop < 4; inner_loop++){
138          calc_p_next(p_mid, p_cur, div_u);
139          calc_p_next(p_cur, p_mid, div_u);
140          calc_p_error(p_mid, p_cur);
141          calc_max_error(error, p_mid);
142       }
143       calc_u_next(u_cur(0,0,(SIZE_X-1),SIZE_Y), u_mid, p_cur, dt);
144       calc_v_next(v_cur(0,0,SIZE_X,(SIZE_Y-1)), v_mid, p_cur, dt);
145    }
146    TimerStop();
147
148    // -- (snip) --
149    return;
150 }
```

Listing 2.5: The LU decomposition written in the SPRAT language.

```
kernel map rowop(
      gather stream<float> gath,
      out stream<float> ostr)
{
   ostr=gath[0][0]-gath[[-1]][0]*gath[0][[-1]];
   return;
}

kernel map normalize(
      gather stream<float> gath,
      out stream<float> ostr){
   ostr=gath[0][0]/gath[[-1]][0];
   return;
}

int main(int argc, char ** argv){
   stream<float> str(N,N);
   float origMat[N*N];
   int i;

   // -- (snip) --

   streamRead(str,origMat);
   for(i = 0; i < N-1; i++){
      stream<float>& s=str[i][i+1](1,N-i-1);
      normalize(s, s); /// kernel invocation
      stream<float>& s=str[i+1][i+1](N-i-1,N-i-1);
      rowop(s, s); /// kernel invocation
   }

   // -- (snip) --

   return 0;
}
```

# Chapter 3

# Automatic Performance Tuning for the Domain-specific Language

## 3.1 Introduction

To achieve high performance in CUDA, it is needed to indicate some *execution parameters* to execute tasks, and to optimize programs to effectively use architecture-specific features. These optimizations and tuning for a particular architecture are necessary to exploit the capability of accelerators.

Execution parameters are related to the *computational granularity*, which is the size of *Cooperative Thread Array* [36], called the *CTA configuration*. As the optimal CTA configuration depends on both the computation of a program and the GPU hardware architecture, it is required to select an appropriate CTA configuration for individual GPUs and programs [24]. Thus, a programmer has to take care of many CTA configurations in an error-prone trial-and-error manner for every application program.

Optimization of memory access patterns is necessary to hide the latency of memory accesses and also to improve the sustained memory bandwidth. In this optimization, use of a low-latency on-chip memory is a key feature to optimize the memory access patterns. However, optimization of memory access patterns is not easy because the overhead of inefficient memory accesses depends on the architecture of GPUs. Moreover, as the size of an on-chip memory is limited, programmers must carefully extract highly-reusable data blocks to store on the on-chip memory. Thereby, programmers have to modify programs for each

GPU, which may cause making bugs and performance degradation by inappropriate optimizations.

As described above, since those performance tunings in CUDA are complicated and labor-intensive, it is difficult even for expert programmers to appropriately optimize and tune CUDA programs. Therefore, it is strongly required to automate these optimizations and tuning to easily exploit the capability of GPUs.

To alleviate difficulties in GPU programming, the SPRAT programming framework has been proposed in Chapter 2 to automate runtime processor selection. The SPRAT compiler outputs a *CPU code* written in the C++ language for a CPU and a CUDA code for GPUs, respectively. Several automatic optimization methods are implemented in the compiler for CPUs, and it can be expected that those methods are automatically applied to the CPU code. On the other hand, a compiler for a GPU cannot sufficiently optimize a CUDA code because CUDA assumes that programmers explicitly apply these optimizations and tuning.

As a SPRAT program is automatically translated into multiple codes, it is impossible to describe a highly optimized SPRAT program for a particular GPU. This problem arises not only in SPRAT, but also in another high-level programming framework such as hiCUDA [37] and HMPP[19]. Therefore, when a CUDA code is automatically generated by the compiler, it is needed to automatically apply those optimizations and tuning for the CUDA code to improve the sustained performance.

In this chapter, to automate architecture-specific optimizations and tuning in CUDA, automatic performance tuning methods are proposed to improve performance of automatically-generated CUDA programs. Firstly, this chapter shows architectural restrictions and features of GPUs. The scope of promising CTA configurations and effective use of memory hierarchy in GPUs are shown based on those restrictions and features. On that basis, two optimization methods of memory access patterns and a tuning method of the CTA configuration are proposed. Finally, it is demonstrated that these proposed optimizations and tuning can automatically improve the performance of a CUDA program.

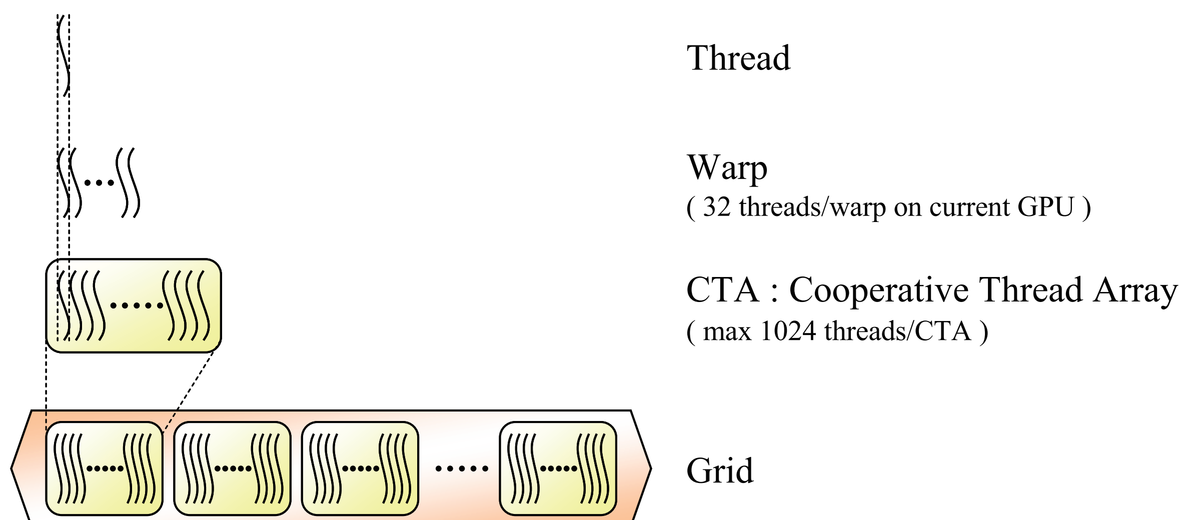



Figure 3.1: The thread hierarchy in CUDA.

## 3.2 Related Work

### 3.2.1 Performance Tuning in CUDA

CUDA is a programming framework for GPU computing provided by NVIDIA [3]. To exploit the GPU potential, CUDA requires careful code optimizations and parameter tunings. Therefore, a programmer often needs to explore several optimizations and many parameter configurations in a trial-and-error fashion to find the optimal one. To achieve high performance in CUDA, it is especially important to optimize memory access patterns and adjust the execution parameters.

Figure 3.1 illustrates the thread hierarchy in CUDA. A grid is a term to denote a set of all threads launched for one kernel execution. A grid is decomposed into several CTAs of the same size, and every CTA is assigned to a MP for parallel execution. In the current GPUs, a CTA is further decomposed into warps each consisting of 32 threads. Each warp is executed on a MP in an SIMD manner.

The configuration of a CTA determines the granularity of a computing task. Although the CTA configuration is one of important factors that affects the sustained performance of GPU computing, the optimal parameter configuration depends on both a program and a GPU architecture. Therefore, it is difficult and labor-intensive even for expert programmers to optimize those parameters according to both a computation and a GPU architecture.

One key to achieve high performance is to exploit the memory hierarchy in the hardware architecture, as shown in Table 3.1. In the CUDA programming, a *global memory* and a *shared memory* are generally used. The global memory has the largest memory space in the memory hierarchy, but a global memory access needs a long access latency. In addition, the memory bandwidth strongly depends on its memory access pattern. 16 threads in a warp, called a *half-warp*, simultaneously access the global memory. These memory accesses are coalesced and executed as one efficient *coalesced memory access* only if some hardware-generation-dependent conditions are met.

The shared memory has only a small capacity, but its access latency is much shorter than the global memory access latency. The data on the shared memory can be accessible from threads in the same CTA. Hence, the shared memory is usually used to cache reusable data.

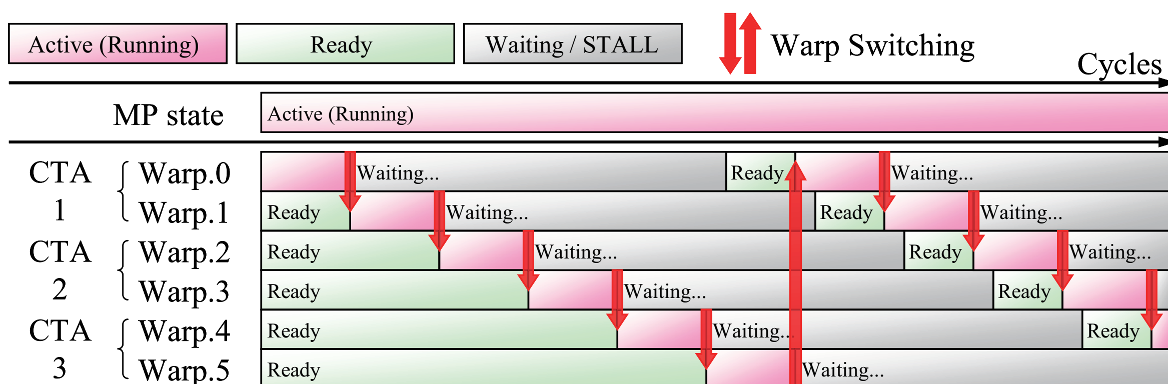
Since each of the global memory and the shared memory has advantages and disadvantages, it is required to appropriately use both of them. Memory coalescing and data prefetching [24] are two important optimization techniques always used to improve the performance of CUDA applications. Therefore, there is a strong demand for automation of those techniques.

The other key is to adjust CUDA execution parameters, i.e. the CTA configuration. When the number of threads in a CTA, called a *CTA size*, is too large, the shared memory capacity required by each CTA also becomes large. In this case, the number of CTAs per MP becomes small. As a result, the number of CTAs executed in parallel decreases, resulting in performance degradation.

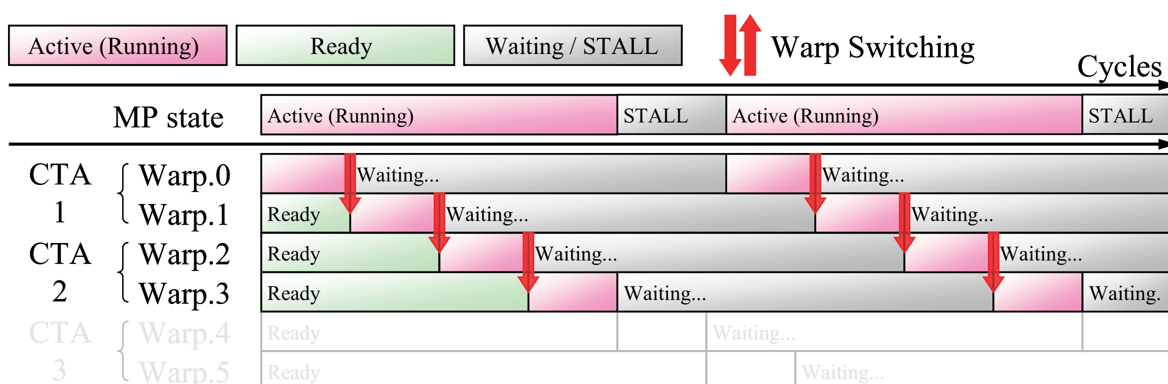
Naruse et al. have reported that tuning of the CTA configuration is important to achieve high performance [38]. They apply several optimizations and tuning to the Himeno benchmark [39] and investigate the effects of the optimizations and tuning. In their work, it is

Table 3.1: The memory hierarchy in CUDA.

| Memory Type | placement | Memory Size | Access Delay(cycle) | Read/Write |
|---|---|---|---|---|
| Global memory | Off-chip | $\leqq$6GB | 200 $\leqq$ | R/W |
| Local memory | Off-chip | $\leqq$ Global | $\fallingdotseq$ Global | R/W |
| Shared memory | On-chip | 16 or 48 KiB/SM | $\fallingdotseq$ Registers | R/W |
| Constant memory | On-chip | 64 KiB/chip | $\fallingdotseq$ Registers | Read Only |
| Texture memory | On-chip | $\leqq$ Global | 100 $\leqq$ | Read Only |




(a) The number of warps assigned to one MP is enough to hide memory access latency.




(b) The number of warps assigned to one MP is not enough to hide memory access latency.

Figure 3.2: Warp switching execution in CUDA.

demonstrated that tuning of the CTA configuration affects the sustained performance and is as important as the optimizations of memory access.

In a GPU architecture, when threads are stalled at accessing the global memory, the MP hides the latency by switching the stalled threads to another ready thread [36]. Figure 3.2 shows the cases that the number of warps assigned to one MP is enough and not enough to hide the global memory access latency. If the number of warps assigned to a MP is sufficiently large, the MP can fully run. Although a warp is stalled at every memory access, the MP switches the warp to another warp that is ready to be executed. The stalled warps wait for data to come when the another warp is executed. However, if the number of warps assigned to a MP is not enough, the MP cannot hide the memory access latency and is stalled.

This technique to hide the latency is called *switch-on-event multithreading* and is adopted also in Sun Ultra SPARC T1 architecture [40]. Since the switch-on-event multithreading

needs massive threads, the CTA size should be sufficiently large, and a small CTA results in decreasing the number of the warps assigned to a MP and thereby degrading the performance. Accordingly, in parameter tuning of the CTA configuration, both the shared memory usage and the number of threads in a CTA must be carefully considered [24].

### 3.2.2 Optimization Tools for GPU computing

Ueng et al. have proposed CUDA-lite to assist writing a code with efficient memory access patterns [41]. In CUDA-lite, a programmer inserts special annotations to specify reusable data blocks in a code, and a compiler translates the annotated code into a standard CUDA code using the shared memory for caching the reusable data. Therefore, a programmer does not need to write a different code for each GPU hardware generation to achieve efficient memory accesses.

Han et al. have proposed hiCUDA that generates a CUDA code from a sequential C code with special annotations [37]. In hiCUDA, special `pragma` annotations, "#pragma hicuda," are inserted into a C code to indicate the code blocks. The code blocks are translated into CUDA codes and are executed on a GPU. In a similar approach, Oshima et al. have proposed OMPCUDA that translates a C code with OpenMP [42] annotations to a CUDA code [43]. PGI compiler [18] and HMPP [19] are software products that provide compilers to translate C or FORTRAN codes with special annotations to CUDA codes. These approaches provide some special annotations to optimize data transfer, memory usages, the CTA configuration, and so on. To achieve high performance in these approaches, a programmer must insert appropriate annotations. Hence, it is required for a programmer to have knowledge of those special annotations and GPU architectures.

All the above programming environments can relax the programming efforts required to develop CUDA applications. However, they still expect that a programmer properly specifies appropriate annotations corresponding to memory accesses and the CTA configuration. For example, in the case of HMPP, the CTA configuration is set to a certain default value if a programmer does not specify the value. Therefore, programmers have to manually adjust it by some special annotations to maximize the sustained performance.

To achieve completely-abstracted programming for GPU computing, a programming framework, named Stream Programming with Runtime Auto-Tuning (*SPRAT*) [32, 44] has proposed in Chapter 2. However, a SPRAT program is neutral to processor architectures and is not optimized for GPU architecture. Thus, it may not exploit the full potential of a

GPU. Thereby, this chapter deals with automatic performance tuning methods for a CUDA program automatically-generated from a SPRAT program.

## 3.3 Optimizing Methods Based on Architectural Features

### 3.3.1 Optimization Methods for Memory Accesses

#### Reusable Data Prefetching

A programmer can use several types of memories in a GPU. The global memory is most commonly used because of its large capacity. However, the latency of access to the global memory is long, and the bandwidth of the global memory tends to be insufficient to exploit the high floating-point operating rate of a GPU. Therefore, to exploit the computing capability of GPUs, it is needed to find reusable data blocks and to store those blocks on the shared memory. By using the shared memory for reusable data, the number of global memory accesses can be reduced. Reading data from the shared memory instead of the global memory can increase the sustained memory bandwidth of a program.

The shared memory has a high bandwidth. However, its capacity is limited such as only 16 or 48KiB. Although there are many reusable data blocks in an application, the shared memory cannot hold all of them due to the capacity shortage. In reusable data blocks, there are data blocks accessed by multiple threads many times that are called *highly-reusable data blocks*. Hence, to efficiently use the shared memory, it is needed to identify highly-reusable data blocks and to preferentially place those blocks on the shared memory. In this section, a *reusable data prefetching* method is proposed to estimate highly-reusable data blocks by analyzing a SPRAT program and to copy those data from the global memory to the shared memory.

In the SPRAT language, reusable data blocks can be found only in `gather` streams because these blocks are accessed multiple times by contiguous threads in the same CTA. An element in `gather` streams can be accessed by neighbor threads. Figure 3.3 shows the memory access by using an absolute indexing operator, and all threads read one element in the input stream. Figure 3.4 shows the memory access by using a relative indexing operator, and each thread reads its corresponding element and neighbor elements in the input stream. In these figures, a box and a cylinder mean an element in streams and a thread of a kernel, respectively. These elements are reusable in multiple threads, and the memory accesses to the same element by multiple threads are redundant. Hence, these redundant memory accesses can be reduced by using the shared memory as a cache memory.

An element in `in`/`out`/`inout` streams, called *sequential access streams*, is related to a particular thread, and only the thread can access the element. Figure 3.5 shows the memory

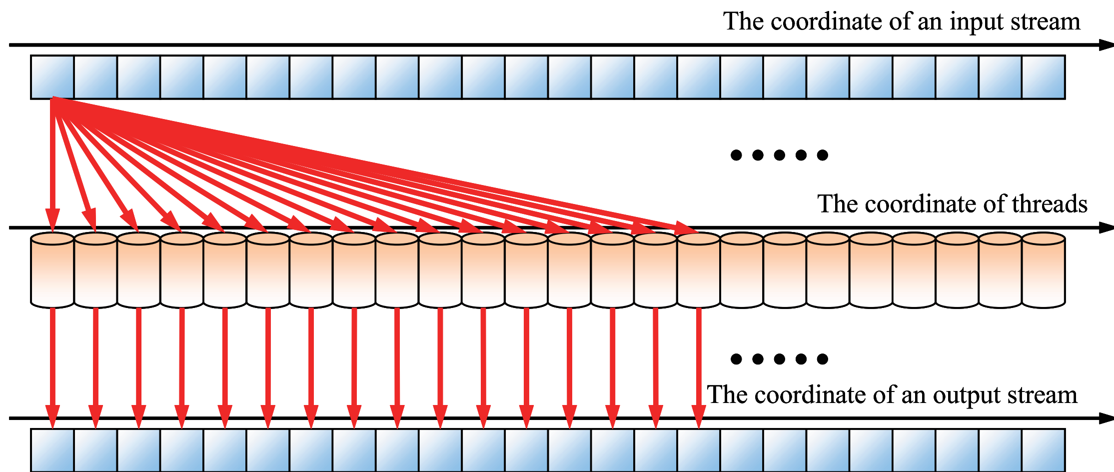



Figure 3.3: The memory access pattern of a `gather` stream by an absolute index.

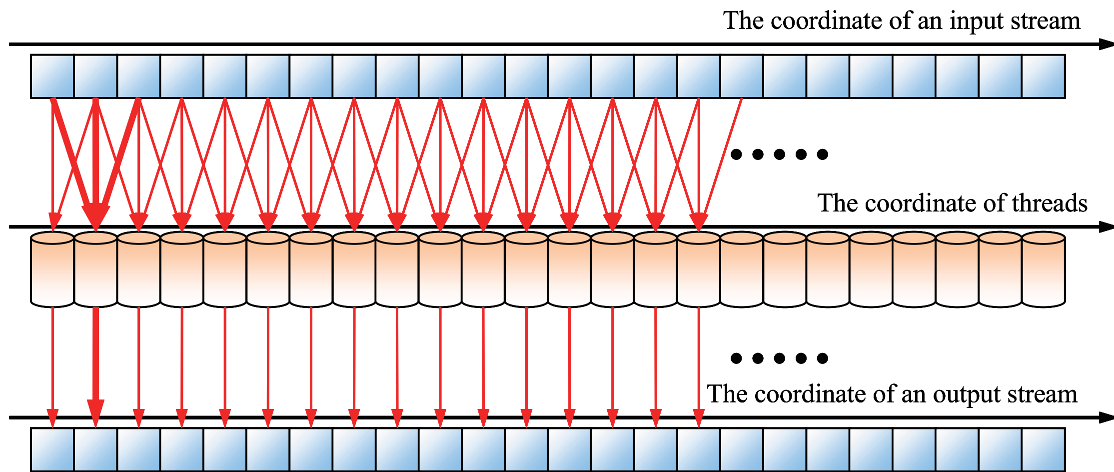



Figure 3.4: The memory access pattern of a `gather` stream by a relative index.

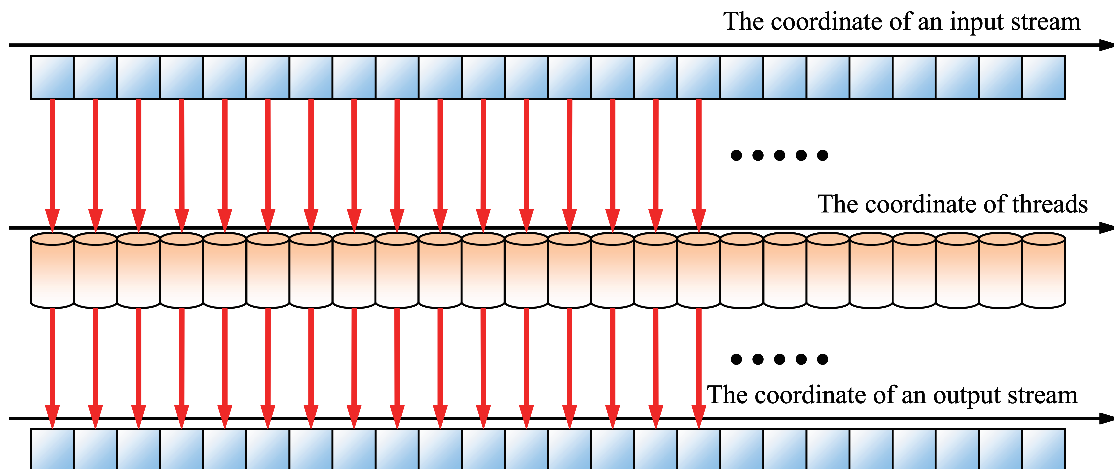



Figure 3.5: The memory access pattern of `in` and `out` streams.

Listing 3.1: The sample code using an absolute indexing operator.

```
1 kernel map func2(gather stream<float> x,
2                  out stream<float> y)
3 {
4     y = x[[0]];
5 }
```

Listing 3.2: The sample code using relative indexing operators.

```
1 kernel map func3(gather stream<float> x,
2                  out stream<float> y)
3 {
4     y = x[-1] + x[0] + x[1];
5 }
```

accesses for sequential access streams, and each thread reads only the corresponding element in the input stream. Therefore, as an element in sequential access streams cannot be accessed by neighbor threads, there is no access to this or the element by multiple threads.

Generally, it is difficult to analyze the pattern of random memory accesses. However, in the SPRAT language, it is relatively easy to analyze memory access patterns because of relative and absolute indexing operators provided by SPRAT. These operators enable programmers to indicate accessed elements by constant values with relative and absolute indexing operators. The SPRAT compiler can statically figure out the access pattern to `gather` streams if programmers use these operators with constant values. As a result, the SPRAT compiler can determine highly-reusable data blocks in `gather` streams and generate a CUDA program in which those blocks are copied to the shared memory in advance of memory accesses.

In the `func2` function shown in Listing 3.1, an absolute indexing operator is used with a constant value. The SPRAT compiler can estimate that all threads read the 0-th element in the `gather` stream `x`, as shown in Figure 3.3. In the case of Listing 3.1, the SPRAT compiler determines that the 0-th element in the `gather` stream `x` is accessed N times, where N is the number of threads in a CTA. Hence, the SPRAT compiler optimizes a program so as to prefetch the 0-th element in the `gather` stream `x` to the shared memory.

In the `func3` function shown in Listing 3.2, three relative indexing operators are used with constant values. The SPRAT compiler can estimate that each element is accessed three times from neighbor threads as shown in Figure 3.4. In this case, it is possible to improve the sustained memory bandwidth by prefetching the elements whose index numbers are between $I_s - 1$ and $I_s + N + 1$, where $I_s$ and N are the index number of the first thread and the CTA

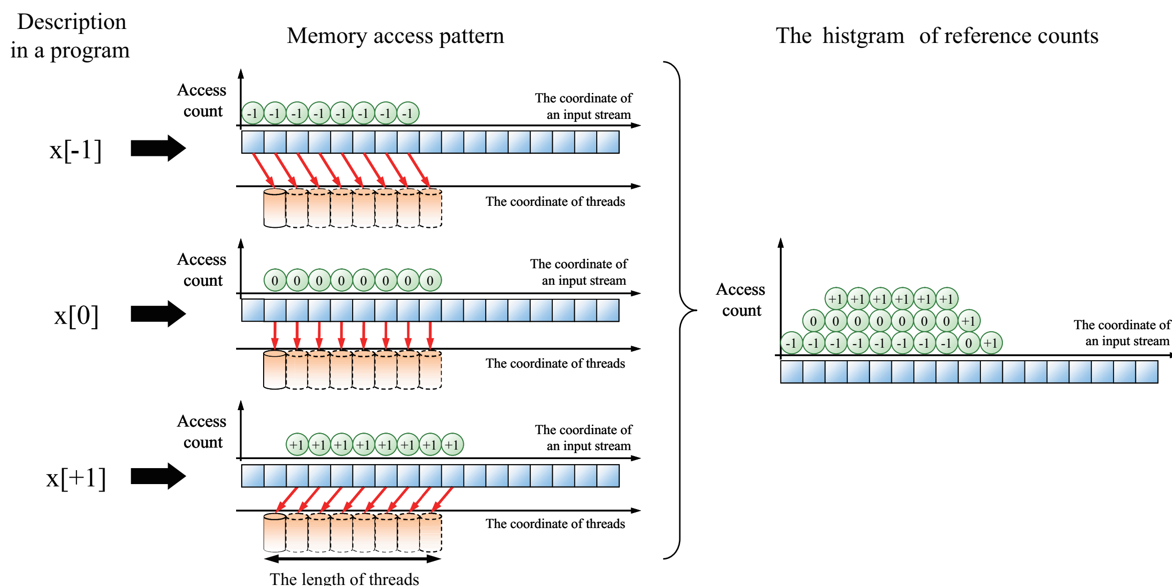



Figure 3.6: Estimation for the histogram of memory access counts.

size, respectively.

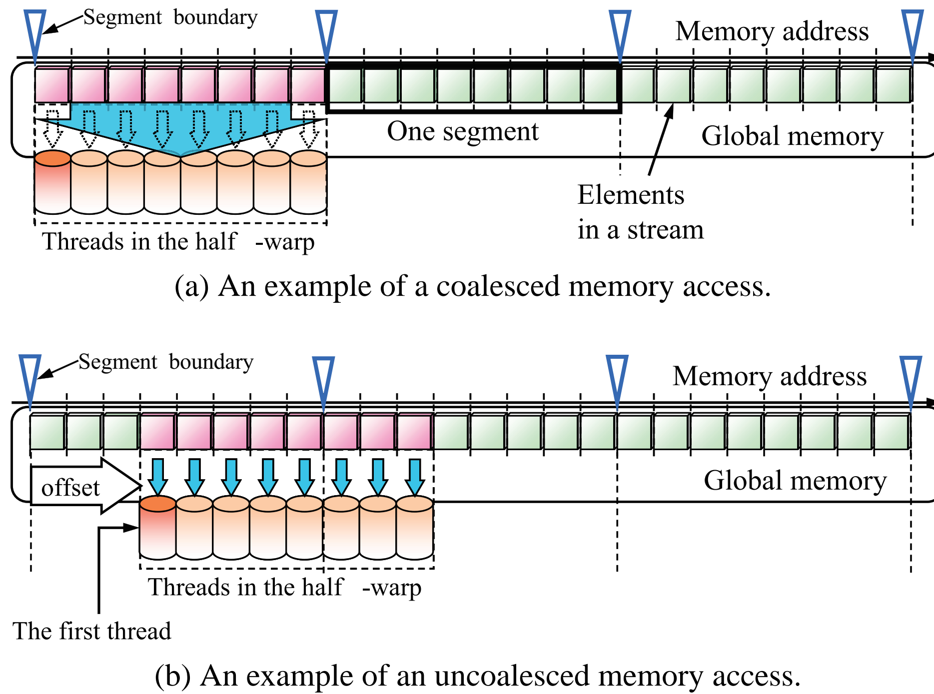
The SPRAT compiler finds absolute and relative indexing operators in a program and estimates the elements accessed by the operators. As the SPRAT compiler knows the CTA size when a program is compiled, the accessed elements and their access counts can be completely estimated. Then, the SPRAT compiler merges the access counts of the elements into one histogram of access counts, as shown in Figure 3.6. The SPRAT compiler can find highly-reusable data blocks from the histogram and optimize a program to copy those data blocks to the shared memory in advance.

### Adjusting Access Patterns

In CUDA, to achieve high memory bandwidths, *memory coalescing* [3] is required when a GPU accesses its global memory. Memory access requests dispatched from threads in the half-warp can be coalesced to one memory operation if the following conditions are met in the case of GeForce 8800 GTX.

(1) The size of the element accessed by the threads must be 4, 8, or 16 bytes.

(2) If the element size is:




(a) An example of a coalesced memory access.

(b) An example of an uncoalesced memory access.

Figure 3.7: Examples of a global memory access in CUDA.

- 4, all 16 elements must lie in the same 64-byte segment,
- 8, all 16 elements must lie in the same 128-byte segment,
- 16, all 8 elements must lie in the same 128-byte segment.

(3) Threads must access the elements in sequence: The k-th thread in the half-warp must access the k-th element.

In CUDA, these coalescing conditions depend on the *compute capability* [3] of a GPU. The coalescing condition (1) can be met if the data type of elements in a stream is `int`, `float`, or `double`. Moreover, in a SPRAT program, the coalescing condition (3) can be naturally met because the stream programming model adopted by SPRAT already assumes this condition except `gather` streams. In `gather` streams, it is difficult to meet the coalescing condition (3) because it cannot be ensured that elements read by threads are continuous. Accordingly, all the conditions are met for sequential access streams in SPRAT if the condition (2) is met by optimizing the access pattern.

Figure 3.7 shows examples of coalesced and uncoalesced memory access patterns in a CUDA program automatically-generated from a SPRAT program. If the first thread in a CTA

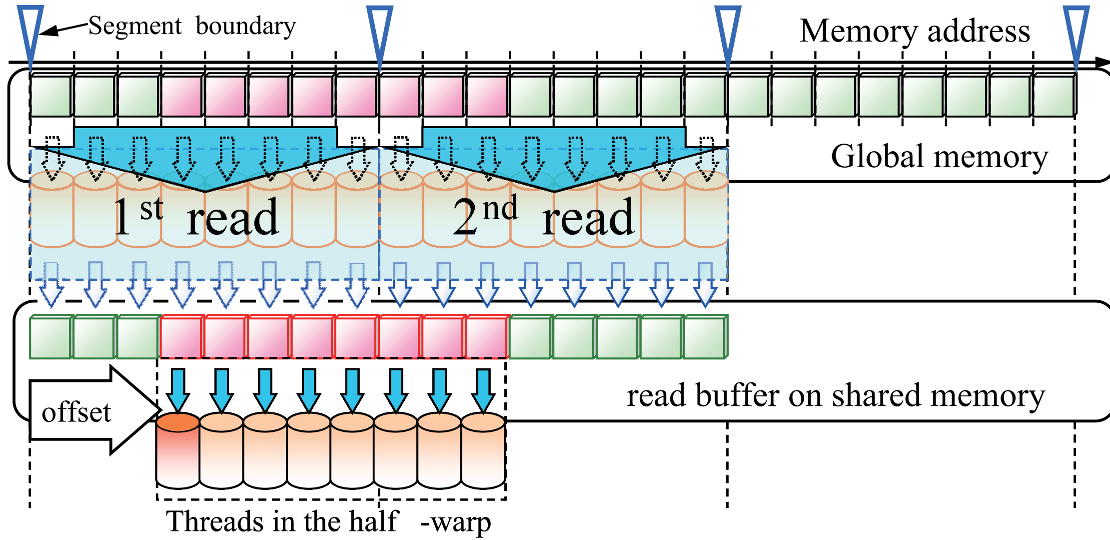



Figure 3.8: The overview of the proposed method to adjust the access pattern.

accesses the element on a segment boundary, as shown in Figure 3.7(a), the other threads in the half-warp also access elements in the same segment, and the coalescing condition (2) is met. However, if the first thread in a CTA accesses the element with a certain offset from a segment boundary, as shown in Figure 3.7(b), the other threads in the half-warp cannot access the elements in the same segment, and the coalescing condition (2) is not satisfied. As a result, this degrades the sustained bandwidth of the kernel.

However, the position of an element read by the first thread is not always on a segment boundary. If it is possible to ensure that the first thread accesses the element on a segment boundary, the memory access pattern can satisfy the coalescing condition (2) and the sustained bandwidth is improved.

To automatically optimize memory access patterns, the SPRAT compiler uses the shared memory as a read buffer. Figure 3.8 shows the overview of the proposed method. The SPRAT compiler adopts a buffering mechanism for each `in` stream in a SPRAT program. In the buffering mechanism, all accesses to elements in an `in` stream are buffered on a read buffer in the shared memory. In this method, one uncoalesced memory access is replaced to two coalesced memory accesses. At the beginning of a kernel code, all threads in a half-warp cooperatively load the data blocks including necessary data by two coalesced memory accesses, and store those blocks to the read buffer. The load operations on the elements in the global memory are changed to the operations on the elements in the read buffer. In this

way, inefficient uncoalesced memory accesses are not used, and the efficiency of memory accesses can be improved.

The proposed method causes redundant memory accesses because neighboring unnecessary data are loaded with necessary data in two coalesced memory accesses. Hence, it is important to check if the proposed method improves the sustained bandwidth. The condition that the proposed method can improve the sustained bandwidth is clarified in the following calculation. The bandwidths of coalesced and uncoalesced memory accesses are assumed $B_{\mathrm{g,coalesced}}$ and $B_{\mathrm{g,uncoalesced}}$, respectively. The sustained memory bandwidth $B_e$ is defined using the *Efficient of Data Usage* $\eta$ as

$$B_{\mathrm{e}} = \eta \cdot B_{\mathrm{g}}, \tag{3.1}$$

and

$$\eta = \frac{D_{\mathrm{necessary}}}{D_{\mathrm{loaded}}}, \tag{3.2}$$

where $D_{\mathrm{necessary}}$ and $D_{\mathrm{loaded}}$ are the size of necessary data and loaded data from the global memory, respectively. Here, the latency of the shared memory is ignored because it can be negligible compared with that of the global memory.

In an unoptimized code, the sustained memory bandwidth of an `in` stream $B_{\mathrm{e,unopt}}$ can be calculated by

$$\eta_{\mathrm{unopt}} \quad = \quad 1, \tag{3.3}$$

and

$$B_{\mathrm{e,unopt}} \quad = \quad \eta_{\mathrm{unopt}} \cdot B_{\mathrm{g,uncoalesced}} = B_{\mathrm{g,uncoalesced}}. \tag{3.4}$$

On the other hand, in an optimized code, the sustained memory bandwidth of an `in` stream $B_{\mathrm{e,opt}}$ can be calculated by

$$B_{\mathrm{e,opt}} \quad = \quad \eta_{\mathrm{opt}} \cdot B_{\mathrm{g,coalesced}}. \tag{3.5}$$

Table 3.2: The bandwidths ratios between coalesced and uncoalesced memory accesses for several GPUs.

| GPU name | $B_{\text{g,coalesced}}$ | $B_{\text{g,uncoalesced}}$ | $\frac{B_{\text{g,coalesced}}}{B_{\text{g,uncoalesced}}}$ |
|---|---|---|---|
| GeForce 8600 GTS | 20.0 GB/s | 2.9 GB/s | 6.84 |
| GeForce 8800 GT | 46.0 GB/s | 4.9 GB/s | 9.38 |
| GeForce 8800 GTX | 66.2 GB/s | 6.8 GB/s | 9.73 |
| GeForce GTX 280 | 112.2 GB/s | 61.6 GB/s | 1.83 |

Therefore, the condition that the optimized code can outperform the unoptimized code is shown as

$$\frac{B_{\text{e,opt}}}{B_{\text{e,unopt}}} = \frac{\eta_{\text{opt}} B_{\text{g,coalesced}}}{B_{\text{g,uncoalesced}}} \quad > \quad 1, \tag{3.6}$$

therefore

$$\frac{B_{\text{g,coalesced}}}{B_{\text{g,uncoalesced}}} \quad > \quad \frac{1}{\eta_{\text{opt}}}. \tag{3.7}$$

In the proposed method, the size of loaded data is the double size of necessary data because two redundant memory accesses are needed to load necessary data. Hence, $\eta_{\text{opt}}$ can be calculated as

$$\eta_{\text{opt}} = \frac{D_{\text{opt,necessary}}}{D_{\text{opt,loaded}}} = \frac{1}{2}. \tag{3.8}$$

Table 3.2 indicates the ratios of the coalesced memory access bandwidth to the uncoalesced one. This table shows that the bandwidth ratio depends on the GPU, and the proposed method is not always effective. The condition for the proposed method to improve the bandwidth is given by

$$\frac{B_{\text{g,coalesced}}}{B_{\text{g,uncoalesced}}} > \frac{1}{\eta_{\text{opt}}} = 2. \tag{3.9}$$

In Table 3.2, GeForce 8600 GTS, GeForce 8800 GT, and GeForce 8800 GTX meet the condition of Equation (3.9). However, GeForce GTX 280 does not meet that condition. Hence, it is better to adopt the proposed method only if the GPU meets the condition.

The SPRAT framework runs a simple benchmark in advance to measure the bandwidths; $B_{\mathrm{e,coalesced}}$ and $B_{\mathrm{e,uncoalesced}}$, and can determine whether a compiler should apply this optimization or not. The SPRAT compiler also has a compiler option to disable this optimization.

### 3.3.2 Automatic Performance Tuning of the CTA configuration

In the CUDA language, a programmer must determine execution parameters that indicate the number of threads in a three-dimensional CTA and the number of CTAs in a two-dimensional grid. When a kernel is invoked, each CTA in a grid is assigned to a MP. There is the upper bound for the number of CTAs per MP determined by the GPU architecture. The number of CTAs running on a MP is also restricted when the total amount of hardware resource such as registers and the shared memory exceeds the given hardware resources of a MP. A decrease in the number of threads running on a MP causes the inefficiency of switch-on-event multithreading. Consequently, a programmer must determine the execution parameters considering the hardware resources required by a CTA.

For tuning the execution parameters, NVIDIA defines an indicator called *occupancy* [45], which is the ratio of the number of actually-assigned warps to the number of architectural maximum warps on a MP. A higher occupancy results in a higher computing performance. Although the occupancy depends on the amount of hardware resource used by a CTA, the amount of hardware resource drastically depends on CUDA codes and their optimization levels.

If a programmer stores too many reusable data on the shared memory, the size of reusable data might exceed the shared memory capacity. Due to the shortage of the shared memory, the occupancy decreases, resulting in performance degradation. However, the occupancy is not always an absolute indicator to find the optimal execution parameters. If a programmer applies the reusable data prefetching, the occupancy decreases even though the prefetched data are provided to the MP at a short latency and hence the computing performance is generally improved.

The CTA configuration consists of its shape and size, which are the number of threads in each dimension and the total number of threads in a CTA, respectively. The CTA shape is as important as the CTA size to achieve a high sustained performance. A programmer can independently indicate the number of threads in each dimension. The gird size is generally calculated by the size of processing data with the CTA configuration.

Listing 3.3: The code of the saxpy kernel.

```
kernel map saxpy(
      float a,
      in stream<float> x,
      in stream<float> y,
      out stream<float> z)
{
   z = a * x + y;
   return;
}
```

### The Space of Parameter Exploration

In the SPRAT framework, a programmer does not need to consider execution parameters because the SPRAT compiler automatically decides the grid size and the CTA configuration. The SPRAT compiler always defines the default configuration of a CTA as $16 \times 16 \times 1$, and the grid size is calculated with this CTA configuration. The CTA configuration predefined by the SPRAT compiler is empirically decided and is adequate in many cases. However, it is obvious that there is room to further improve the performance of the codes automatically generated by the SPRAT compiler.

To explore an optimal parameter configuration, this section presents an automatic parameter tuning mechanism based on performance profiling with various parameter combinations. As it is impractical to explore the whole tuning space, the proposed mechanism limits the exploration space using the following conditions.

1. **Reducing the exploration space based on the characteristics of GPU hardware.**

   There is an explicit relationship between sustained performance and the CTA configuration. Figure 3.9 shows the sustained memory bandwidth obtained by changing the CTA configuration of a simple data copy kernel. Figure 3.10 shows the sustained performance of a `saxpy` kernel shown in Listing 3.3. In these figures, the two axes indicate the numbers of threads in two dimensions, $CTAx$ and $CTAy$, respectively. These results clearly indicate that the performance of each kernel becomes high when $CTAx$ is a multiple number of 16. This is because of an architectural reason as follows. When a CTA is executed on a MP, the CTA is decomposed into warps. The threads in a half-warp dispatch memory access requests to the global memory at the same time. These requests can be coalesced if the coalesced conditions are met. Hence, the efficiency of memory accesses is improved when $CTAx$ is only a multiple of 16.

   Therefore, the proposed mechanism assumes that $CTAx$ is a multiple of 16 in order to

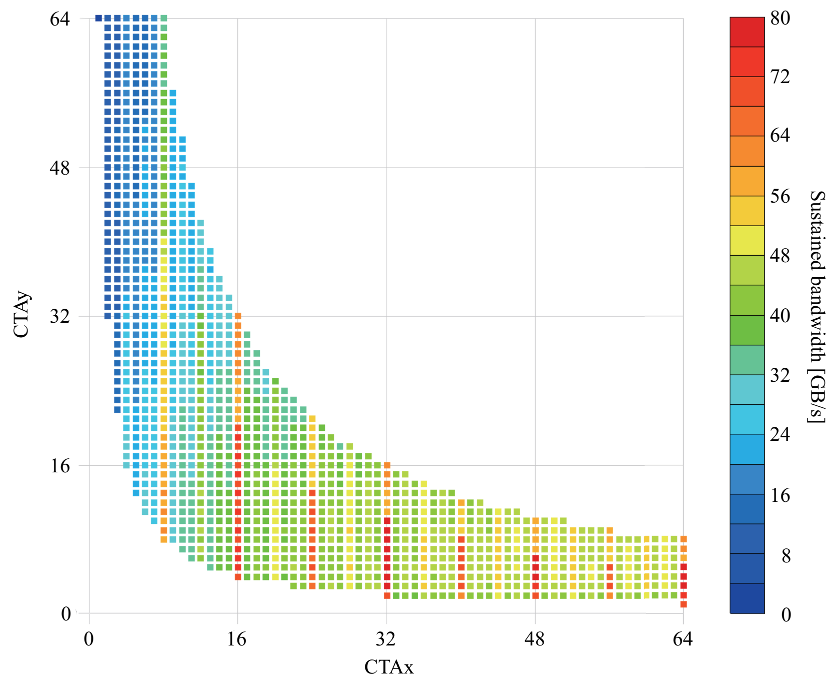



Figure 3.9: Relationship between sustained bandwidths and CTA configurations on the copy kernel.

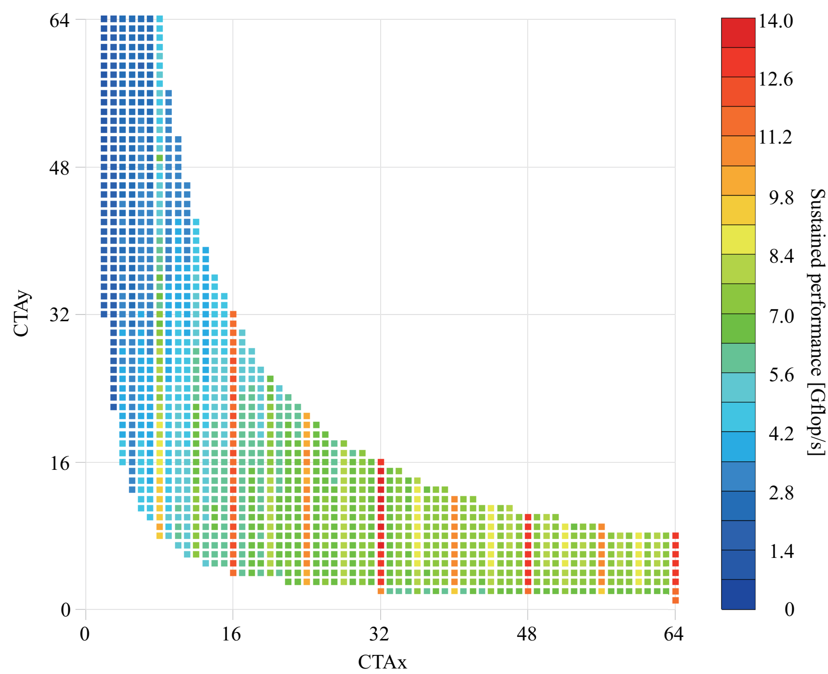



Figure 3.10: Relationship between sustained computing performances and CTA configurations on the saxpy kernel.

limit the exploration space.

2. **Reducing the exploration space based on the CUDA specification.**

   The CUDA specification defines the lower bound and the upper bound for the number of threads per CTA; the lower bound is 64 threads and the upper bound is 512 threads for a GPU whose compute capability is 1.x. Therefore, the proposed mechanism searches the CTA configuration between those bounds.

In addition, the number of threads in the z dimension cannot exceed 64 and is not flexible compared to those in the other dimensions. Hence, $CTAz$ is fixed to 1 and the other dimensions are tuned.

The parameter exploration space in the proposed mechanism is summarized as follows

$$CTAx = 16N \text{ ($N$ is an integer)}, \tag{3.10}$$

$$64 \leqq CTAx \cdot CTAy \cdot CTAz \leqq 512, \tag{3.11}$$

and

$$CTAz = 1. \tag{3.12}$$

The grid size is calculated by

$$gridDim.x = ceil\left(\frac{X}{CTAx}\right), \tag{3.13}$$

and

$$gridDim.y = ceil\left(\frac{Y}{CTAy}\right), \tag{3.14}$$

where X and Y are the width and height of an `output` stream to be computed by stream processing in the kernel, respectively.

#### Auto-Tuning Mechanism

In the SPRAT compiler, the profiling system and the profile data analyzer work together to find an appropriate CTA configuration. Algorithm 1 shows a pseudo code of the automatic

**Algorithm 1** Pseudo code of automatic tuning.

1: $CTAz = 1$
2: **for** $CTAy = 1$ to $512$ **do**
3:    **for** $CTAx = 1$ to $512$ **do**
4:       **if** $(CTAx\ mod\ 16 = 0)$ and $(64 \leqq CTAx \cdot CTAy \cdot CTAz \leqq 512)$ **then**
5:          Compile the program with ($CTAx$, $CTAy$, $CTAz$).
6:          Profile the program with sample data.
7:       **end if**
8:    **end for**
9: **end for**
10: Analyze profiling data and output the optimal CTA configuration.
11: Compile with the optimal CTA configuration; ($CTAx$, $CTAy$, $CTAz$).

tuning to explore the optimal CTA configuration under the reduced exploration space. Once a programmer specifies a kernel name and input data, the mechanism automatically generates kernels with various CTA configurations (Line 5 in Algorithm 1), profiles their performances (Line 6), and looks for an optimal parameter configuration (Line 10). Here, $CTAx$ and $CTAy$ are changed from 1 to 512.

In data-parallel processing, the execution time of a kernel is in proportion to its data size in many cases. Thus, the linear performance prediction model made from small-size data sets can be used to predict the execution time of an arbitrary data size. The proposed mechanism can find the optimal CTA configuration by using the small-size data sets in a much shorter time than the profiling time with large data sets. The validation of this strategy to reduce the tuning time is evaluated in Section 3.4.3.

To select an appropriate CTA configuration, the performance of the kernel for the data size that is not given by a programmer must be predicted. The proposed mechanism firstly obtains the performance data from several execution times using various data sizes and generates a linear performance prediction model based on the least square approximation [46], as shown in Figure 3.11. Thus, the performance of a kernel for an arbitrary data size is predicted by the slope and the intercept. In the proposed mechanism, these parameters such as the slope and the intercept of a prediction model are calculated for each CTA configuration. The optimal CTA configuration can be automatically found in the candidates of CTA configurations by analyzing these parameters.

As the angle of a slope is dominant in predicted execution times for large data, the proposed mechanism selects the CTA configuration with the minimum slope as the optimal one. For example, in Figure 3.12, three prediction models are obtained. In this case, the

CTA configuration 3 is the optimal configuration because its slope is the smallest in all the CTA configurations.

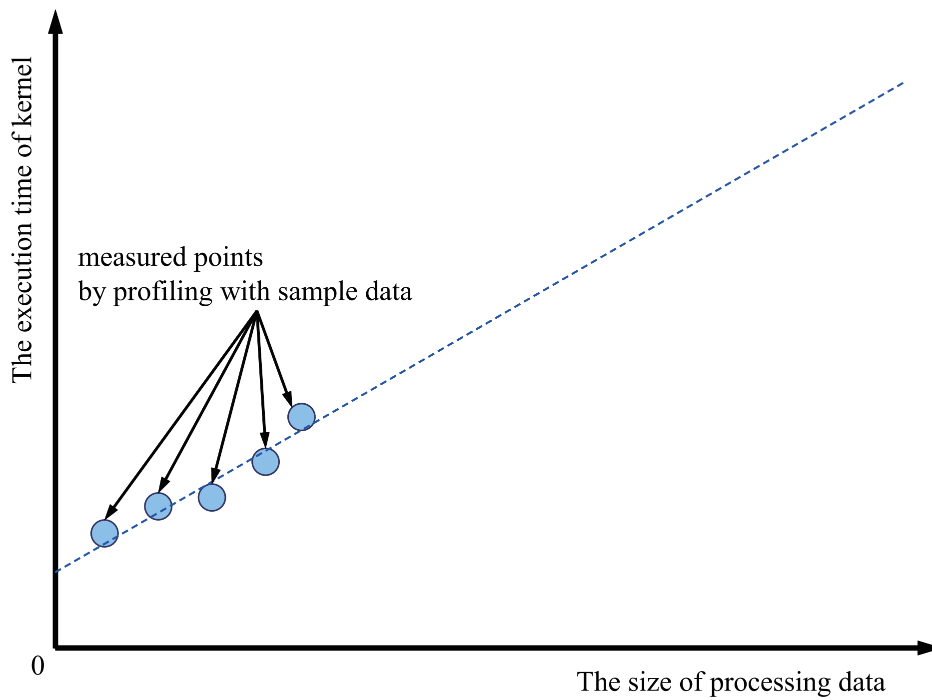



Figure 3.11: The method of building a prediction model by profiling.

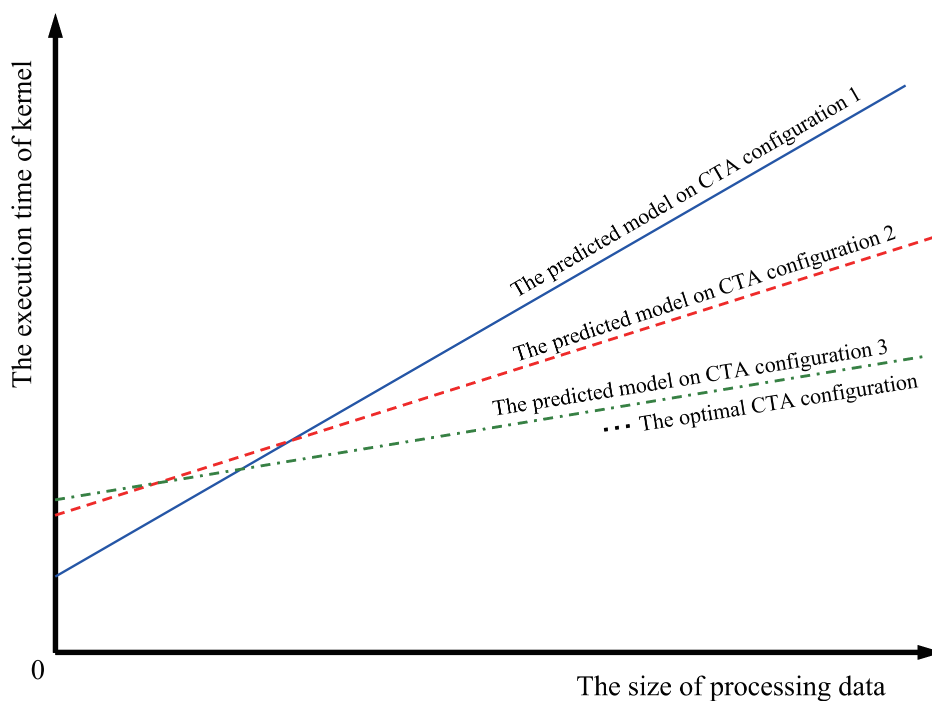



Figure 3.12: The policy of selecting the optimal CTA configuration.

## 3.4 Evaluation

### 3.4.1 Benchmarks for Evaluations

In the evaluation, two benchmark programs are used. One is the Himeno benchmark that is a three-dimensional single-precision Jacobi kernel with various sizes [39]. Listing 3.4 shows the main kernel of the Himeno benchmark implemented by the SPRAT language. This kernel is a typical stencil processing code that updates every element of an array by using its neighboring elements. Each element in `gather` stream `p` is accessed 18 times in one iteration of the kernel. The SPRAT compiler detects those reusable data blocks and generates a code that prefetches elements in the stream to the shared memory. Since the SPRAT compiler converts a three-dimensional array into a two-dimensional array, it optimizes the kernel code to reuse each fetched element 9 times.

The other benchmark is the LU decomposition that is often used in scientific applications. The LU decomposition consists of two kernels. Listing 3.5 shows the kernels of the LU decomposition implemented by the SPRAT language. As the start address of data blocks passed to the kernel varies in each iteration, many uncoalesced memory accesses occur in executing the LU decomposition. In this case, adjusting the access pattern is effective to improve the sustained performance. In these kernels, as the execution time of the `rowop` kernel is dominant, only the `rowop` kernel is used for evaluation.

### 3.4.2 Evaluation of Optimization Methods for Memory Accesses

To demonstrate the effect of the optimization methods for memory accesses proposed in Section 3.3.1, two benchmarks are evaluated under the experimental setup shown in Table 3.3.

To clarify the performance improvement by reusable data prefetching, the evaluation using the Himeno benchmark is conducted. The problem size of the Himeno benchmark is set to `MIDDLE`($256 \times 128 \times 128$). In this evaluation, the performances of six programs are compared; CUDA(baseline), CUDA(xy-reuse), CUDA(no-reuse), SPRAT(baseline), SPRAT(auto), and SPRAT(auto+manual).

The CUDA(baseline) implementation is a hand-tuned code by an expert programmer. It efficiently uses the shared memory for highly-reusable data blocks. The CUDA(xy-reuse) implementation is also a hand-tuned code without considering data reusability of the array `p` for the z-direction. This is the same condition of a CUDA code automatically-generated by

Listing 3.4: The main kernel of the Himeno benchmark implemented by SPRAT.

```
kernel map jacobi(
      gather stream<float> p,
      in stream<float> a0, in stream<float> a1,
      in stream<float> a2, in stream<float> a3,
      in stream<float> b0, in stream<float> b1,
      in stream<float> b2, in stream<float> c0,
      in stream<float> c1, in stream<float> c2,
      in stream<float> bnd, in stream<float> wrk1,
      out stream<float> wrk2, out stream<float> gosa)
{
   float ss, s0;
   s0 =
     + a0 * p[+1][ 0][ 0]
     + a1 * p[ 0][+1][ 0]
     + a2 * p[ 0][ 0][+1]
     + b0 * ( p[+1][+1][ 0] - p[+1][-1][ 0]
          - p[-1][+1][ 0] + p[-1][-1][ 0] )
     + b1 * ( p[ 0][+1][+1] - p[ 0][-1][+1]
          - p[ 0][+1][-1] + p[ 0][-1][-1] )
     + b2 * ( p[+1][ 0][+1] - p[-1][ 0][+1]
          - p[+1][ 0][-1] + p[-1][ 0][-1] )
     + c0 * p[-1][ 0][ 0]
     + c1 * p[ 0][-1][ 0]
     + c2 * p[ 0][ 0][-1]
     + wrk1;
   ss = ( s0 * a3 - p[0][0][0] ) * bnd;
   gosa = ss * ss;
   wrk2 = p[0][0][0] + 0.8f * ss;
   return;
}
```

Listing 3.5: The main kernel of the LU decomposition implemented by SPRAT.

```
kernel map rowop(inout stream<float> x, gather stream<float> y)
{
   x -= y[[-1]][0] * y[0][[-1]];
   return;
}

kernel map normalize(out stream<float> x, gather stream<float> y, )
{
   x = y[0][0] / y[[-1]][0];
   return;
}
```

the SPRAT compiler with reusable data prefetching. The CUDA(no-reuse) implementation is a simple code without considering any data reusabillty in the array `p`.

The SPRAT(baseline) implementation is a simple CUDA code that is not optimized and is translated directly from a SPRAT program. The SPRAT(auto) implementation is an automatically-optimized CUDA code with reusable data prefetching. The SPRAT(auto+manual) implementation is also an automatically-optimized CUDA code, but memory accesses to non-reusable sequential access streams are adjusted by hand to meet the coalesced conditions. In all implementations, the CTA configuration is fixed to $16 \times 16 \times 1$.

Figure 3.13 shows the sustained performance of the Himeno benchmark on GeForce 8800 GTX and GeForce GTX 280. In this figure, the performance is measured in floating-point operations per second. The evaluation results clearly indicate that reusable data prefetching improves the performance of an automatically-generated CUDA program on both GeForce 8800 GTX and GeForce GTX 280. Especially, the performance improvement on GeForce 8800 GTX is 2.08 times of the SPRAT(baseline) implementation. As the bandwidth degradation of an uncoalesced memory access on GeForce 8800 GTX is larger than that on GeForce GTX 280, reusable data prefetching is effective on GeForce 8800 GTX. On the other hand, the performance improvement by reusable data prefetching is not so large on GeForce GTX 280.

The CUDA(baseline) implementation outperforms all SPRAT implementations because it exploits data reusability of the array even in the z-direction. The SPRAT compiler does not consider data reusability in the z-direction because of its specification. Hence, the maximum performance of the SPRAT implementations is expected to the performance of the CUDA(xy-reuse) implementation.

Table 3.3: Experimental setup in Section 3.4.2.

| Components | Specifications |
|---|---|
| CPU | Intel Core 2 Quad Q6600 2.4GHz |
| Main Memory | DDR2-800MHz 4 Gbytes |
| GPU | NVIDIA GeForce 8800 GTX, GeForce GTX 280 |
| OS | Linux 2.6.18 x86_64 |
| Compiler | gcc version 4.1.2 with “-O2” option |
| Video driver | NVIDIA driver version 180.44 |
| CUDA | version 2.0 |

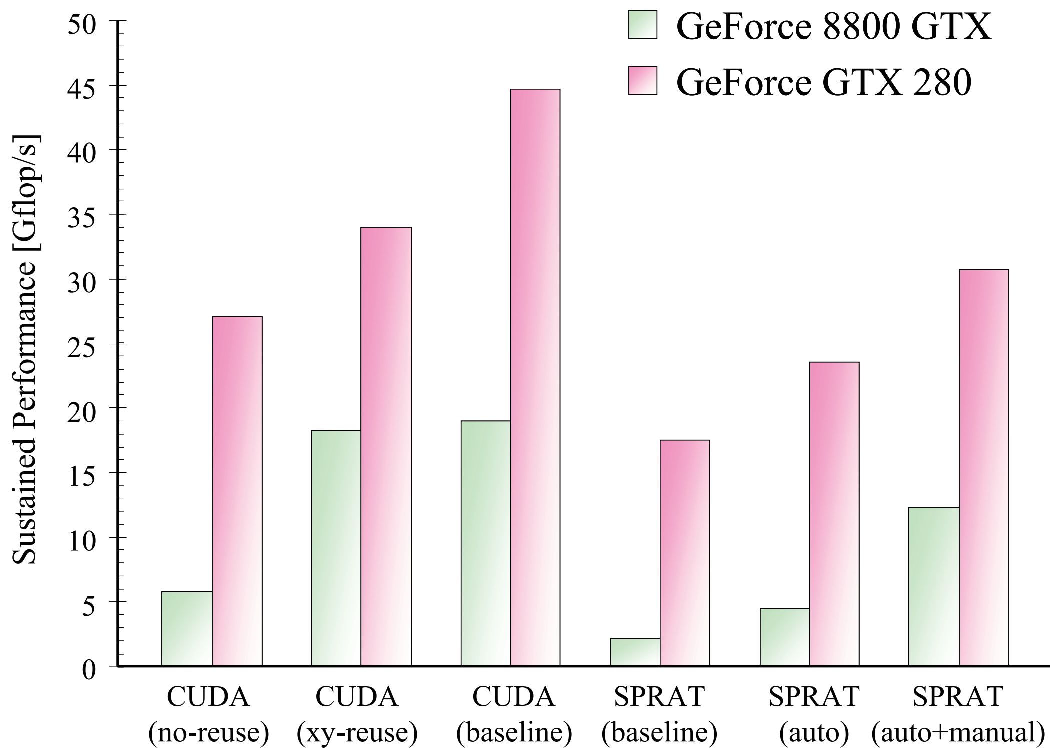



Figure 3.13: Evaluation results with the Himeno benchmark.

However, the performance of the SPRAT(auto) implementation is less than that of the CUDA(xy-reuse) implementation because of uncoalesced memory accesses to `in` streams. A simple implementation of the Himeno benchmark cannot meet the coalesced conditions, and many uncoalesced memory accesses cause the serious performance degradation, especially on GeForce 8800 GTX. Hence, the SPRAT(auto+manual) implementation that manually adjusts memory accesses to `in` streams achieves higher performance than other SPRAT implementations. The performance difference between the SPRAT(auto+manual) implementation and the CUDA(xy-reuse) one is caused by the memory accesses to an `out` stream. In CUDA(xy-reuse) implementation, memory accesses to an `out` stream are manually adjusted.

The effects of two memory access optimizations can be evaluated with the LU decomposition. Figures 3.14 and 3.15 show the sustained performance in several matrix sizes on GeForce 8800 GTX and GeForce GTX 280, respectively. In these figures, the horizontal and vertical axises indicate the sizes of square matrices and the sustained performances of the corresponding matrix sizes, respectively.

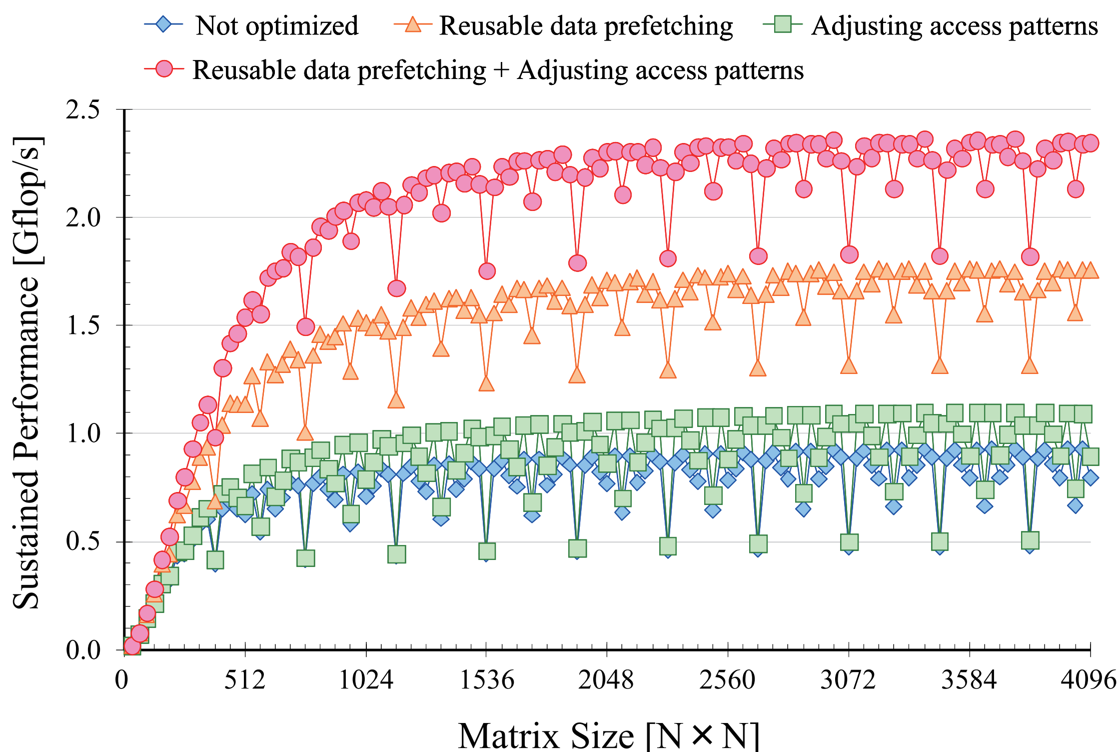



Figure 3.14: Evaluation results with the LU decomposition. (GeForce 8800 GTX)

From the evaluation results shown in Figure 3.14, it is obvious that two memory access optimizations can improve the sustained performance by 2.95 times compared to an unoptimized code on GeForce 8800 GTX when the matrix size is $4096 \times 4096$. In the kernel of the LU decomposition, there is a high reusability in a $y$ stream because the $y$ stream is accessed by using absolute indexing operators and memory accesses from threads are concentrated to a few elements in that stream. Hence, reusable data prefetching is effective to improve sustained bandwidths of the kernel in the LU decomposition. Moreover, as the bandwidth ratio between coalesced and uncoalesced memory accesses is large, adjusting the access pattern is also effective on GeForce 8800 GTX.

On the other hand, adjusting the access pattern degrades the sustained performance on GeForce GTX 280 because the bandwidth ratio between coalesced and uncoalesced memory accesses on GeForce GTX 280 is smaller than the threshold ratio required by the proposed method to work well. For those GPUs, adjusting the access pattern should be disabled to avoid performance degradations.

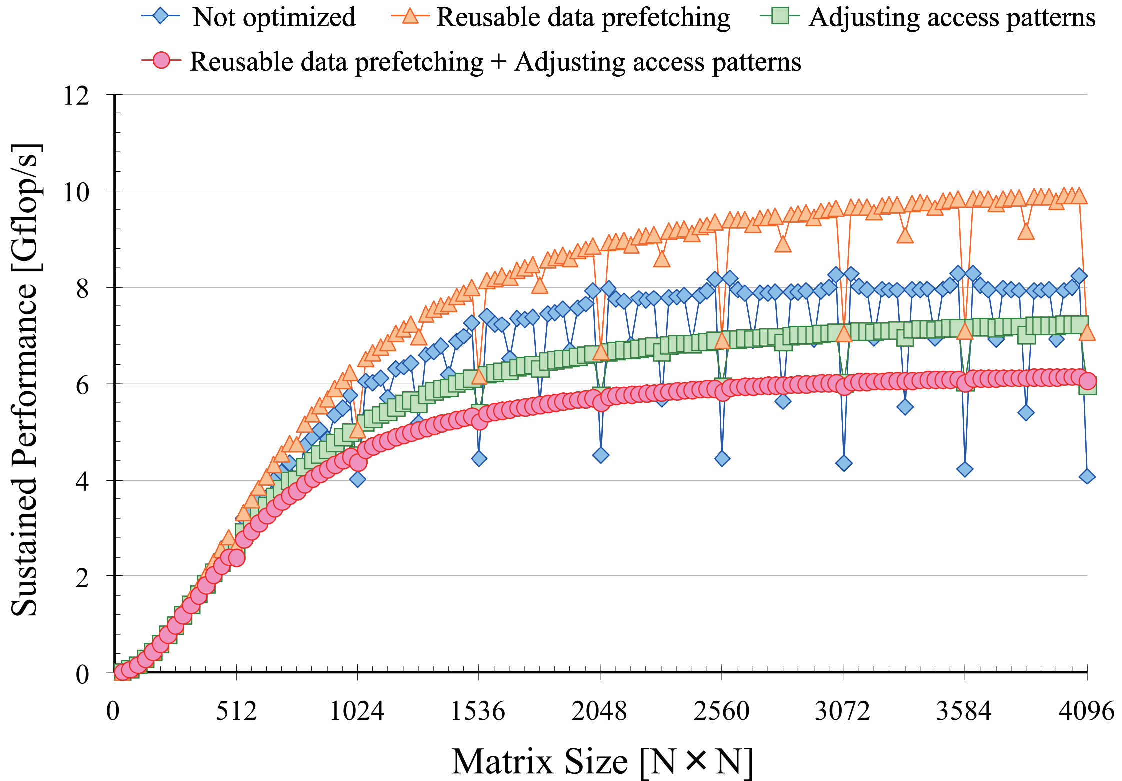



Figure 3.15: Evaluation results with the LU decomposition. (GeForce GTX 280)

### 3.4.3 Evaluation of the CTA Configuration Tuning

To clarify the effect of automatic performance tuning of the CTA configuration, this section evaluates the proposed tuning method with two benchmarks. The experimental setup of this evaluation is shown in Table 3.4.

The evaluation with the Himeno benchmark uses five input data sets for profiling and one input data set for performance evaluation. The grid sizes of the five data sets used in profiling are $64 \times 32 \times 32$, $128 \times 64 \times 64$, $192 \times 96 \times 96$, $256 \times 128 \times 128$ and $320 \times 160 \times 160$.

The proposed tuning mechanism automatically generates CUDA codes with various CTA configurations and carries out performance profiling with the five data sets. Figure 3.16 shows the performance model of each CTA configuration. In this figure, "Automatically Optimized" and "Not Optimized" indicate the predicted performance with and without reusable data prefetching optimization, respectively. "Selected CTA configuration" is the selected one by the proposed mechanism. Using these performance models, the proposed mechanism decides the optimal CTA configuration as $(CTAx, CTAy) = (16, 6)$ whose slope of

the performance model is the smallest.

Next, the performance of the CTA configuration determined by the proposed tuning mechanism is evaluated with a test data set which is not used in profiling. The grid size of the test data set is $512 \times 256 \times 256$ that is larger than the profile data sets. For the test data set, Figures 3.17 and 3.18 show the predicted and actual execution times of each CTA configuration, respectively. In the case of $CTAy = 1$, as the number of gridDim.y of the code exceeds its upper bound, the code cannot be executed on a GPU. The execution times of such CTA configurations are not shown in the Figure 3.18. By the same reason, the experimental results in the case of $CTAy = 1$ are not shown in the Figures 3.19 and 3.20. However, the performance of such a parameter configuration is always low and hence this is not a problem in practical uses.

These execution times in these figures are normalized by the minimum one. The maximum difference and the average difference are 0.411 and 0.079 in the normalized execution times, respectively. The correlation coefficient between the predicted and actual execution times is 0.969. The correlation coefficient between the predicted and actual rankings is 0.821. These two distributions explicitly show the correlation between the predicted and actual execution times.

In Figure 3.19, all the CTA configurations are sorted in ascending order of the actual execution time. The actual execution time of the selected CTA configuration is marked with a circle. These results clearly indicate that the selected CTA configuration can achieve high performance for the test data set. The selected CTA configuration is a suboptimal CTA configuration whose performance is almost the same as the optimal one. The difference in execution time between the automatically-selected CTA configuration and the optimal one

Table 3.4: Experimental setup in Section 3.4.3.

| Components | Specifications |
|---|---|
| CPU | Intel Core i7 920 2.66GHz |
| Main Memory | DDR3-1066MHz 12 Gbytes |
| GPU | NVIDIA Tesla C1060 |
| OS | Linux 2.6.18 x86_64 |
| Compiler | gcc version 4.4.0 with “-O2” option |
| Video driver | NVIDIA driver version 190.29 |
| CUDA | version 2.3 |

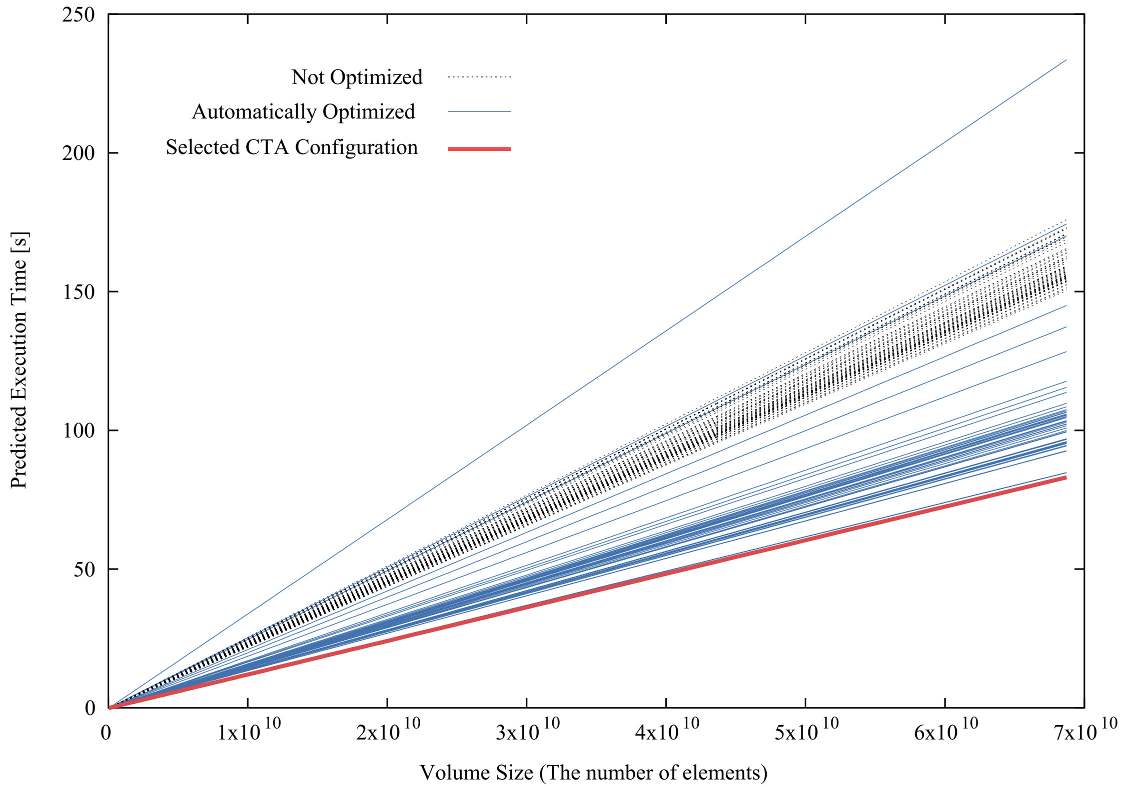



Figure 3.16: The performance model of each CTA configuration on the Himeno benchmark.

is less than 1%.

Figure 3.20 shows the correlation between the predicted and the actual rankings of CTA configurations. Each circle in Figure 3.20 corresponds to one CTA configuration. Its two coordinates are the predicted ranking and the actual ranking of the CTA configuration in execution time, respectively. This figure illustrates that there is a strong correlation between the predicted and the actual rankings, resulting in the accurate prediction of an appropriate CTA configuration for the Himeno benchmark.

In the evaluation with the LU decomposition kernels, five matrices and one matrix are used for profiling and for performance evaluation, respectively. The sizes of five matrices used in profiling are $16 \times 16$, $32 \times 32$, $64 \times 64$, $128 \times 128$ and $256 \times 256$. The size of a matrix used for evaluation is $4096 \times 4096$.

Figure 3.21 shows the performance model of each CTA configuration constructed from the profile data. The CTA configuration selected by the proposed mechanism is $(CTAx, CTAy) = (32, 9)$ because its slope is the smallest. For the test matrix, Figures 3.22 and 3.23 show the

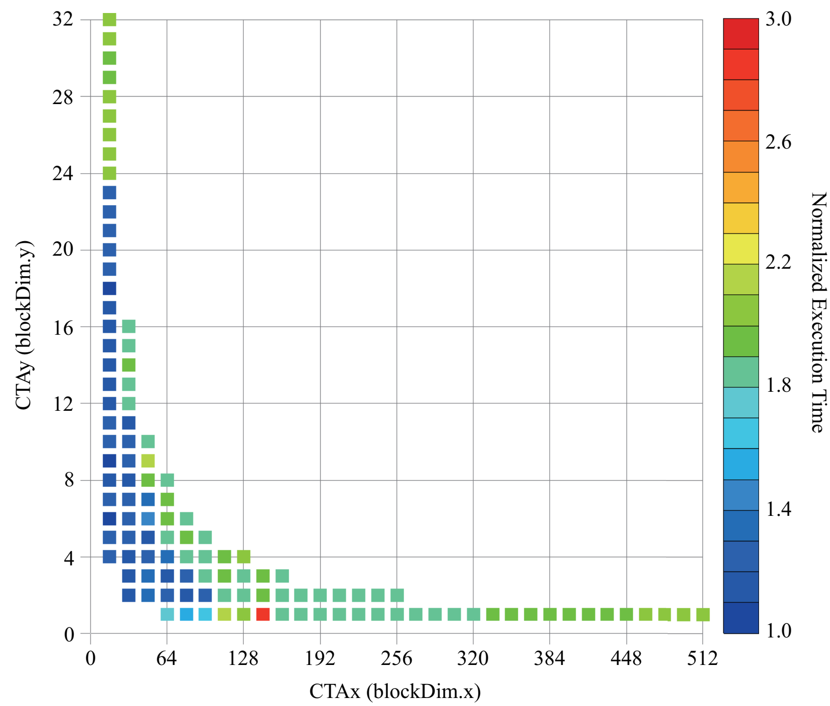



Figure 3.17: The predicted execution time of each CTA configuration on the Himeno benchmark.

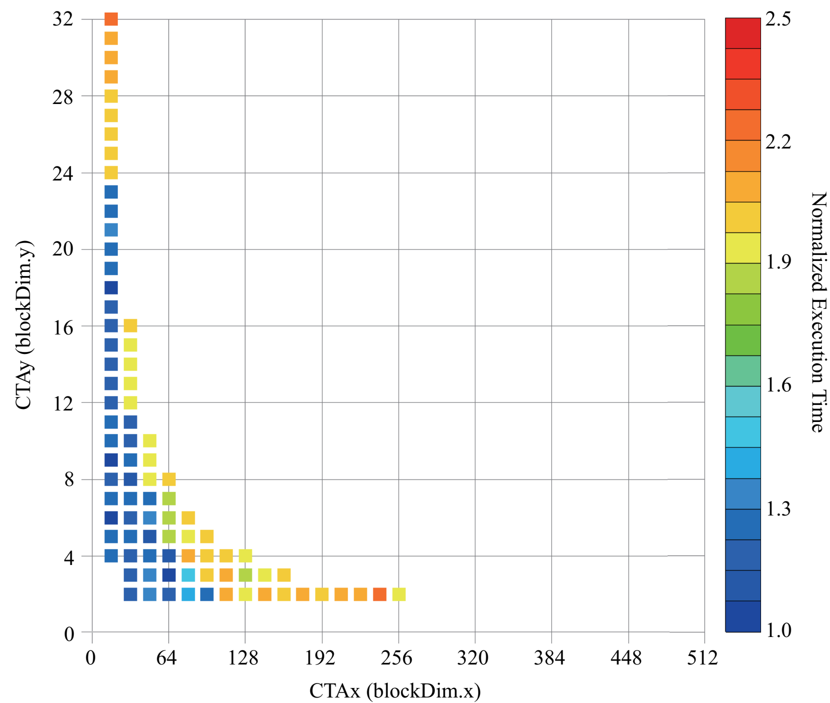



Figure 3.18: The actual execution time of each CTA configuration on the Himeno benchmark.

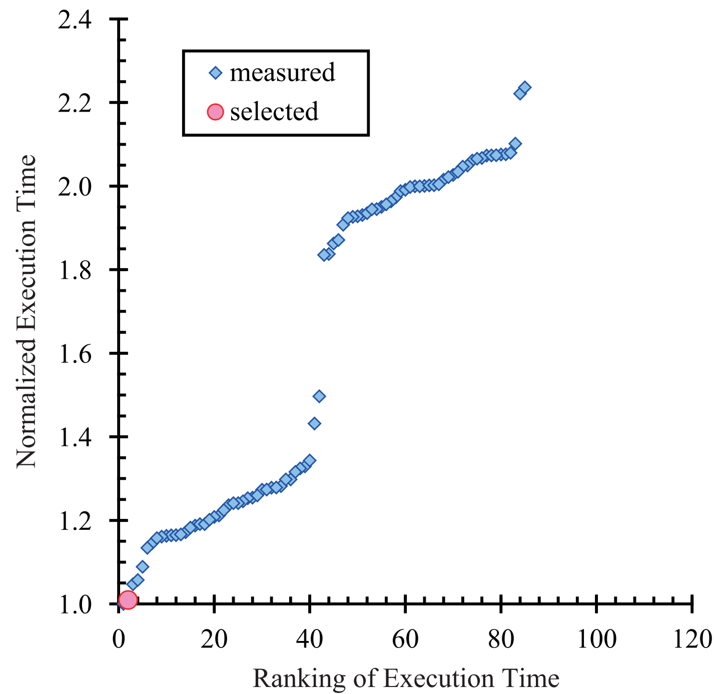



Figure 3.19: The ranking of actual execution time on the Himeno benchmark.

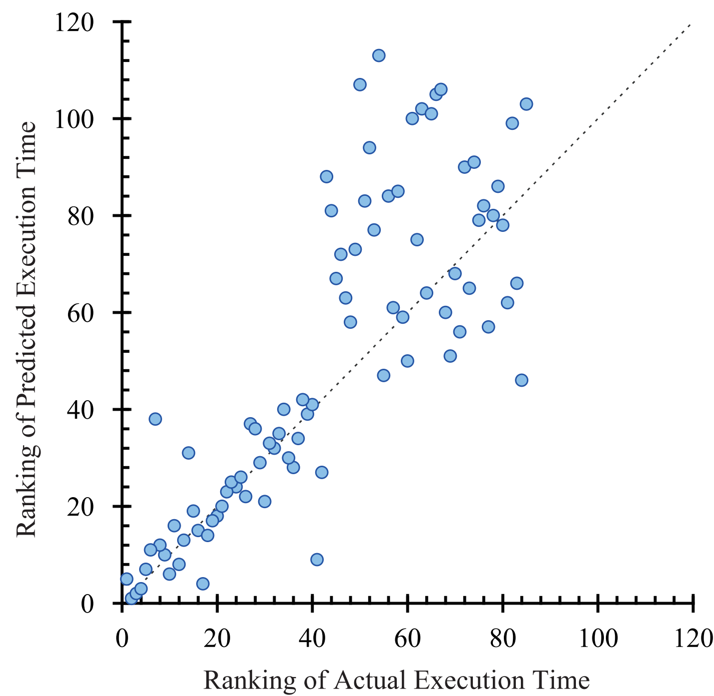



Figure 3.20: The correlation between the predicted and the actual rankings of CTA configurations on the Himeno benchmark.

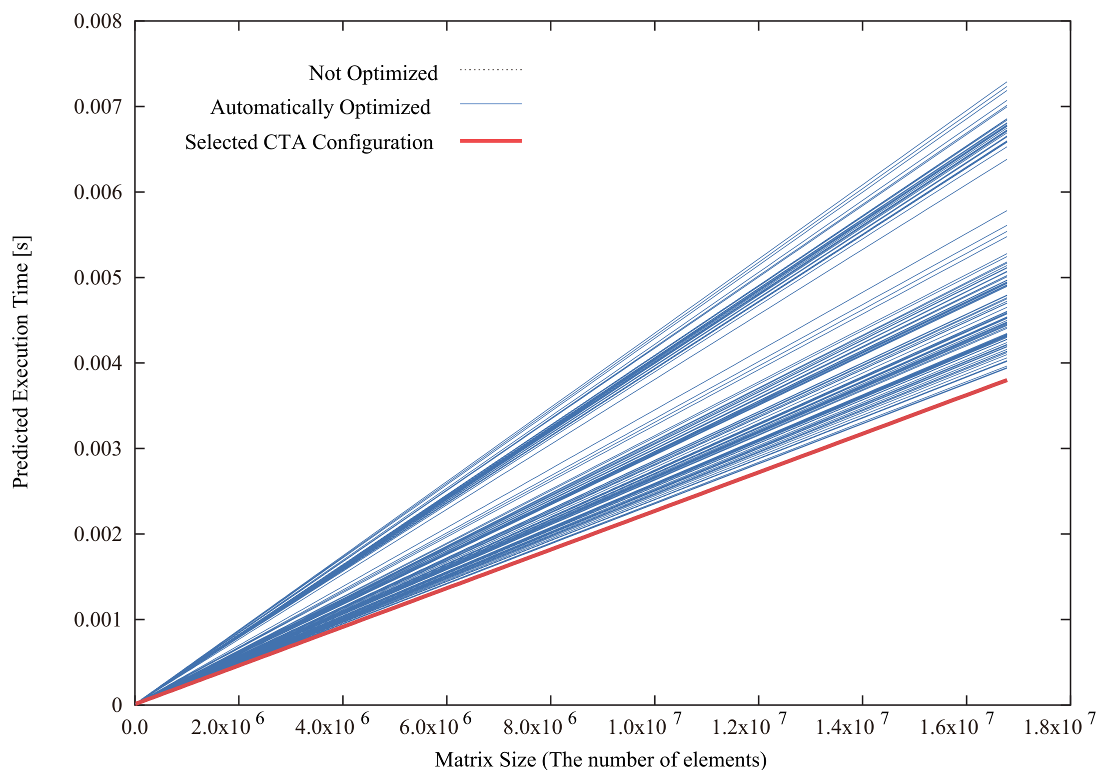



Figure 3.21: The performance model of each CTA configuration on the LU decomposition.

predicted and actual execution times of each CTA configuration, respectively. The maximum difference and the average difference are 0.582 and 0.187 in the normalized execution times, respectively. The correlation coefficient between the predicted and actual execution times is 0.627. The correlation coefficient between the predicted and actual rankings is 0.729. These two distributions show the emphatic correlation between the predicted and actual execution times.

In Figure 3.24, all the CTA configurations are sorted in ascending order of the actual execution time. In this figure, the actual execution time of the selected CTA configuration is marked with a circle. The proposed mechanism can also find the suboptimal CTA configuration for the LU decomposition program. Unlike in the case of the Himeno benchmark, the selected CTA configuration is the 13-th in the actual performance ranking, because there are many CTA configurations whose performances are comparable to the optimal. The selected CTA configuration is the first in the predicted performance ranking. Since the actual performance ranking in suboptimal CTA configurations frequently varies by measuring errors, the difference in execution time is important. Figure 3.25 shows that the prediction of the

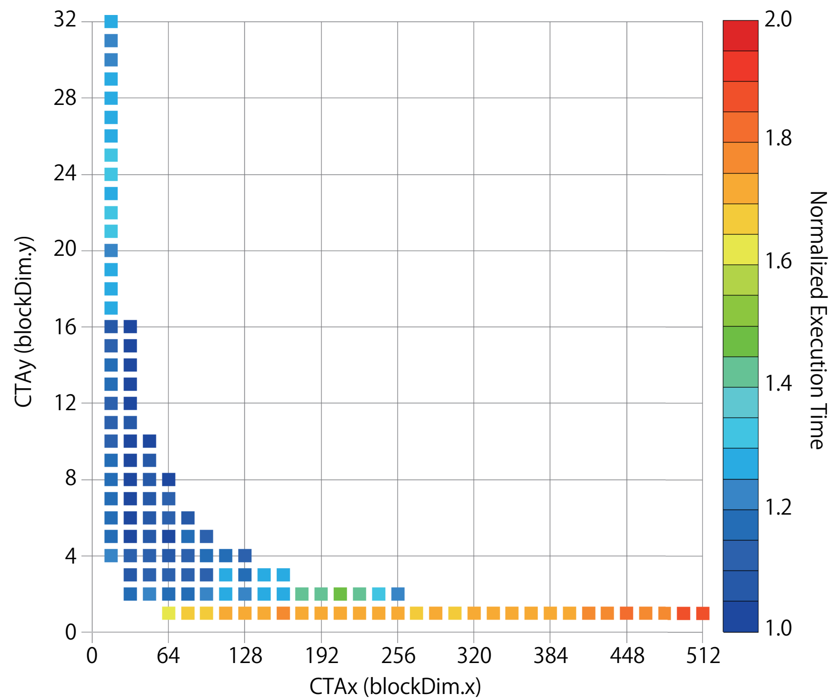



Figure 3.22: The predicted execution time of each CTA configuration on the LU decomposition.

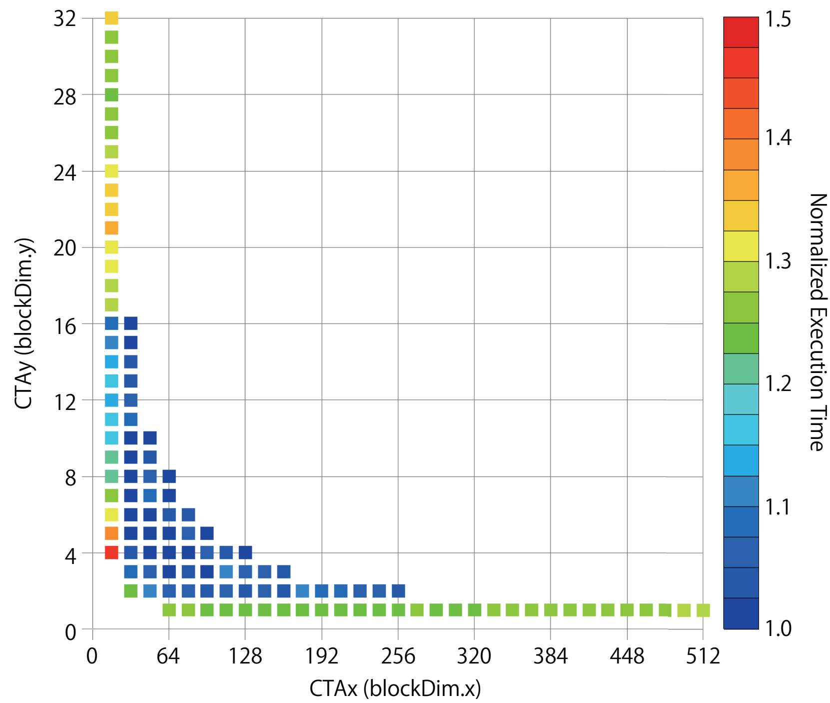



Figure 3.23: The actual execution time of each CTA configuration on the LU decomposition.

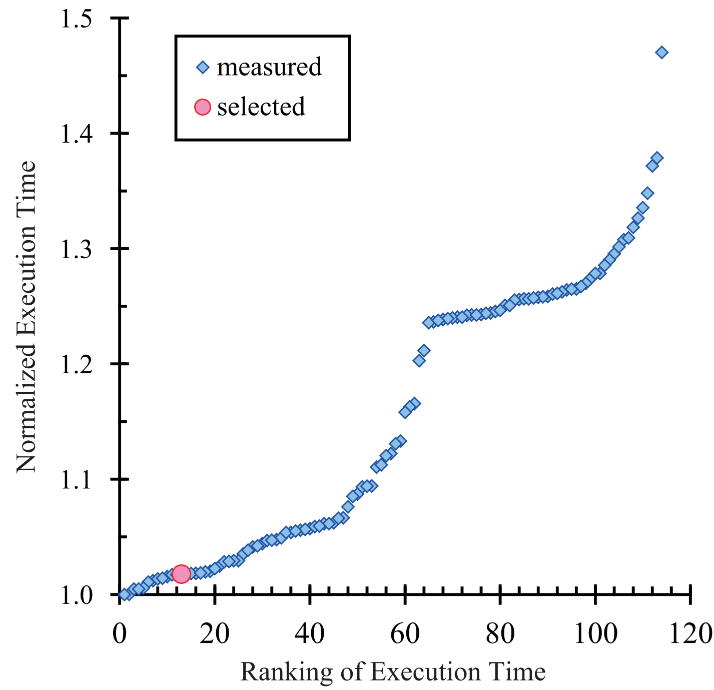



Figure 3.24: The ranking of actual execution time on the LU decomposition.

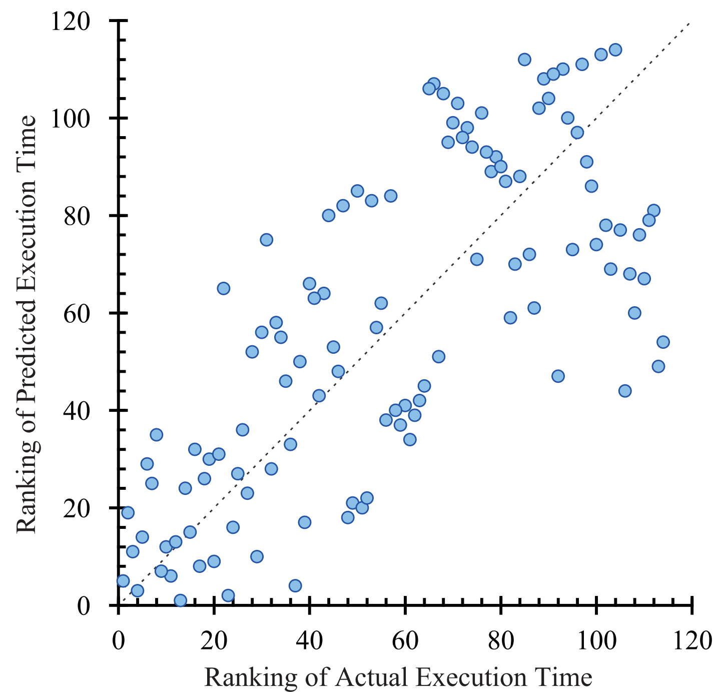



Figure 3.25: The correlation between the predicted and the actual rankings of CTA configurations on the LU decomposition.

mechanism is not so accurate in terms of the correlation between the predicted ranking and the actual ranking of the CTA configurations. However, the difference in execution time between the selected CTA configuration and the optimal one is still small and only 1.7%, of the total execution time. In this evaluation, the difference in execution time between the worst CTA configuration and the optimal one is 47%, and the average difference is 15%.

The evaluation results demonstrate that the proposed mechanism can find the appropriate CTA configuration for the Himeno benchmark and the LU decomposition based on the profile data obtained with smaller-size data sets. When only a few CTA configurations can achieve the comparable performance with the optimal one, the proposed method can easily find an appropriate CTA configuration. In addition, it is also shown that the prediction accuracy decreases because there are many suboptimal configurations. In this case, although it may be difficult to find the optimal configuration, the proposed mechanism can select a suboptimal one and avoid inappropriate CTA configurations. It is acceptable in many cases because the performance difference between the selected and the optimal ones is sufficiently small.

## 3.5 Concluding Remarks

In CUDA, to improve sustained performances of programs, it is required to optimize those programs for a particular architecture of a GPU. In this chapter, two performance tuning strategies are proposed for automatically-generated CUDA programs.

One of the performance tuning strategies is optimization of memory accesses that consists of reusable data prefetching and adjusting access patterns. These optimizations can improve sustained memory bandwidths of a program by efficiently using the shared memory. The reusable data prefetching method finds highly-reusable data blocks in streams and stores those blocks on the shared memory in advance. As the latency of the shared memory is very short, threads in a CTA can access at high memory bandwidths. The adjusting access patterns method uses the shared memory as a low-latency read buffer to avoid inefficient memory access.

The other is automatic tuning of CTA configuration that determines the number of threads executed on a MP. In GPUs, to hide the global memory access latency, a sufficient number of threads must be assigned to one MP. However, too many threads cause the shortage of hardware resource. Since the optimal CTA configuration depends on both the computation of a program and a GPU, it is labor-intensive even for expert programmers to find the optimal CTA configuration. The proposed tuning mechanism automatically finds the optimal CTA configuration from certain candidates that are derived from architectural features of GPUs. To find the optimal one, the proposed method uses the slope of a linear performance model. If the angle of the slope is the smallest, the corresponding CTA configuration is selected as the optimal one.

Evaluation results with the Himeno benchmark and the LU decomposition indicate that two performance tuning strategies effectively improve the sustained performance of their CUDA programs. From the evaluation results, it is demonstrated that reusable data prefetching can improve sustained memory bandwidth irrespective of GPU architectures, especially for programs that have absolute indexing operators in the SPRAT language.

The CTA configuration tuning method can automatically find the optimal or suboptimal CTA configuration based on the profiling data. In the case of the Himeno benchmark, there are a few suboptimal configurations. In this case, the proposed method selects the second optimal configuration whose difference in execution time to optimal one is less than 1%. In the case of the LU decomposition, there are many suboptimal configurations. The proposed method selects the appropriate one and avoids inappropriate configurations. Hence, the CTA

configuration tuning method reduces the labor-intensive performance tunings.

As described above, the proposed methods for automatic performance tuning are effective to alleviate burdens of programming in CUDA. Although these methods use some architecture-specific features, the proposed methods are applicable to other accelerators. Since these methods exploit the features of the SPRAT language to extract useful information such as the position of highly-reusable data blocks and memory access patterns of the kernels, these methods can optimize programs based on the information even if any other accelerators are utilized. Moreover, it can always be assumed that the execution time of a SPRAT program is proportional to the size of output data if an accelerator can execute stream processing. Therefore, the proposed method can find the optimal execution parameters of such an accelerator in a short time by using the performance prediction model of SPRAT.

# Chapter 4

# Online Task Scheduling in Standard Programming Environments

## 4.1 Introduction

OpenCL [4] has been proposed as a standard programming interface that can use various accelerators in a unified manner. Currently, there are OpenCL implementations for major accelerators such as NVIDIA GPUs [47], AMD GPUs [48], Cell Broadband Engine [49], and multi-core CPUs [50]. Moreover, OpenCL can utilize accelerators in other nodes via a network by using Virtual OpenCL [51]. An OpenCL program can run with various accelerators without modification of the source code. Therefore, OpenCL alleviates the difficulty of runtime processor selection among various processors for programmers. However, it does not solve the task assignment problem, and a programmer has to explicitly assign each task to one of processors in the system.

An OpenCL program needs to be linked with an appropriate runtime library when it is launched. This means that the user should select an appropriate accelerator to exploit the full potential of the heterogeneous computing system, as shown in Figure 4.1(a), even though an application user may not know the details of the program. As the sustained performance of a program strongly depends on the combination of a computation and an accelerator, inappropriate task assignment results in severe performance degradation. This problem becomes more serious when multiple accelerators are available in a heterogeneous computing system.

If a heterogeneous computing system has multiple accelerators and an application has many *parallel tasks*, the sustained performance can be improved by assigning parallel tasks

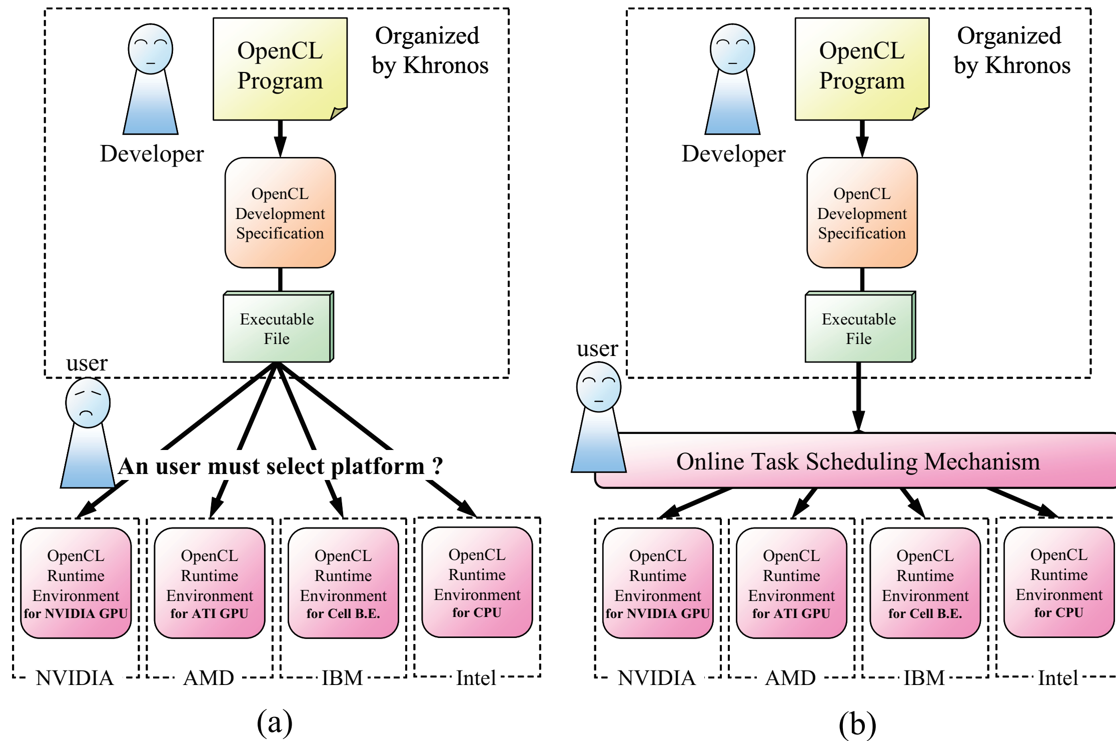



Figure 4.1: The problem of (a) appropriate task assignment and (b) the proposed solution.

for different accelerators. Ideally, the sustained performance is proportional to the number of used accelerators if appropriate load balancing is performed among multiple accelerators. Hence, load balancing is a key feature to exploit the high potential of the heterogeneous computing system. However, programmers may not know the number of available accelerators and the execution times on each accelerator in the user's system. Moreover, the execution times of tasks often depend on the argument values passed to the tasks and the size of input data. Therefore, it is difficult even for expert programmers to determine the optimal task assignment in advance of execution. On the other hand, it is impractical to delegate appropriate task assignment to application users.

To alleviate these difficulties in OpenCL, online task scheduling is useful to automate appropriate task assignment according to the computations at runtime. If programmers describe programs without considering the number of available accelerators in the system and their individual performance, the runtime environment has to assign each task to an appropriate accelerator. Hence, an online task scheduling method is needed to appropriately assign

tasks, as shown in Figure 4.1(b). For online task scheduling in OpenCL, three key technologies are necessary: a performance prediction method, a data and event dependency analysis method, and a task scheduling method.

Highly-accurate performance prediction is needed for online task scheduling in OpenCL. Since OpenCL has high programming flexibility and enables a programmer to implement various algorithms and data structures as with the standard C language, it is impossible to assume the explicit correlation between the execution time and the number of launched threads. Hence, simple prediction models that assume the correlation are not available in the performance prediction of OpenCL programs. There are two major approaches in performance prediction: analytic approaches [52, 53] and empirical approaches [44, 29]. The former approaches require a particular hardware model of each accelerator and its parameters that characterize the behavior of a program. These analytic approaches are not applicable to accelerators whose hardware details are not disclosed because the accurate hardware models cannot be built. On the other hand, the empirical approaches can predict the performance based on profile data and are applicable to any accelerators. However, a naive empirical method such as averaging past execution times may not achieve accurate prediction when the execution times vary drastically. Therefore, a new history-based method to predict the performance of OpenCL programs is required for online task scheduling.

To execute tasks in parallel, it is required that there are many parallel tasks in an application. When there are a few parallel tasks, it is impossible to improve performance even if multiple accelerators are available. Hence, programmers have to describe programs that have many parallel tasks. However, it is generally labor-intensive to find promising tasks that can potentially run in parallel. In this chapter, a data dependency analysis method is proposed to find parallelisms among tasks to facilitate task parallelization. In addition, an event dependency analysis method is also proposed to find unnecessary synchronization points that prevent the performance improvement. These methods enable programmers to optimize their programs without labor-intensive analyses.

Finally, an online task scheduling method based on performance prediction is proposed. The proposed scheduling method uses the *Minimum Completion Time* (MCT) algorithm [54] and tasks are assigned to an appropriate accelerator that can early complete executing the task. The performance evaluation is conducted using benchmarks that have parallel tasks.

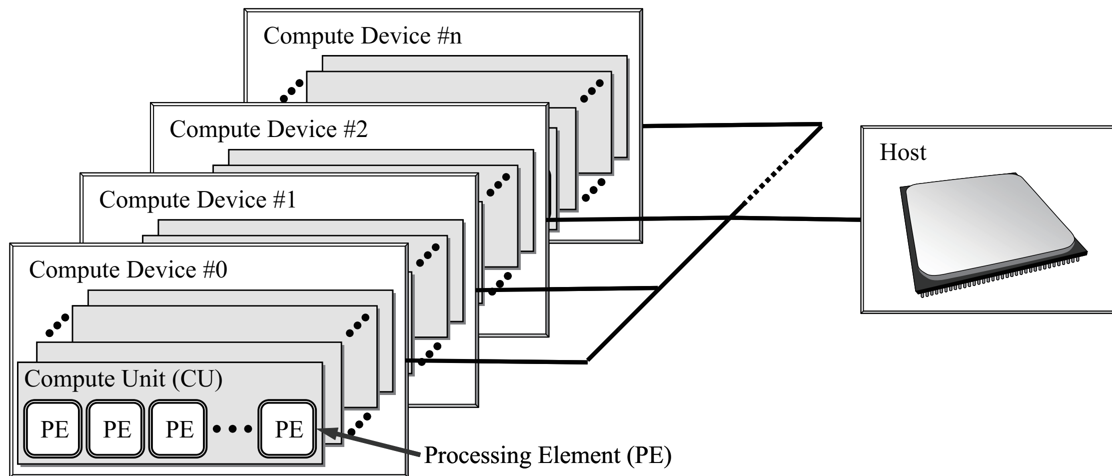



Figure 4.2: The platform model of OpenCL.

## 4.2 Related Work

### 4.2.1 OpenCL

OpenCL is a programming interface to use different accelerators in a unified manner. Figure 4.2 shows the platform model assumed in OpenCL. The platform model of OpenCL is similar to that of CUDA. One host manages multiple accelerators, called *compute devices*. In a compute device, there are several *compute units* that run blocks of threads. A compute unit consists of several *processing elements*, and each processing element executes one thread.

In the OpenCL programming model, a part of an application program running on accelerators is defined as a *kernel*, which is duplicated and executed by many threads in parallel. Figure 4.3 shows the thread hierarchy of OpenCL. When a kernel is launched, a programmer should set executing parameters according to the thread hierarchy of OpenCL. A thread called a *work-item* is the smallest unit in the thread hierarchy to execute a kernel. A *work-group* consists of some work-items and is a unit to be assigned to a compute unit. The number of work-items in a work-group is called the *work-group size* or the *local work size*, which is an important parameter that determines the granularity of computations to be assigned to each compute unit. An *NDRange* consists of several work-groups and is assigned to a compute device when a kernel is launched. The number of work-items in an NDRange is called the *NDRange size* or the *global work size*.

Figure 4.4 shows the queuing model of OpenCL. A *command* is dispatched to transfer

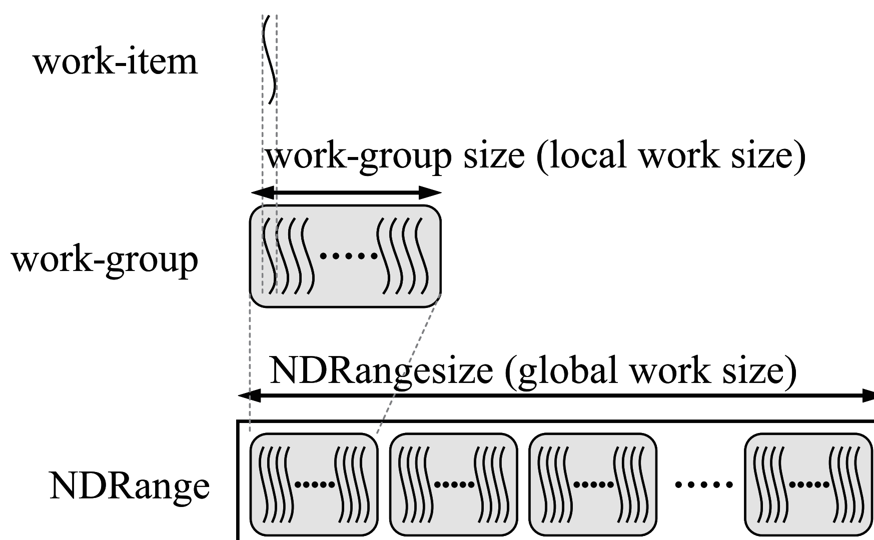



Figure 4.3: The thread hierarchy of OpenCL.

data and to execute kernels. A command that executes a kernel corresponds to one task of the kernel. To execute tasks in OpenCL, a programmer enqueues kernels to a waiting list corresponding to a compute device, called a *command queue*. Commands in a command queue are asynchronously executed from a host thread on a CPU. Commands in each command queue are independently executed each other. To exploit multiple accelerators, programmers take care of multiple command queues corresponding to each compute device.

When a command is enqueued to a command queue, explicit dependencies to other commands can be set by using an *event object*. An event object is an object related to the state of a particular command and is created at enqueuing the command. For example, when a command to execute a kernel K1 is enqueued, a programmer can get the event object E1 related to that command. Next, when a command to execute a kernel K2 that has a dependency to a kernel K1 is enqueued, a programmer can explicitly indicate the dependency by passing an event object E1. Thus, a programmer can give an order constraint to execute kernels K1 and K2.

In OpenCL, there are two types of functions to enqueue commands to a command queue; a *non-blocking* function and a *blocking* function. When a non-blocking function is called, the host thread does not wait for the completion of the command, and the command is executed asynchronously. On the other hand, when a blocking function is called, the host thread waits for the completion of the command. The blocking function call creates

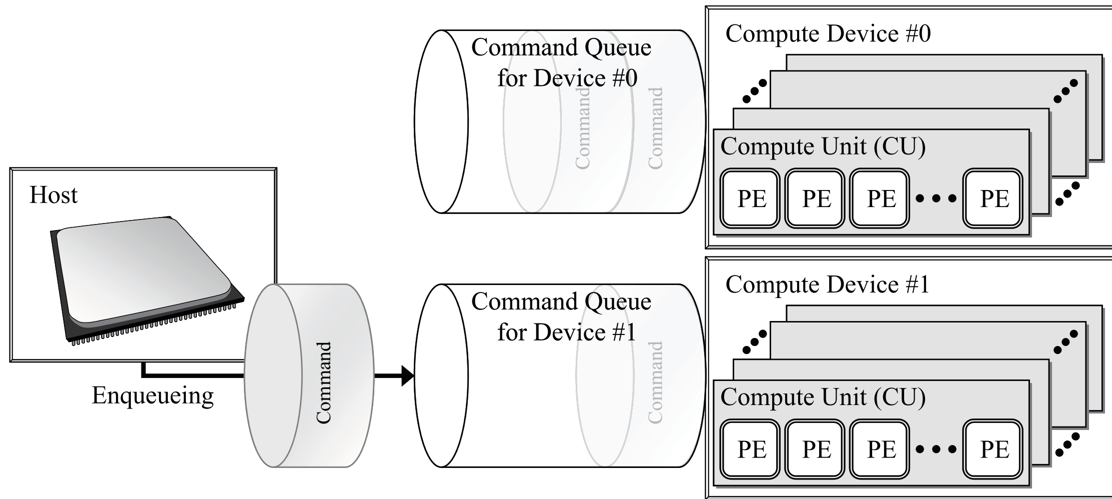



Figure 4.4: The queuing model of OpenCL.

implicit order constraints because the subsequent command enqueued after this call cannot be enqueued until the preceding command is completed.

All data accessed by kernels are allocated on the device memory and managed as *memory objects*. Only data transfer APIs and kernels can read and write data in memory objects. When a memory object is created, flags of access modes such as read-only, write-only, and read-write are specified. Hence, accesses to a memory object, called a *Read/Write relation*, can be roughly analyzed by using these flags. However, one kernel might need to read the object written by another kernel. In this case, the Read/Write relations of the kernels are not read-write even if read-write flag is set. Therefore, it is needed to analyze a kernel to figure out the Read/Write relation between a kernel and a memory object.

### 4.2.2 Performance Prediction Methods

There are several analytic methods that build a prediction model by considering the characteristics of a given target architecture and an application program.

Baghsorkhi et al. have proposed an adaptive performance prediction model for GPUs[52]. Their work focuses on the NVIDIA GPU execution model, in which threads are grouped to a unit called a *warp* and executed in a *Single-Instruction, Multiple-Threads*(SIMT) manner [36]. This prediction model can achieve a high accuracy by precisely analyzing the execution flow of a program with considering the warp execution mechanism. However, this model cannot be applied to the hardware of a different execution mechanism.

Hong et al. have also proposed an analytic performance prediction model for GPUs[55]. In their work, a model that can achieve highly-accurate performance prediction is derived from the numbers of memory accesses, operations, and active warps of the target architecture. However, the accuracy would be low when the performance is predicted for unexpected architectures. In addition, it is difficult to predict the performance if the compute device used in execution of a kernel is decided at runtime.

The simplest empirical method to predict execution time of a kernel is to average all past execution times under the assumption that the execution time of a kernel does not significantly change. This simplest method is the most light-weight method. However, the accuracy of this prediction method is degraded if the execution time varies drastically.

Some researches use runtime information to further improve the accuracy of the predicted execution time. If there is a relationship between runtime information and execution time, the accuracy of the predicted execution time can be improved by using the runtime information. Trancoso et al. have reported that the difference in performance between CPUs and GPUs varies depending on the size of processed data and the number of operations on the data [29]. As the size of data determined by the problem size is usually decided at runtime, it is impossible to statically estimate the speedup ratio obtained by using GPUs.

In runtime environments adopting a so-called SPMD (Single-Program, Multiple-Data) programming model such as CUDA and OpenCL, there is a correlation between the number of threads and the execution time in many cases. Therefore, in the SPMD programming model, the execution time of a kernel is roughly modeled by

$$\mathrm{T}_{\text{execution}} = \frac{T_{\text{thread}} \times N_{\text{thread}}}{N_{\text{parallel}}} + \text{overheads}, \quad (4.1)$$

where $T_{\text{execution}}$ is the total execution time of a kernel, $T_{\text{thread}}$ is the execution time of each thread, and $N_{\text{thread}}$ and $N_{\text{parallel}}$ are the number of threads and the number of processing elements in the underlying hardware, respectively. Equation (4.1) is a linear prediction model with the global work size.

Since the execution time of each thread is almost unchanged, it is possible to achieve accurate prediction by Equation (4.1). For simple kernels such as computation without loops and branches, this simple approach can achieve sufficiently-accurate prediction. In SPRAT [44], this performance model is used for the prediction of execution times. SPRAT

assumes a linear relationship between the size of a memory object to be written computation results and the execution time, and it builds a performance model by using the size of the memory object. Evaluation results in Chapter 2 show that this performance model can achieve accurate prediction enough for stream processing in SPRAT.

However, in the case where $T_{\mathrm{thread}}$ depends on runtime parameters, the simple approach cannot accurately predict the execution time of a kernel. One effective approach is to use General Linear Least Square(GLLS) [56]. This approach assumes that there are relations between the execution time and multiple runtime parameters such as the global work size, the local work size, and argument values of a kernel. Thus, this approach can build a linear model with these runtime parameters. Although GLLS is an effective method to build such a linear model, the accuracy of the performance prediction is degraded if there is a non-linear relationship between some runtime parameters and the execution time. Therefore, it is necessary to identify useful runtime parameters to improve the accuracy of a prediction model built by GLLS.

For online task scheduling, a history-based performance prediction method is proposed in this chapter under the assumption that a compute device to execute a kernel is decided at runtime. The purpose of the proposed prediction method is to accurately predict the execution time of an OpenCL program without changing the OpenCL programming model. To improve the accuracy of the prediction, the proposed method uses not only the time of the past kernel execution, i.e. profile data, but also runtime parameters given at the kernel launch. It does not assume any particular architecture for performance prediction. Therefore, it enables to predict the performance of even an unknown processor if the profile data of the processor are available.

### 4.2.3 Methods for Dependency Analysis

Data dependency analysis is fundamental to get important information for optimizing programs, and many methods have been proposed. In compilers, data dependency analysis among instructions is performed to schedule instructions [57]. Moreover, in C or FORTRAN languages, it is important to find *parallel code blocks* that can be independently executed on different processors. There are many researches to find parallel code blocks in high-level programming languages.

Kasahara et al. proposed an automatic parallelization method for a FORTRAN program [58]. This method extracts parallel basic blocks, iterative blocks, and subroutine blocks

from a program. In this method, a *macro-flow graph* that shows data dependencies and control dependencies between basic blocks is generated. A *macro-task graph* that indicates parallel blocks is also created from a macro-flow graph by using *Earliest Executable Condition* analysis [59]. Then, this method automatically inserts OpenMP directives to FORTRAN programs based on the macro-task graph.

Diamos et al. have proposed a method to perform speculative execution of kernels in order to exploit task-level parallelism [60]. Their proposed method statically analyzes data and control dependencies in a program written by using the Harmony framework. In addition, this method shows the maximum limit of kernel-level parallelism in a program and the estimated performance improvement by speculative kernel execution. This method assumes that programmers can appropriately insert special annotations to a program. Therefore, programmers must modify a program to insert annotations in order to indicate accurate Read/Write states of a kernel. If a programmer inserts inappropriate annotations, dependency analysis and speculative execution are failed. Hence, programmers must have sufficient knowledge of annotations provided by the Harmony framework.

Dependency analysis methods without adding any annotations are proposed in this chapter to facilitate task parallelization. The proposed method records memory accesses from kernels and API function calls. It analyzes these histories to extract data and event dependencies. Programmers can optimize a program based on the analysis results and modify it to effectively exploit multiple accelerators.

### 4.2.4 Methods for Task Scheduling

In task scheduling, there are two approaches; offline scheduling and online scheduling.

Topcuoglu et al. have proposed two offline task scheduling algorithms for heterogeneous processors; *Heterogeneous Earliest Finish Time* (HEFT) and *Critical Path on a Processor* (CPOP) [61]. In the HEFT algorithm, a task that has a long path of upward dependencies is given a high priority, and that task is preferentially scheduled to the processor that can finish executing early. The CPOP algorithm is similar to the HEFT algorithm, but it finds a critical path of dependencies among tasks, and then all tasks in the critical path are assigned to a *critical-path* processor.

Ilavarasan et al. have proposed *Performance Effective Task Scheduling* (PETS) algorithm for scheduling in heterogeneous computing environments [62]. Their proposed method performs static scheduling based on a data dependency graph. In the PETS algorithm, tasks

are sorted based on their data dependencies, and a priority is given to each task according to its average execution time, the cost of data transfer, and the rank of predecessor task. Then, a task with the highest priority is scheduled to a processor that can early complete the execution of the task; this scheduling is applied to all tasks in order of priority. However, as these methods assume that the execution times of tasks are already known and are not changed at runtime, these methods cannot perform in the case where the execution time varies dynamically according to the runtime parameters such as the argument values.

Grewe et al. have proposed a static task partitioning approach in OpenCL [63]. They suppose that experimental task partitioning cannot achieve maximum performance, and inappropriate task partitioning causes performance degradation. Hence, their proposed method determines task partitioning by using *Support Vector Machines*(SVMs) based on the characteristics of programs. However, this method cannot deal with three or more processors, and its accuracy of task partitioning depends on the quality of machine learning. Therefore, this method needs a sufficient number of benchmarks for machine learning. Moreover, as this method assumes only 11 classes of task partitioning, it is impossible to finely adjust the ratio of task partitioning.

As these offline scheduling approaches assume that the execution time is already known, offline scheduling methods are not suitable for the case where the execution time varies at runtime. Therefore, online scheduling is needed to perform automatic load balancing in OpenCL.

*Minimum Execution Time* (MET) is a simple online scheduling algorithm. The scheduler based on the MET assigns a task to a processor that can minimize the execution time of the task. In the MET algorithm, the execution time ($E_i^j$) of a task $j$ is estimated for each processor $i$. Then, the task is assigned to a processor whose $E_i^j$ is minimum. In the MET algorithm, if there is the difference in performance between processors, tasks are concentrated on the faster processors.

*Minimum Completion Time* (MCT) is another scheduling algorithm proposed by Maheswaran et al. for heterogeneous computing systems [54]. In the MCT algorithm, the completion time of the task $j$ on the processor $i$ is calculated by

$$C_i^j = W_i + T_i^j, \tag{4.2}$$

where $W_i$ is the total execution time of assigned tasks on the processor $i$, and $T_i^j$ is the execution time of the task $j$ on the processor $i$. Then, the task $j$ is assigned to a processor

whose $C_i^j$ is minimum. The MCT algorithm is superior to the MET algorithm because it can avoid the concentration of tasks on a particular processor.

As these algorithms need the accurate execution times of tasks, they are often used for offline scheduling based on profiling. However, if the execution time of a task varies at run-time depending on input data and available accelerators, it is difficult to obtain the accurate execution times of the tasks by only a profiling. Hence, the proposed online task scheduling method adopts the MCT algorithm for online scheduling based on the performance prediction with profiling.

## 4.3 Online Task Scheduling Based on Performance Predictions

### 4.3.1 History-based Performance Prediction with Profile Data Classification

In OpenCL, several runtime parameters are necessary, and some of the parameters affect the execution time. If the value of a runtime parameter has an effect on the execution time of a kernel, the effect can further be categorized into a linear or non-linear effect. The parameters with linear effects include the number of threads, the length of a loop in the kernel, and so on. The parameters with non-linear effects include a flag that changes the execution path of a kernel code, an index that changes the memory address accessed in the kernel, and so on.

In Section 4.3.1, a performance prediction method is proposed to improve the accuracy of the prediction by profile data classification. The proposed method can eliminate parameters with non-linear effects from the explanatory variables for GLLS to build an accurate prediction model. This section describes

- runtime parameters that are available at runtime in OpenCL,
- runtime parameter types and how to categorize runtime parameters into three types, and
- the procedure of the proposed prediction method.

Figure 4.5 shows the overview of the proposed prediction mechanism. The proposed method categorizes runtime parameters into three types according to the effect of each runtime parameter on the execution time, and classifies profile data according to the values of the non-linear parameters. Then, the proposed method builds multiple linear prediction models with classified profile data to eliminate the effect of non-linear parameters and predicts the execution time of a kernel with one of the prediction models.

#### Runtime parameters in OpenCL

The proposed method uses all the runtime parameters available when a kernel is launched in OpenCL. There are four groups of available runtime parameters.

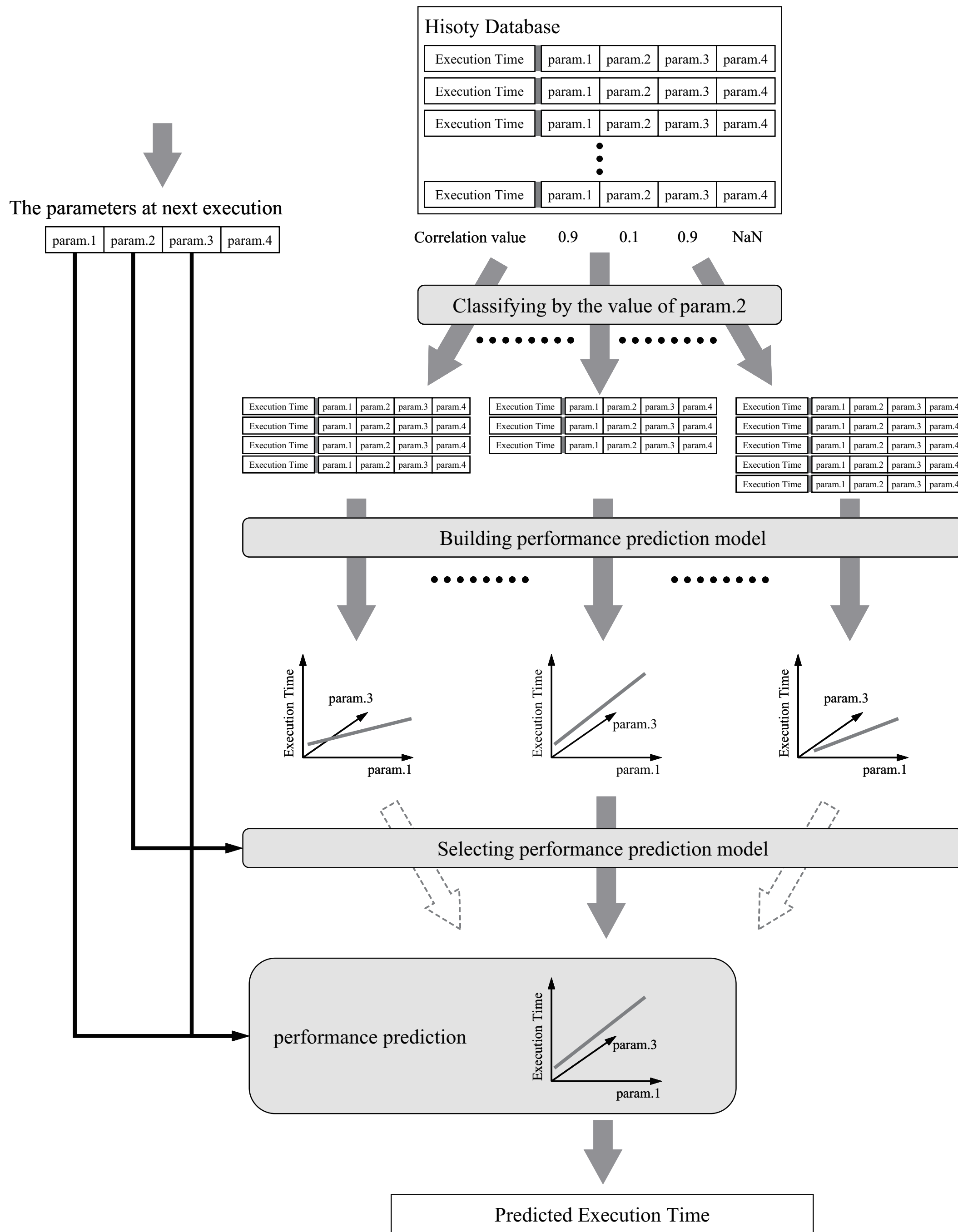



Figure 4.5: Prediction procedure of the proposed method.

(1) The memory address of an API call to launch a kernel, i.e. the memory address of calling clEnqueueNDRangeKernel.

(2) The global work size, which is the number of threads in a NDRange.

(3) The local work size, which is the number of threads in a work-group.

(4) Argument values given to a kernel.

From the memory address of a caller, the proposed performance predictor can detect that a kernel is called at different positions in a program. This is important for accurate performance prediction because a kernel might be called in different ways depending on the caller positions. The global and local work sizes usually affect the execution time because they change the number of threads working in parallel and sharing data. The values of kernel arguments may or may not affect the execution time of a kernel because they are used in various ways, e.g., to determine the length of a loop, the target of a branch, the address of processed data, and so on. If a pointer of a memory object is passed to a kernel, the execution time of a kernel depends not on the value of the pointer itself but often on the size of the pointed memory object. Therefore, the size of the memory object is recorded and used for performance prediction.

#### Profile data classification

As GLLS [56] used in the proposed method assumes that there is a strong correlation between explanatory variables and an objective variable, its accuracy of the model might decrease if a parameter with non-linear effects is included in the explanatory variables. Hence, to clarify the effect of each parameter on the execution time, runtime parameters are categorized into three types, N-type, W-type, and S-type.

**N(No effect)-type** parameters do not have any effects on the execution time of a kernel. N-type parameters include constant values and variables that do not vary during the execution. Use of parameter values in this type may degrade the accuracy of the prediction and increase the prediction time. Hence, these parameters are not used for performance prediction.

**W(weak correlation)-type** parameters have non-linear and/or irregular effects on the execution time of a kernel. W-type parameters include a flag to determine a branch target and an index of a memory array accessed in the kernel. As these parameters are only weakly

correlated with the execution time of a kernel, the accuracy of the prediction is degraded if these parameters are used to build a performance model. However, the execution time of a kernel sometimes drastically changes depending on these values. Therefore, the proposed method builds a different performance model for each value of a W-type parameter. The profile data that have the same values of W-type parameters are used to build one performance model. As a result, the proposed method uses multiple linear performance models to predict the execution time of one kernel.

**S(strong correlation)-type** parameters are strongly correlated with the kernel execution time. Therefore, the execution time can be predicted with a linear prediction model of these parameters built by GLLS. S-type parameters include the length of a loop and the global work size.

In the proposed method, every runtime parameter is categorized into one of the three types based on the correlation value between the parameters and the execution time of a kernel. If a correlation value cannot be calculated, the parameter is categorized into N-type and is not used for prediction. If a correlation value is less than a threshold, $C_T$, the parameter is categorized into W-type. Otherwise, it is categorized into S-type.

In this dissertation, the threshold, $C_T$, is empirically defined. The performance model is robust to the threshold value. Therefore, unless the threshold value is set to an extreme value, it hardly affects the prediction results.

#### The procedure of the proposed method

The prediction mechanism is performed when a sufficient amount of profile data has been obtained. At launching a kernel, as all the runtime parameters used for prediction are available, the mechanism records necessary parameters. The proposed prediction mechanism consists of four phases, as shown in Figure 4.5.

In the first phase, a correlation value is calculated to decide the type of each runtime parameters. As a result, each runtime parameter is categorized into one of N-type, W-type, and S-type.

In the second phase, profile data classification is performed. Based on the values of W-type parameters, profile data that have been obtained with the same values of W-type parameters are grouped and are used to build a linear prediction model. For example, param2 in Figure 4.5 is a W-type parameter, and hence a linear model is derived from some of the profile data that are obtained with a particular value of param2. As a result, param2 can be

considered a constant value when the linear model is selected for prediction in the fourth phase. Therefore, non-linear and/or irregular effects of param2 are removed from prediction models.

In the third phase, a linear prediction model for particular values of W-type parameters is built by GLLS with only S-type parameters. To build the prediction model, the proposed method determines model parameters $\alpha$ in Equation (4.3) by GLLS. GLLS is an arithmetic method to find the parameters of a linear model that conduct an objective variable such as execution time from explanatory variables such as runtime parameters. Hence, GLLS can minimize the difference between the predicted execution time and the actual execution time.

$$T_{predicted} = \alpha_0 + \sum_{i=1}^{n} \alpha_i S_i, \tag{4.3}$$

where $T_{predicted}$ is the predicted execution time, $S_i$ is the value of the $i$-th S-type parameter, and $n$ is the number of S-type parameters, respectively. For example, param1 and param3 in Figure 4.5 are S-type parameters and are used as explanatory variables of a linear prediction model. However, if there is a strong correlation between two explanatory variables, it is failed to properly build a linear prediction model because of multicollinearity. In the proposed method, if a strong correlation between two explanatory variables is detected, the mechanism uses only one of these parameters that has a large correlation value to avoid multicollinearity. In addition, if two runtime parameters have the same correlation value, the mechanism selects one runtime parameter according to a predetermined order.

In the fourth phase, the execution time is predicted by using the values of runtime parameters. The values of W-type parameters are used to select a linear prediction model. Then, the values of S-type parameters are used with the selected model to predict the execution time.

Since performance prediction models must be built for individual compute devices, the proposed method has to execute a kernel on every available device. The prediction mechanism does not work until a statistically-sufficient amount of profile data is obtained. Until the mechanism works, the average execution time of past kernel executions is used as the predicted execution time.

### 4.3.2 Data and Event Dependency Analysis Methods

Since a program should have many parallel tasks to effectively use multiple accelerators, a data and event dependency analysis method is proposed in Section 4.3.2 to facilitate task parallelization. A data dependency analysis method detects data dependencies among tasks. It visualizes these dependencies for programmers to support optimizing programs. An event dependency analysis method detects event dependencies that are constraints of parallel execution such as unnecessary synchronization points.

The dependency analysis method uses the histories of memory accesses from tasks and API calls to enqueue commands. As a result, three graphs are created to visualize dependencies in a program; a data dependency graph, a memory access graph, and an event dependency graph. A data dependency graph indicates intrinsic data parallelism in a program. A memory access graph indicates the memory objects that cause false data dependencies and enables programmers to easily find these memory objects. An event dependency graph indicates the synchronization points and explicit/implicit dependencies among commands that execute tasks. By using these graphs, programmers can find optimizing points described below;

- the parallel tasks that can be executed in parallel,
- unnecessary order constraints and synchronization points, and
- wrong flags of access modes indicated at creation of the memory object.

Table 4.1: The types of data dependencies among tasks.

| Dependency type | the preceding task | the subsequent task | |
|---|---|---|---|
| Read after Write (RaW) | Write | Read | true data dependencies |
| Write after Write (WaW) | Write | Write | false data dependencies |
| Write after Read (WaR) | Read | Write | false data dependencies |
| Read after Read (RaR) | Read | Read | ignorable dependencies |

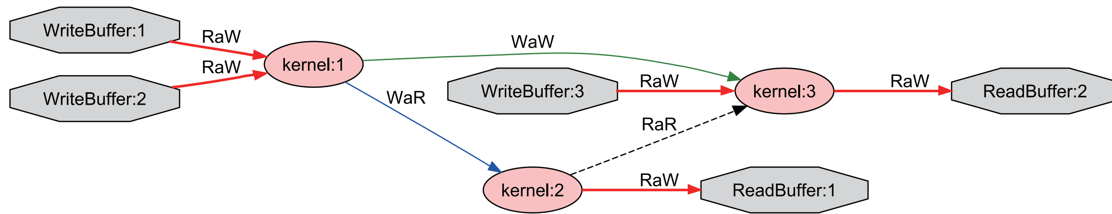



Figure 4.6: Examples of data dependencies among tasks.

### Data dependency analysis based on memory access histories

There are four kinds of data dependencies among tasks that access the same memory object, as shown in Table 4.1. In Figure 4.6, data dependencies are illustrated by nodes and directional edges that indicate tasks and dependencies, respectively[1]. Figure 4.6 shows different dependencies by different colors and labels. In this figure, RaW is the abbreviation of a `Read after Write` dependency, and the other dependencies are also abbreviated in the same way.

For parallel execution of tasks, it is required to keep *true data dependencies* and to eliminate *false data dependencies* to increase parallel tasks. The dependency of `Read after Write` is a true data dependency. On the other hand, the dependencies of `Write after Write` and `Write after Read` are generally false data dependencies. Hence, these dependencies can be removed by duplication of the memory object. However, in elimination of the `Write after Write` dependencies, it is needed to ensure that the subsequent task rewrites the whole of the memory object because there may be a true data dependency if the subsequent task writes only a part of the memory object. In this case, this false data dependency cannot be eliminated by duplicating the memory object.

The proposed analysis method records the history of memory accesses from tasks and can analyze all data dependencies by tracing this history. As a result, the proposed method outputs a *memory access graph* that indicates the read/write relations among tasks and memory objects.

If a kernel code is compiled at runtime by invoking an API function of clBuildProgram, the runtime environment of the proposed method implicitly parses the code to know the Read/Write states of each memory object. The analysis is performed as described below.

- If the left term of an expression is a pointer variable to a memory object, the pointer

[1]Graphviz[64] is used for visualization of graphs.

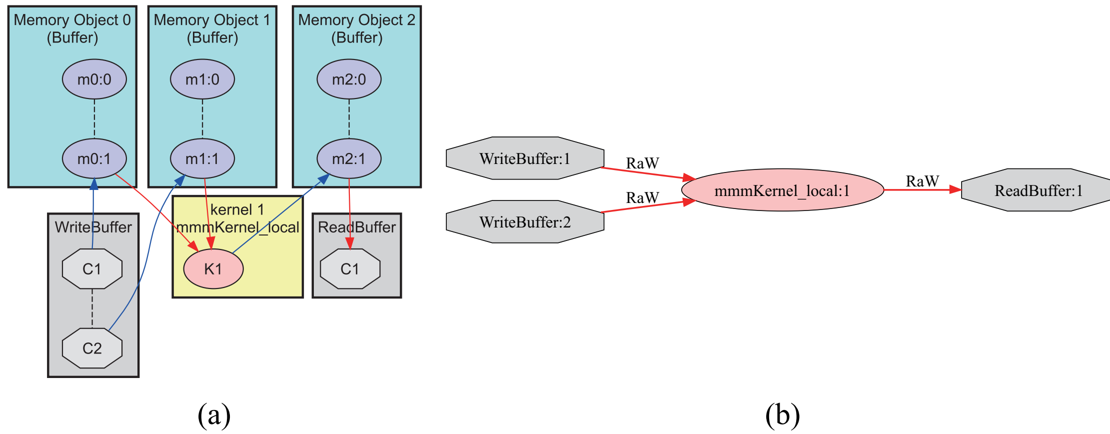



Figure 4.7: Examples of (a) a memory access graph and (b) a data dependency graph for matrix multiplication.

variables in the right term of the expression are related to the pointer variable in the left term.

- When an array indexing expression is found, a `read` flag is set to the pointer variable if it is in right term of an expression. Similarly, a `write` flag is set to the pointer variable if it is in left term of an expression. If the pointer variable has related pointer variables, the same flags are set to those related pointer variables.

This analysis is recursively performed on all kernel functions. If a kernel calls another function, the Read/Write states of the callee kernel are merged into those of the caller kernel. The runtime environment refers to the history of memory accesses and the Read/Write states of kernels. Then, it determines data dependencies among tasks. If it is failed to parse a kernel code, all memory objects passed to the task are estimated as the memory object is both read and written by the kernel.

Figure 4.7(a) shows a memory access graph of the `MatrixMultiplication` benchmark in ATI Stream SDK 2.3 [65]. In a memory access graph, octagon and ellipse nodes indicate the tasks and the data in memory objects, respectively. Write and read accesses to a memory object are shown by blue and red edges, respectively. The numbers in nodes indicate the count of kernel execution and the generation of data in a memory object that increase when a task is executed and data in a memory object are updated. For example, Figure 4.7(a) shows the relations;

- the `WriteBuffer` task writes the memory objects 0 and 1,

- the mmmKernel_local task reads the memory objects 0 and 1,
- the mmmKernel_local task writes the calculation results to the memory object 2, and
- the ReadBuffer task reads the results of multiplication from memory object 2.

Next, a data dependency graph is created from the memory access graph. In Figure 4.7(a), the Read after Write data dependency can be found by tracing the update history of the memory objects. For example, the first generation of data in the memory object 0 is read from the mmmKernel_local task after the WriteBuffer task writes. Thereby, the data dependency between those tasks becomes obvious, and the dependency edge is output in a data dependency graph. This procedure is applied to all generations of data in all memory objects. Then, a data dependency graph is created. Figure 4.7(b) shows a data dependency graph of the MatrixMultiplication benchmark.

A data dependency graph drawn by the proposed method visualizes the tasks that are potentially executed in parallel. Therefore, programmers can easily optimize programs to increase parallel tasks by using the graph.

#### Event dependency analysis based on histories of API function calls and event objects

For effective parallel execution, it is also important to remove unnecessary synchronization points in a program. Hence, an event dependency analysis method is also proposed to easily find unnecessary synchronization points. The proposed method outputs an *event dependency graph* that visualizes synchronization points in a program.

Event dependencies are the relations among commands that constrain the execution order. There are two kinds of event dependencies: explicit dependencies and implicit dependencies. An explicit event dependency is specified through an event object by programmers. On the other hand, an implicit event dependency is caused by calling a blocking function.

The proposed method uses the history of API function calls and event objects to analyze event dependencies. The history of API function calls includes the type of the command and whether it is enqueued by a blocking function or not. In an event dependency graph, a command that is enqueued by a blocking function is called a *blocking command*, and a command that is enqueued by a non-blocking function is called a *non-blocking command*. The proposed method analyzes this history and outputs event dependency edges according to the rule described below.

- If the subsequent command is a blocking command, the analysis method outputs the edge of implicit dependencies between the subsequent command and all the preceding non-blocking commands. Moreover, the analysis method outputs the edge of implicit dependency between the subsequent command and its immediately-preceding blocking command.

- If the subsequent command is a non-blocking command, the analysis method outputs the edge of implicit dependency between the subsequent command and its immediately-preceding blocking command.

- If dependency is specified by using an event object, the analysis method outputs the edge of explicit dependency between the commands of creating the event object and of using the object.

- If a pair of commands has both explicit and implicit dependencies, the proposed method outputs only the edge of explicit dependency.

- The API functions that just wait the completion of its preceding commands are certainly blocking functions and cannot be changed to non-blocking functions. Hence, the analysis method outputs the edges of implicit dependencies that connect the commands of such API function calls to their subsequent commands.

Figure 4.8 shows the event dependency graph of the `MatrixMultiplication` benchmark. In an event dependency graph, nodes indicate the commands to execute tasks, to transfer data, and to wait for the completion of its preceding commands. The explicit and implicit dependencies are expressed by black and blue edges, respectively. Moreover, the frame color of a blocking command is red, and that of a non-blocking command is black. The number in a node indicates the count of function calls.

By using this event dependency graph, programmers can easily find unnecessary synchronization points in a program. Figure 4.8(a) shows an event dependency graph of unoptimized program. There are unnecessary event dependencies due to unnecessary use of blocking functions. In this case, it is needed to specify necessary dependencies by using event objects and to convert blocking commands to non-blocking command. Then, unnecessary dependencies can be removed based on the event dependency graph. The program is optimized as shown in Figure 4.8(b).

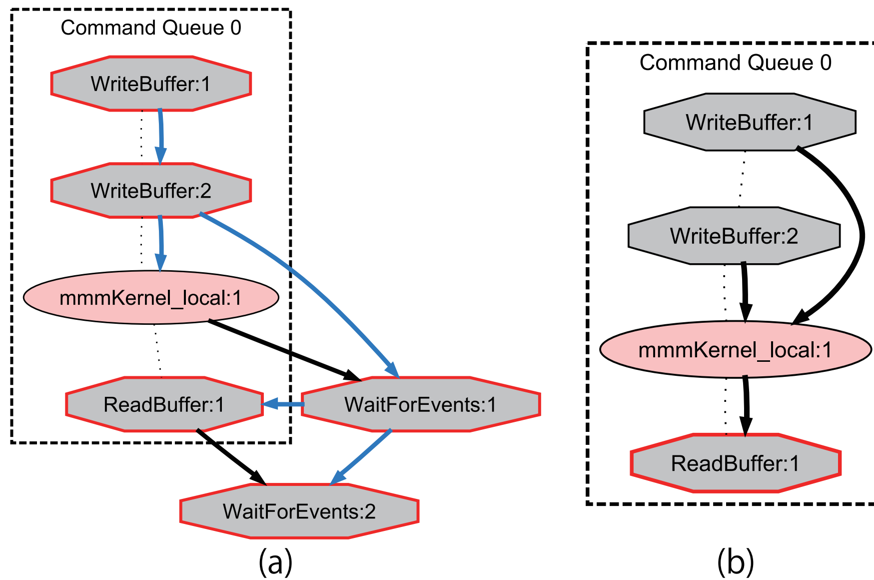


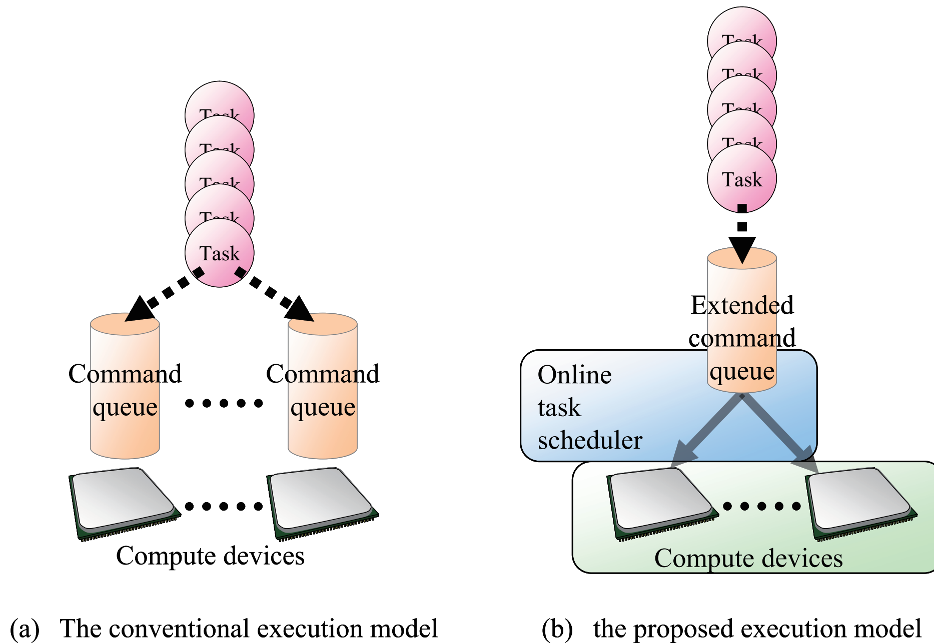

Figure 4.8: Examples of event dependency graphs for matrix multiplication. (a) The graph of unoptimized program. (b) The graph of optimized program.

### 4.3.3 Online Task Scheduling Based on the MCT Algorithm

In a conventional execution model of OpenCL shown in Figure 4.9(a), each command queue is directly bound with one compute device. In this case, programmers need to manage multiple command queues to use multiple accelerators. However, if programmers explicitly manage multiple command queues, the number of compute devices to execute tasks is fixed. Hence, it is impossible to fully exploit available devices if there are more devices in a heterogeneous computing system.

To use available accelerators efficiently, a new execution model with online task scheduling is proposed in Section 4.3.3. The proposed execution model binds a command queue to multiple compute devices. The command that is enqueued to the extended command queue is automatically assigned to an appropriate device by an online task scheduler shown in Figure 4.9(b). The proposed execution model enables programmers to describe programs without any consideration of the number of available accelerators and the performance difference among accelerators.

This online scheduling method assumes a program that is optimized based on the dependency graphs proposed in Section 4.3.2. The commands enqueued by non-blocking functions are considered *parallel commands* that execute parallel tasks.




(a) The conventional execution model (b) the proposed execution model

Figure 4.9: The conventional and proposed execution models in OpenCL.

The proposed scheduler works based on the MCT algorithm. Figure 4.10 shows the overview of the proposed online task scheduling method. First, when the command is enqueued, the scheduler predicts the execution time of each compute device to complete the command, as shown in Figure 4.10(a).

If a device does not have the valid data required by a task, it is needed to transfer the data from another device. In this case, the completion time of the device for the task is calculated with considering the overhead of data transfer, as shown in Figure 4.10(b). The overhead of data transfer can be easily estimated from the data size because that overhead is proportional to the size. Hence, the scheduler builds a simple linear prediction model to estimate the overhead. If the data transfer cannot start until the preceding command is completed due to the data dependency, the overhead of waiting for the completion is also required to predict the completion time of the subsequent command.

Finally, the command is assigned to an appropriate device that has the earliest completion time, as shown in Figure 4.10(c).

In this scheduling, once a task is assigned to a compute device, it is not rescheduled any more. In addition, the subsequent command does not overtake its preceding command: in-order scheduling is assumed in the proposed scheduling method. This is because compute

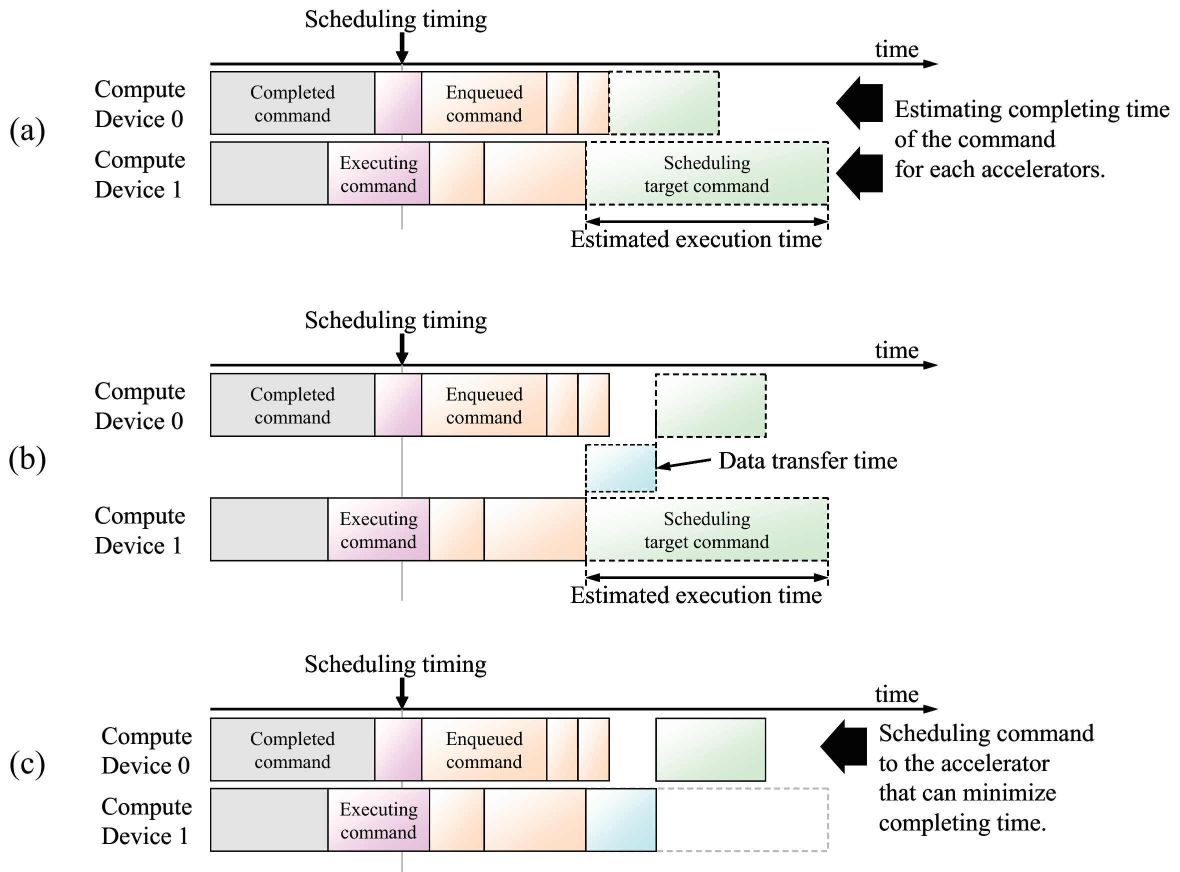



Figure 4.10: The procedure of the proposed online scheduling method based on the MCT algorithm.

devices work asynchronously with the host and a considerable synchronization overhead is required to migrate assigned tasks. Moreover, since the task assumed in this section is executed in a short time, the execution of the task may already be completed when the scheduler reschedules the task.

## 4.4 Evaluations

### 4.4.1 Evaluation of the Performance Prediction

In the following evaluation, four well-known benchmark suites are used to demonstrate the advantages of the proposed prediction method. The *AMD SDK benchmarks* and the *NVIDIA SDK benchmarks* are included in software development kits provided by accelerator vendors. The *SHOC benchmark suite*[66] and the *Parboil benchmark suite*[67] contain many fundamental and practical benchmarks. As all the programs in the Parboil benchmark suite are written in CUDA, three of them have been ported to OpenCL; the ported programs are CP, MRI-FHD, and MRI-Q. 126 kernels in these benchmark suites are used to evaluate the proposed prediction method.

Table 4.2 shows the experimental setup to evaluate the proposed prediction method. Each benchmark program is executed 40 times to obtain profile data and 10 times to evaluate the accuracy of the prediction. For several benchmark programs whose problem sizes can be adjusted without any source code modification, those programs are executed 10 times with four problem sizes for profiling, and then 10 times with the maximum problem size for evaluation. The threshold to discriminate between W-type and S-type parameters, $C_T$, is set to 0.8.

Figure 4.11 shows the breakdown of N-type, W-type, and S-type parameters categorized by the proposed method. Generally, in OpenCL, the global work size and the local work size significantly affect the execution time. However, in the 97.6% of the kernels, the local work size is set to a constant value. Hence, the local work size does not help improving the accuracy of the prediction. On the other hand, the global work size and argument values of a kernel change when the kernel is launched in different ways. Therefore, differences in

Table 4.2: Experimental setup to evaluate the proposed prediction method.

| Components | Specifications |
|---|---|
| CPU | Intel Core i7 920 2.66GHz |
| GPU | NVIDIA Tesla C1060 |
| OS | CentOS 5.5 (Linux 2.6.18) |
| Video Driver | NVIDIA Video Driver 260.19.21 |
| OpenCL | version 1.1 |

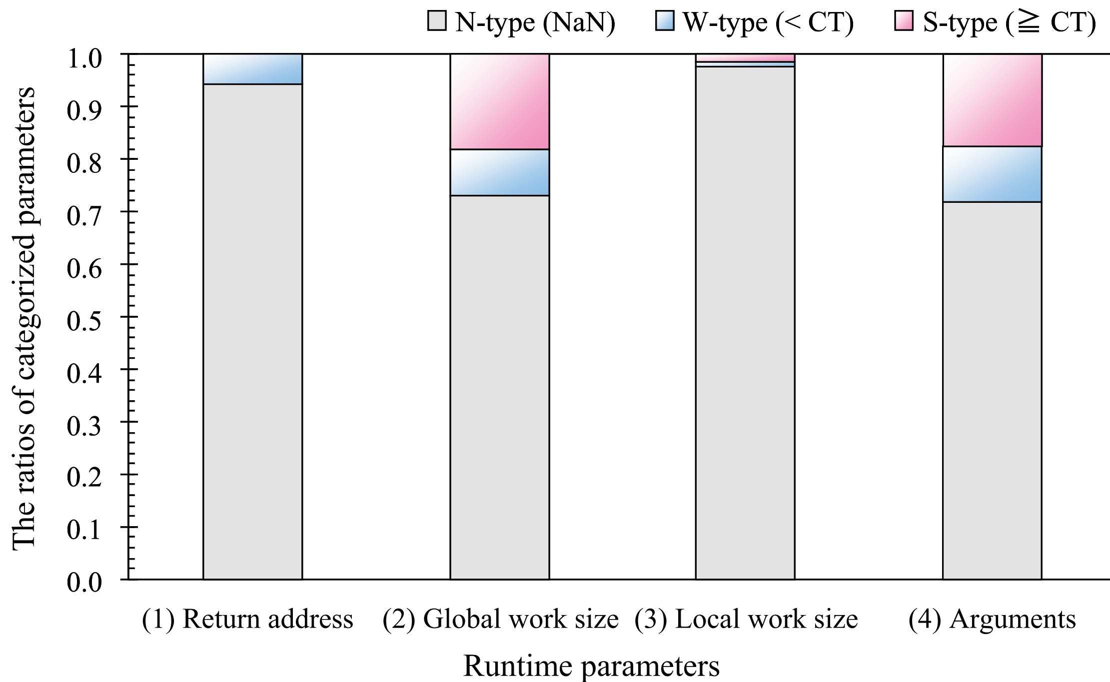



Figure 4.11: Usage of runtime parameters.

those parameters are worth considering to improve the prediction accuracy. The 18.3% of the global work size and the 17.6% of argument values of the kernels are categorized into S-type, respectively. Thus, it is important to use the global work size and argument values to build a linear model.

In the kernels used in this evaluation, the values of many runtime parameters are always the same at every kernel invocation. Hence, those parameters are categorized into N-type. As these N-type parameters do not contribute to the accuracy of the prediction, they should not be included in explanatory variables of a linear prediction model. The percentages of N-type parameters in the return address, the global work size, the local work size, and argument values are 94.4%, 73.0%, 97.6%, and 71.7%, respectively. These results indicate that there are many runtime parameters categorized into N-type. The proposed method can detect these N-type parameters and exclude them from prediction models.

The following evaluation compares the four prediction methods described below.

- *proposal* : The proposed method that builds a prediction model by GLLS using all the available runtime parameters **with** profile data classification.
- *glls* : The *glls* method that builds a prediction model by GLLS using all the available

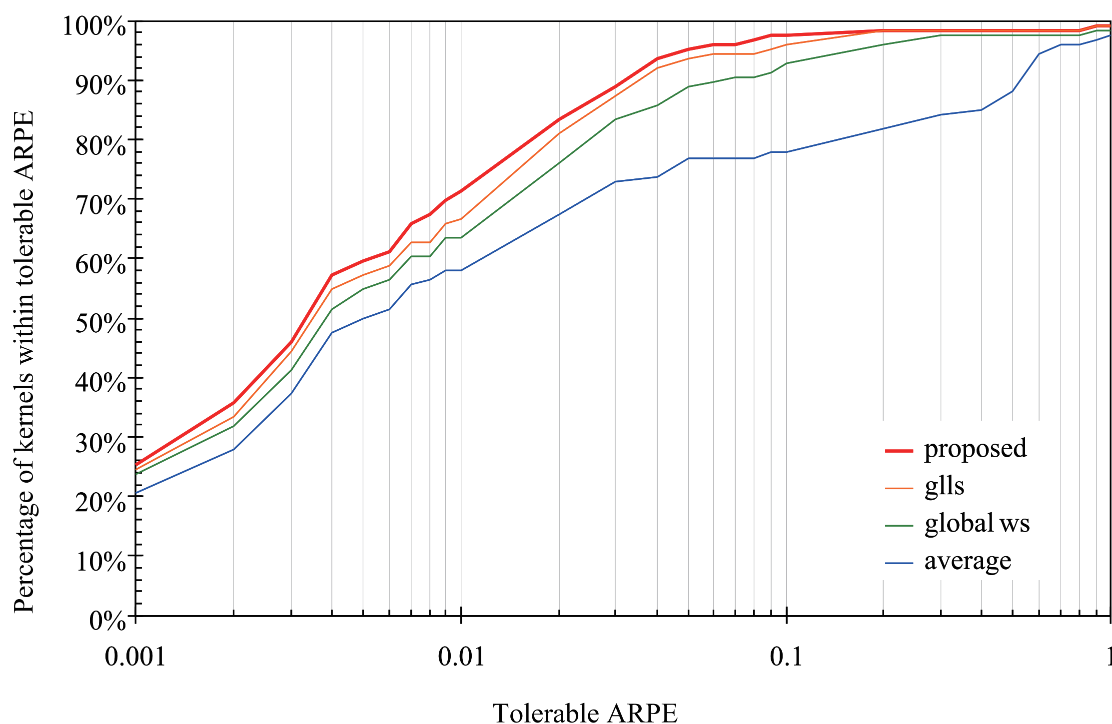



Figure 4.12: Percentage of kernels within tolerable ARPE.

runtime parameters **without** profile data classification.

- *global ws* : The *global ws* method used in SPRAT[44] that builds a prediction model by GLLS using **only** the global work size.
- *average* : The *average* method that uses the average value of past execution times without considering the effects of runtime parameters.

The accuracy of the prediction is evaluated by *Average Relative Prediction Error* (ARPE) defined by

$$\mathrm{ARPE} = \frac{\sum_{i}^{n} \frac{|T_{\mathrm{actual},i} - T_{\mathrm{predicted},i}|}{T_{\mathrm{actual},i}}}{n}, \tag{4.4}$$

where $T_{\mathrm{actual},i}$ and $T_{\mathrm{predicted},i}$ are the actual execution time and the predicted execution time, respectively, and $n$ is the number of prediction trials.

Figure 4.12 shows the percentage of kernels predicted within a tolerable ARPE. This figure indicates the relationship between a tolerable ARPE and the percentage of kernels

whose ARPEs are in a tolerable range. In every method, the percentage of kernels increases with the tolerable ARPE. There are two kernels, for which no method can achieve accurate prediction even if the tolerable ARPE is set to 1.0. For all the predictable kernels, the average ARPEs of *proposal*, *glls*, *global ws*, and *average* are 1.03%, 1.29%, 26.24%, and 55.99%, respectively. The average deviations of *proposal*, *glls*, *global ws*, and *average* are 0.000349, 0.000596, 7.30, and 13.5, respectively. Thus, the proposed method can achieve a low ARPE more steadily. Furthermore, the percentages of kernels whose ARPEs are less than 10% in *proposal*, *glls*, *global ws*, and *average* are 97.6%, 96.0%, 92.9%, and 77.8%, respectively. Accordingly, the proposed method can predict more kernels more accurately.

In many kernels, runtime parameters do not significantly vary even if their problem sizes are changed. In these cases, there is no difference in accuracy between the prediction methods, and all the methods can achieve highly-accurate prediction. However, for the kernels whose runtime parameters change with the problem sizes, the accuracy of *average* remarkably degrades. Even in such situations, the other methods can achieve high prediction accuracies.

Unlike SPRAT, $T_{\mathrm{thread}}$ and $N_{\mathrm{thread}}$ in Equation (4.1) do not always depend on the problem size in OpenCL programs. As a result, *global ws* cannot achieve as accurate prediction as *glls* and *proposal*. For example, in the `bitonicSortLocal` kernel of the oclSortingNetworks benchmark, there is a loop whose length is decided by an argument value passed to the kernel. In this kernel, the execution time varies according to those argument values even though the global work size does not change. Therefore, it is obvious that *global ws* cannot achieve accurate prediction. On the other hand, since *glls* and *proposal* can find the correlation between the argument value and the execution time, they can predict the execution time based on the argument value.

Even for some kernels whose execution times are not accurately predicted by *glls*, the proposed method can achieve accurate prediction. For example, in the cases of `bitonicSort`, `fastWalshTransform`, and `reorderData` kernels, there are W-type runtime parameters. Thus, it is difficult to build an accurate model if those parameters are included in the explanatory variables. Consequently, the proposed method can outperform *glls* for the prediction of those kernels.

Figure 4.13 shows the comparison results in ARPE where the proposed method outperforms the others. In the figure, the horizontal and vertical axes indicate the benchmarks and ARPE, respectively. Based on profile data classification, the proposed method can use

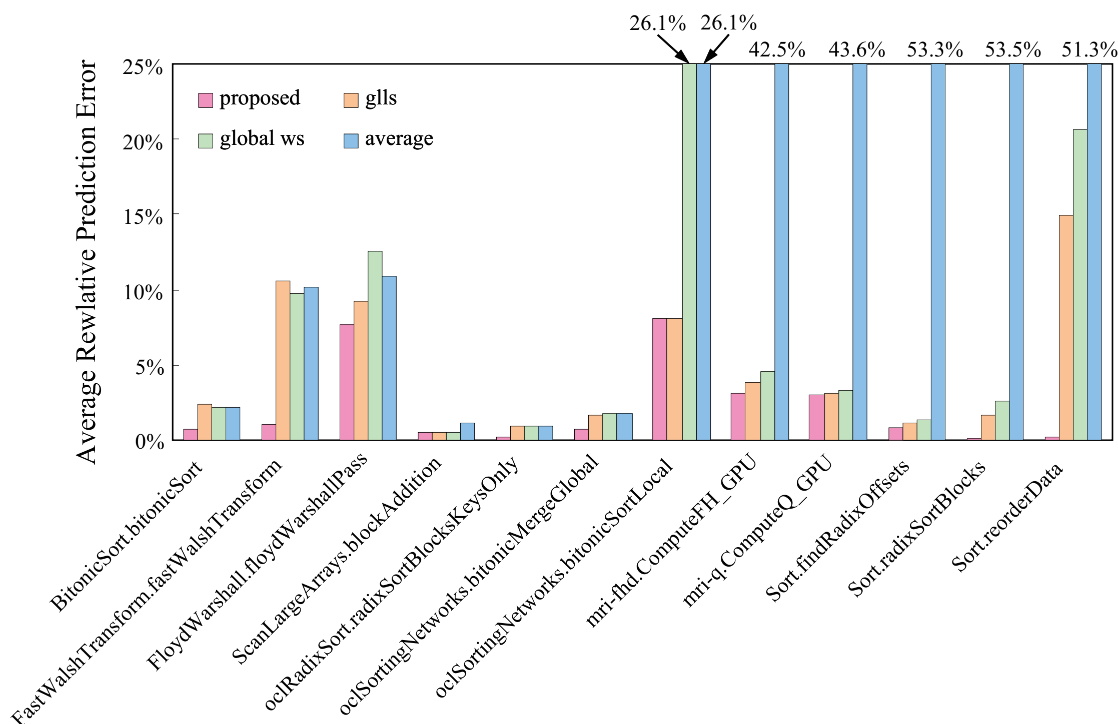



Figure 4.13: Comparison results of ARPE in the cases where the proposed method outperforms the others.

multiple prediction models, each of which is corresponding to one of profile data classes. When a kernel is invoked in a different way, it switches the prediction model so that the most appropriate model is used for prediction. For example, if a W-type parameter changes the memory access pattern in a kernel, the proposed method can select the prediction model of the same access pattern. This is effective to predict the execution time of a kernel running on a GPU, because its performance strongly depends on the pattern. Therefore, the proposed method can achieve more accurate prediction than the others for the benchmark programs.

Classifying profile data is effective to improve the accuracy of any prediction method based on GLLS, because it can remove harmful parameters for GLLS. Accordingly, it is clear that the proposed method is superior to the others from the viewpoint of the accuracy of the prediction in most cases. The superiority will become more remarkable in the cases of practical kernels with more complicated control flows because they usually have more W-type parameters.

The accuracy of the proposed method becomes low in some cases such as the `recalculateEigenIntervals` kernel whose execution time varies irrespective of the

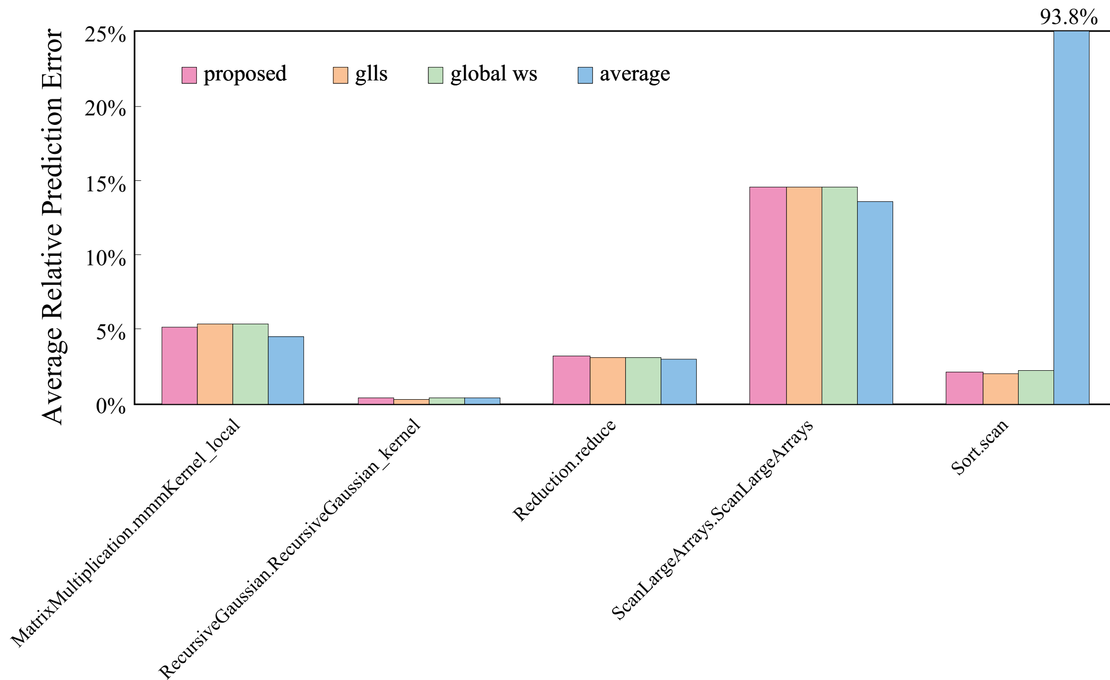



Figure 4.14: Comparison results of ARPE in the cases where the accuracy of the proposed method is less than those of the others.

runtime parameters. This is because the length of a loop in the kernel depends on a value in a memory object. In such a case, runtime parameters do not help the prediction at all. Since it is not practical to look for such a value in a memory object, developing an effective prediction method for such a kernel remains an open problem.

All the methods including the proposed one cannot accurately predict the execution time of the `peak` kernel in the MaxFlops program. This is because the MaxFlops program rewrites and recompiles the source code of the `peak` kernel several times during the execution. Although the function name of a kernel does not change, the computation in the kernel has changed by recompilation. As a result, all the methods fail in the performance prediction. This problem can be solved if a different prediction model is used when a program object[4] is replaced with a new one.

For the above two kernels, `recalculateEigenInterval` and `peak`, the ARPE values of all the prediction methods exceed 100%; no method can accurately predict their performances. In the cases where the proposed method cannot reduce the ARPE for a given kernel, it aborts using GLLS for prediction and uses the simplest *average* method to save the prediction time.

Figure 4.14 shows the comparison results where ARPE of the proposed method is less than those of others. In this figure, the horizontal and vertical axes indicate the benchmarks and ARPE, respectively. For the programs in the figure, the proposed method is less accurate than *glls* and *global ws*. This is mainly because the proposed method is affected by the measurement errors more sensitively. However, the difference in accuracy among those three methods is small. Therefore, this will not be a serious problem in practical uses.

### 4.4.2 Evaluation of the Performance Improvement by Online Task Scheduling

The MonteCarloAsian benchmark in ATIStream SDK[65] is first used to evaluate the effect of online task scheduling among multiple accelerators. This program has many parallel tasks and is optimized by using the proposed dependency analysis method to remove unnecessary synchronizations among commands and false dependencies among tasks. The number of parallel tasks corresponds to the number of steps in the simulation, and it is set to 60 steps in the evaluation.

Table 4.3 shows the experimental setup to evaluate the proposed online task scheduling method. Two different GPUs are used for load balancing, and there is a significant difference in performance. The average execution times of the tasks in `MonteCarloAsian` are 14.9 ms on GeForce GTX 480 and 32.3 ms on Tesla C1060. Therefore, the optimal ratio of loads between these GPUs must satisfy the following condition.

$$N_{\mathrm{GeForceGTX480}} \cdot T_{\mathrm{GeForceGTX480}} = N_{\mathrm{TeslaC1060}} \cdot T_{\mathrm{TeslaC1060}}, \quad (4.5)$$

where $N_{\mathrm{GeForceGTX480}}$ and $N_{\mathrm{TeslaC1060}}$ are the numbers of assigned tasks to the GPUs, and

Table 4.3: Experimental setup to evaluate the proposed online task scheduling method.

| Components | Specifications |
|---|---|
| CPU | Intel Core i7 920 2.66GHz |
| GPU | GeForce GTX 480, NVIDIA Tesla C1060 |
| OS | CentOS 5.5 (Linux 2.6.18) |
| Video Driver | NVIDIA Video Driver 290.10 |
| OpenCL | version 1.1 |

$T_{\text{GeForceGTX480}}$ and $T_{\text{TeslaC1060}}$ are the average execution times of the tasks on the GPUs, respectively. Hence, the optimal ratio of loads between those GPUs is calculated by

$$\begin{aligned} N_{\text{GeForceGTX480}} : N_{\text{TeslaC1060}} &= T_{\text{TeslaC1060}} : T_{\text{GeForceGTX480}} \\ &= 32.3 : 14.8 \quad (\fallingdotseq 0.685 : 0.315). \end{aligned} \tag{4.6}$$

Then, the ideal performance improvement with the optimal ratio is calculated by

$$\begin{aligned} \text{Speedup}_{\text{ideal}} &= \frac{T_{\text{single}}}{T_{\text{load}-\text{balancing}}} \\ &= \frac{T_{\text{GeForceGTX480}} \times N_{tasks}}{T_{\text{GeForceGTX480}} \times 0.685 \cdot N_{tasks}} \\ &\fallingdotseq 1.46, \end{aligned} \tag{4.7}$$

where $T_{\text{load}-\text{balancing}}$ and $T_{\text{single}}$ are the total execution time of a program with load-balancing between the GPUs and that with a single GPU, respectively. $N_{tasks}$ is the number of parallel tasks in the benchmark.

To evaluate the effect of automatic load balancing by online task scheduling, several implementations described below are used.

- *Automatic*: A program that is described to exploit the proposed online task scheduler. In this program, commands to execute tasks are enqueued to a special command queue with online task scheduling.

- *Single*: A program that uses only a single GPU. This program is the same program with *Automatic*, but online task scheduling is disabled.

- *Hand-select*: This is a set of programs in which the ratios of loads between the GPUs are fixed to 11 patterns. In this programs, it is assumed that a user runs these programs and selects the one whose execution time is the shortest.

As the sustained performance of GeForce GTX 480 is higher than that of Tesla C1060, GeForce GTX 480 is always used in the *Single* implementation.

Figure 4.15 shows the evaluation results of these implementations. The ratios of loads between the two GPUs are also indicated in this figure. The execution time is normalized by the execution time of the *Single* implementation. These results indicate that the execution time of the *Automatic* implementation is the shortest, and the ratio of loads is almost optimal.

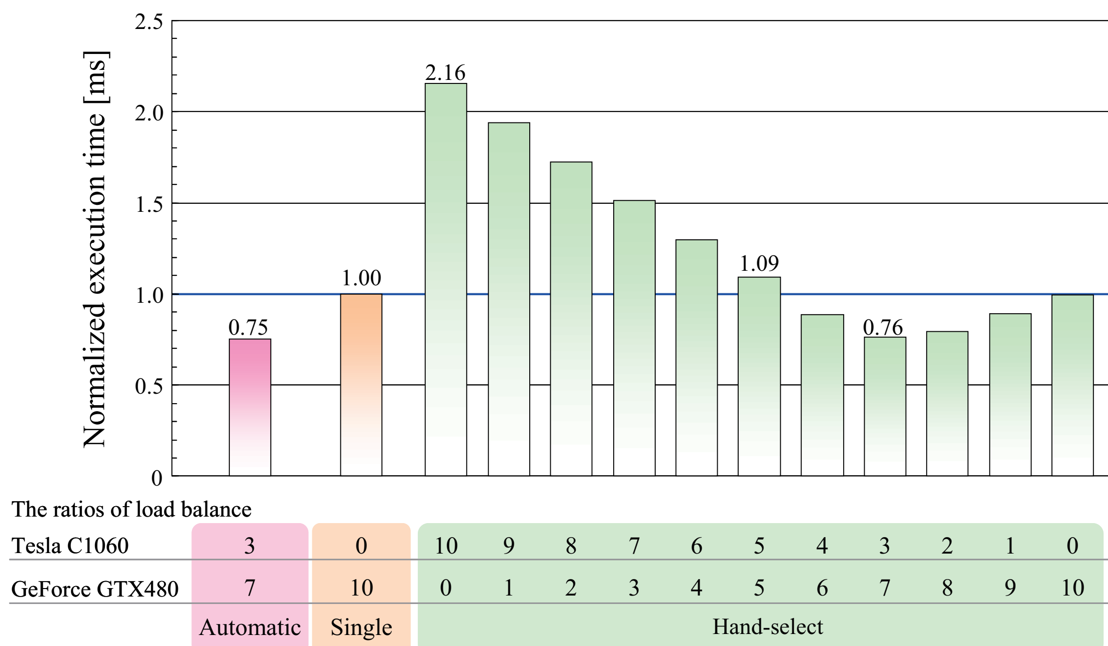



Figure 4.15: The evaluation results of the MonteCarloAsian.

The effect of load balancing can be seen in *Hand-select* implementations. As *Automatic* implementation can accommodate to accidental delay due to device synchronization, the execution time of *Automatic* implementation is slightly shorter than that of the best *Hand-select* implementation.

Moreover, these results show the performance degradation caused by inappropriate load balancing. As there is a big performance difference between GeForce GTX 480 and Tesla C1060, the performance with multiple GPUs degrades if their performance difference is not considered and the GPUs simply execute the same number of parallel tasks. The proposed scheduling method enables a programmer to exploit multiple accelerators without risks of performance degradation even if the programmer does not know the number of available accelerators and the difference in performance among the accelerators.

The ideal performance improvement from the *Single* implementation is 46.0%. However, the actual performance improvement by the *Automatic* implementation is 32.8%. Figure 4.16 shows that the operating times of the GPUs. In the figure, the operating time of the CPU for thread scheduling is also shown. The overhead of scheduling is 6.06% and causes

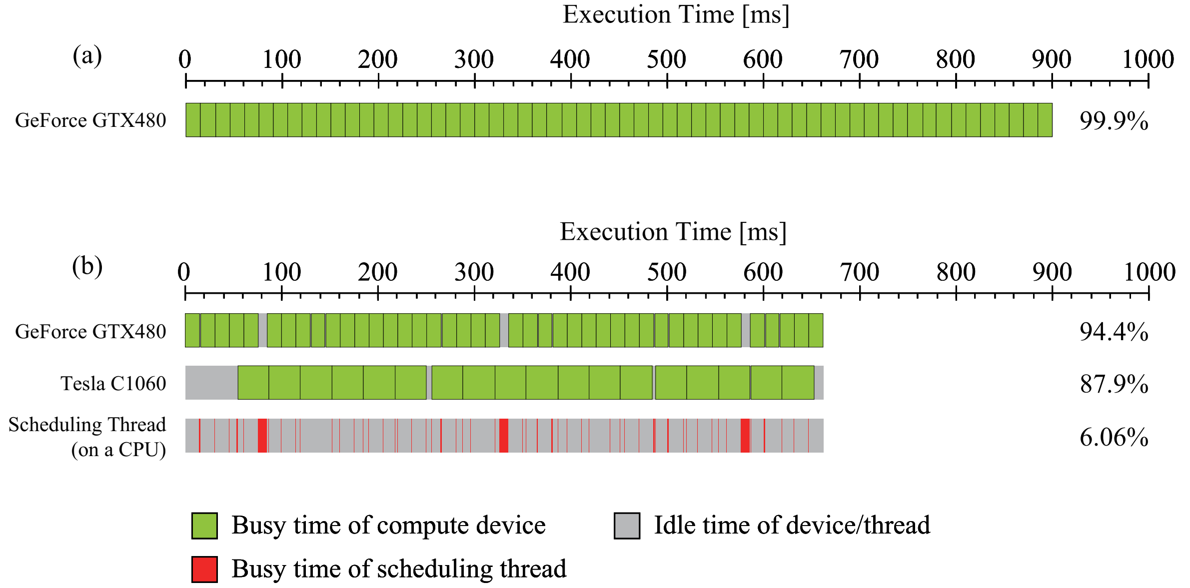



Figure 4.16: The operating rates of GPUs and a scheduling thread. (a) the *Single* implementation. (b) the *Automatic* implementation.

stalls of GPUs. The operating rate of a GPU can be calculated by

$$\eta = \frac{T_{busy}}{T_{total}}, \tag{4.8}$$

where $T_{busy}$ and $T_{total}$ are the busy time of a GPU and the total execution time of the computing, respectively. In the *Automatic* implementation, the operating rates of GeForce GTX 480 and Tesla C1060 are 94.4% and 87.9%, respectively. On the other hand, in the Single implementation, the operating rate of GeForce GTX 480 is 99.9%. These operating rates indicate that online task scheduling method can efficiently use these two GPUs. In this evaluation, 42 tasks and 18 tasks are assigned to GeForce GTX 480 and Tesla C1060, respectively. The speed-up ratio of the *Automatic* implementation in this case is calculated by

$$\begin{aligned} \text{Speedup}_{\text{actual}} &= \frac{T_{\text{single}} \times \frac{1}{\eta_{Single}}}{T_{\text{load-balancing}} \times \frac{1}{\eta_{Automatic}}} \\ &= \frac{T_{\text{GeForceGTX480}} \times N_{tasks} \times \frac{1}{0.999}}{T_{\text{GeForceGTX480}} \times \frac{42}{60} \cdot N_{tasks} \times \frac{1}{0.944}} \\ &= 1.35, \end{aligned} \tag{4.9}$$

where $\eta_{Single}$ and $\eta_{Automatic}$ are the operating rates of GeForce GTX 480 in the *Single* and

*Automatic* implementations, respectively.

This theoretical value is almost the same as the actual speedup ratio observed in the evaluation. In this evaluation, the overhead of online task scheduling affects the sustained performance because the execution time of each task is considerably short. For a practical application of time-consuming tasks, the scheduling overhead will become relatively small. Therefore, the overhead will be negligible in practical use.

The two practical benchmarks are next used to demonstrate the effectiveness of the proposed online task scheduling method. One is a benchmark of the *Building Cube Method* (BCM) [68, 69]. The BCM method divides a computational domain into cubes that can be processed in parallel. Hence, the program using the BCM method has massive parallelism and is promising to be accelerated by multiple GPUs. The benchmark used in this evaluation is only a computation-intensive kernel of the `BCM` benchmark to calculate the pressure of a fluid by the *Red-Black successive over-relaxation* (Red-Black SOR) method. There are 20 parallel tasks in the benchmark that execute two kinds of kernels: 10 parallel tasks execute the `Red` kernel, and the other tasks execute the `Black` kernel. Each task calculates 256 cubes, and the `BCM` benchmark handles 2560 cubes in total.

The other is a benchmark of the *Phase-Only Correlation* (POC) method [70] for image matching. The POC method finds corresponding points between two stereo images to estimate the positions of objects in three-dimensional space. In this program, there are two parallel tasks. Since these tasks are dominant in the execution time of the benchmark, the execution time reduces by executing the tasks in parallel. The size of input images used in the following evaluation is $1280 \times 960$.

Figure 4.17 shows evaluation results of two benchmarks. In this figure, two execution modes are shown: the *Single* mode and the *Automatic* mode. In the *Single* mode, the programs use only GeForce GTX 480. On the other hand, in the *Automatic* mode, the programs use multiple accelerators with online task scheduling. The execution time is normalized by that in the *Single* mode.

The execution time of the `BCM` benchmark in the *Automatic* mode is 14.2% shorter than that in the *Single* mode. The average execution times of the tasks are 0.837 ms on GeForce GTX 480 and 2.066 ms on Tesla C1060, respectively. Hence, the optimal ratio of loads

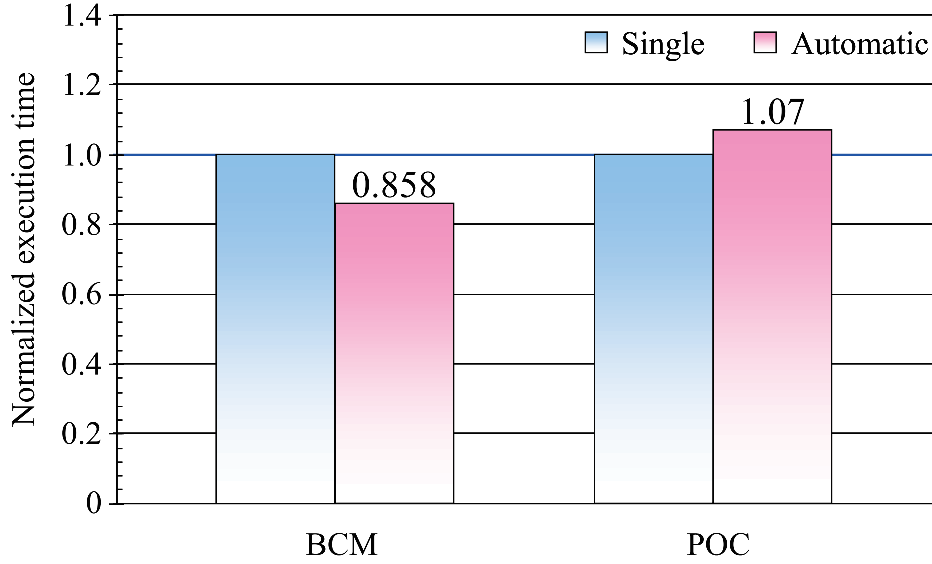



Figure 4.17: Evaluation results of the POC and BCM benchmarks.

between those GPUs and ideal performance improvement is calculated by

$$\begin{aligned} N_{\text{GeForceGTX480}} : N_{\text{TeslaC1060}} &= T_{\text{TeslaC1060}} : T_{\text{GeForceGTX480}} \\ &= 2.066 : 0.837 \quad (\fallingdotseq 0.712 : 0.288), \end{aligned} \tag{4.10}$$

and

$$\begin{aligned} \text{Speedup}_{\text{ideal}} &= \frac{T_{\text{single}}}{T_{\text{load-balancing}}} \\ &= \frac{T_{\text{GeForceGTX480}} \times N_{tasks}}{T_{\text{GeForceGTX480}} \times 0.712 \cdot N_{tasks}} \\ &\fallingdotseq 1.40. \end{aligned} \tag{4.11}$$

However, unlike the `MonteCarloAsian` benchmark, the `BCM` benchmark has a synchronization point between the `Red` and `Black` kernels. Moreover, since the `Black`-kernel tasks use the calculation results of the `Red`-kernel tasks, data transfer is needed to execute the `Black`-kernel tasks on the other GPU. As a result, the operating rates of GPUs decrease. The operating rates of GeForce GTX 480 and Tesla C1060 are 85.2% and 70.2%, respectively. On the other hand, in the *Single* mode, the operating rate of GeForce GTX 480 is 99.5%. In this evaluation, 15 tasks and 5 tasks are assigned to GeForce GTX 480 and Tesla

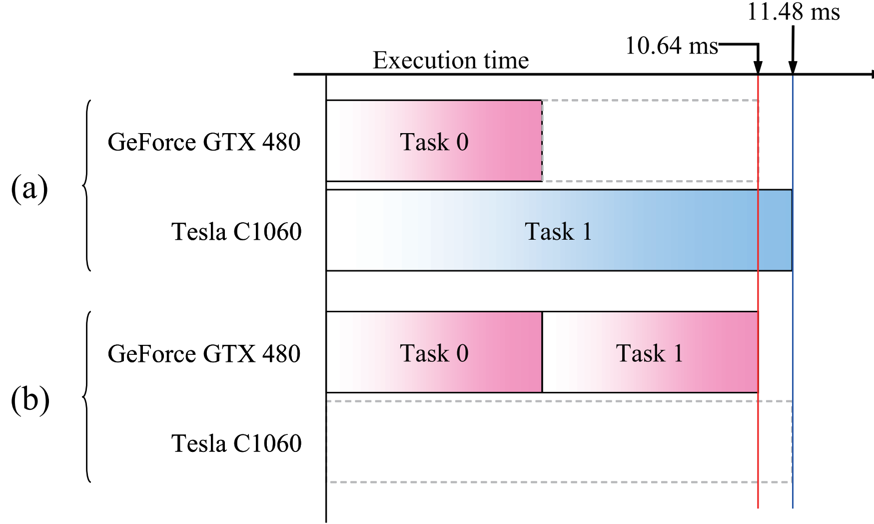



Figure 4.18: Task assignment in the POC benchmark. (a) The case of balancing loads. (b) The case of using only a GPU.

C1060, respectively. From these parameters, the speed-up ratio of the `BCM` benchmark in the *Automatic* mode is conducted.

$$\begin{aligned}
\text{Speedup}_{\text{actual}} &= \frac{T_{\text{single}} \times \frac{1}{\eta_{Single}}}{T_{\text{load-balancing}} \times \frac{1}{\eta_{Automatic}}} \\
&= \frac{T_{\text{GeForceGTX480}} \times N_{tasks} \times \frac{1}{0.995}}{T_{\text{GeForceGTX480}} \times \frac{15}{20} \cdot N_{tasks} \times \frac{1}{0.852}} \\
&= 1.142. \qquad (4.12)
\end{aligned}$$

This theoretical value is equivalent to the speedup ratio observed in the evaluation. From these results, it is demonstrated that the online task scheduling can work to exploit multiple accelerators even if the program has a synchronization point among parallel tasks.

The execution time of the `POC` benchmark in the *Automatic* mode is longer than that in the *Single* mode. The `POC` benchmark has only two parallel tasks and the average execution times of the tasks are 5.32 ms on GeForce GTX 480 and 11.48 ms on Tesla C1060, respectively. Hence, the execution time when the two tasks are executed on GeForce GTX 480 is shorter than the execution time when the loads are balanced, as shown in Figure 4.18. In this case, the online task scheduling method assigns the parallel tasks to only GeForce GTX 480, and the scheduling overhead increases the execution time of the program. Therefore, if there is a certain difference in performance of GPUs, a sufficient number of parallel tasks

are needed for load balancing by the proposed online task scheduling method.

## 4.5 Concluding Remarks

This chapter has proposed an online task scheduling method to realize automatic load balancing among multiple accelerators. In addition, two methods required in the scheduling methods have been proposed: a performance prediction method and a data-dependency and event-dependency analysis method.

Since OpenCL has high programming flexibility and an OpenCL program can be executed on several accelerators, performance prediction of arbitrary OpenCL programs is difficult. The proposed performance prediction method uses the argument values passed to the kernel to classify profile data. Then, the proposed prediction method can remove the harmful effects on linear prediction models, and it can improve the prediction accuracy. Since this prediction method is independent of the architecture of GPUs, it is applicable to any accelerators.

A data and event dependency analysis method is proposed to facilitate task parallelization by visualizing several dependencies among tasks. The proposed analysis method enables programmers to easily find unnecessary data dependencies and synchronization points among tasks. As a result, the proposed method can increase the number of parallel tasks and improve the sustained performance of a program using multiple accelerators.

An online task scheduling method based on performance prediction is then proposed. In this method, a new execution model of OpenCL is proposed to bind a command queue to multiple accelerators. The proposed scheduling method estimates the completion time of an enqueued task for each accelerator. Then, it automatically assigns a task to the accelerator that is expected to complete the task earliest.

The accuracy of performance prediction is evaluated using 126 kernels in several benchmark suites, and it is demonstrated that the proposed prediction method can achieve the highest accuracy in the other evaluated methods. Although the execution time of the task nonlinearly depends on the argument values passed to a task, the proposed method can accurately estimate the execution time by using multiple prediction models.

The effect of online task scheduling is evaluated using the `MonteCarloAsian`, `BCM`, and `POC` benchmarks. The evaluation results show that the proposed scheduling method can appropriately adjust the load balance. As a result, it enables to exploit multiple accelerators. Moreover, the evaluation results of the `POC` benchmark indicate that the proposed online scheduling method needs many parallel tasks in a program to balance loads among multiple accelerators.

The proposed online task scheduling method enables programmers to exploit multiple accelerators without risks of performance degradation. In the future, when the number of available accelerators increases, the proposed scheduling methods will become more important.

# Chapter 5

# Conclusions

In this dissertation, automatic performance tuning methods are explored to exploit the computing power of heterogeneous computing systems. Although heterogeneous computing systems can achieve high performance and high energy efficiency, it is difficult for programmers to appropriately exploit the computing capability of the system without labor-intensive performance tunings. Therefore, this dissertation proposes automatic performance tuning methods to easily exploit heterogeneous computing systems.

A GPU is one of accelerators widely used in heterogeneous computing systems. Although a GPU has a high computing capability, unsuitable task assignment causes serious performance degradation. Moreover, GPUs have architecture-specific features and execution parameters that must be given at execution. Programmers have to perform performance tunings with considering these features to achieve a high performance. However, these performance tunings require knowledge of the GPU architecture. Hence, these performance tunings are difficult and labor-intensive even for expert programmers.

To alleviate difficulties in performance tunings, it is effective to abstract hardware configurations and to automate performance tunings. To this end, a programming framework should automatically apply some optimizations and tuning to a program. Such a framework realizes programming without consideration of the system configuration. Therefore, this dissertation proposes automatic performance tuning methods according to the three following approaches.

- To alleviate the difficulty of appropriate processor selection, this dissertation designs the SPRAT framework consisting of a domain-specific programming language and its runtime environment to automatically select an appropriate processor based on linear

performance prediction.

- To improve performance of an automatically-generated program from a program written in high-level programming languages such as the SPRAT language, this dissertation proposes the two automatic optimization methods to effectively use the memory hierarchy of GPUs and an automatic parameter tuning method to determine the optimal CTA configuration.

- To exploit multiple accelerators without consideration of the number of available accelerators and the difference in their performance, this dissertation proposes an online task scheduling method based on highly-accurate performance prediction.

In Chapter 2, appropriate processor selection is automated to alleviate one of the difficulties in programming for heterogeneous computing systems. To this end, the SPRAT framework is proposed. A SPRAT program is translated to a CPU code and a CUDA code for GPUs, respectively. The SPRAT language enables a programmer to easily write stream processing without considering which processor is used for execution. The SPRAT runtime environment automatically selects an appropriate processor for a task based on performance prediction. As the correlation between the output data size and the execution time can be assumed in the execution model of SPRAT, a linear prediction model is built for each SPRAT program. This prediction model can be also used for energy-aware computing. From the evaluation results, it is clarified that the SPRAT framework enables even a non-expert programmer to benefit from the use of GPUs without performance degradation. Moreover, the SPRAT runtime environment can select an appropriate processor based on energy-aware policy and can optimize energy efficiency by a simple prediction model of energy consumption.

In Chapter 3, architecture-specific optimizations and tuning are automated for high-level programming languages. Firstly, key features to achieve high performance in CUDA are pointed out based on the specification of CUDA and the architecture of GPUs. In CUDA, efficient use of memory hierarchy and the optimal CTA configuration are important to exploit the potential of GPUs. The proposed automatic optimization method finds highly-reusable data elements in streams by exploiting the features of the SPRAT language. An automatically-generated CUDA program is optimized to copy those highly-reusable data elements into the shared memory to improve sustained memory bandwidth. Moreover, an inefficient memory access is converted to two coalesced memory accesses by using the shared memory as a read buffer. A method to automatically find the optimal CTA configuration

based on profiling data is also proposed. Evaluation results indicate that the two performance tuning strategies are effective to improve sustained performance of CUDA programs. It is also demonstrated that the proposed tuning method can automatically find the optimal or suboptimal CTA configuration. Therefore, the two strategies for automatic performance tunings enable programmers to improve performance without knowledge of GPU architectures and the constraints in CUDA.

In Chapter 4, load balancing among multiple accelerators is automated by online task scheduling based on performance prediction. By using a standard programming framework such as OpenCL, a single program can run on any accelerators. It is more important to assign each task for an appropriate accelerator to efficiently use various accelerators. In practical situations, although there may be big differences in performance between accelerators, it is difficult for programmers to know the number of available accelerators and the differences in their performance. Hence, it is necessary to automate load balancing among multiple accelerators in order to exploit the computing capability of a heterogeneous computing system without performance degradation. For online task scheduling, it is required to accurately predict the execution time of an OpenCL program. To improve the prediction accuracy, the proposed performance prediction method uses not only past execution times but also the execution parameters and argument values passed to the tasks. In the execution parameters and/or argument values, there are non-linear parameters whose values are not proportional to the execution time, and the non-linear parameters are harmful for performance prediction by using GLLS. The proposed prediction method uses multiple prediction models to remove the harmful effects of the non-linear parameters. As a result, the proposed prediction method can accurately predict the execution times.

In the online task scheduling method, a task in a program is automatically assigned to an appropriate accelerator that can complete the task earliest. In addition, a data and event dependency analysis method to facilitate removing unnecessary dependencies and synchronization points is proposed. Evaluation results show that the proposed prediction method can improve the accuracy of the prediction even for a program that has non-linear parameters. Evaluation results of online task scheduling indicate that the proposed scheduling method can appropriately adjust the ratio of loads assigned to the GPUs and can improve the sustained performance of the OpenCL programs. Accordingly, it is demonstrated that an automatic load balancing method by online scheduling is effective to exploit multiple accelerators.

In conclusion, this dissertation establishes the proposed automatic performance tuning methods that enable programmers to exploit the computing power of heterogeneous computing systems without labor-intensive performance tunings and performance degradation. The accomplishment of this dissertation opens up the way for programmers to easily use different types of processors in the heterogeneous systems. Moreover, this dissertation indicates the importance of automatic performance tunings for efficient use of heterogeneous computing systems.

Finally, the future work is summarized as follows.

- Efficient use of accelerators in other nodes via a network
  The number of accelerators in a single node is limited by several reasons such as power budget. It is impractical to increase available accelerators in only a single node. Hence, use of accelerators in other nodes via a network is promising to increase available accelerators. Virtual OpenCL [51] has been proposed to use accelerators in other nodes and partially realizes this strategy. However, since the network overhead is too big, the proposed simple method cannot achieve high performance when the remote accelerators are used. Accordingly, online scheduling based on the prediction of network delays is needed to hide the network overheads.

- An extension of the queuing model of OpenCL to improve the quality of scheduling
  In some cases, the quality of online scheduling is degraded because of the constraint that only one command can be enqueued at every API function call. Hence, if a preceding parallel task is already scheduled on an accelerator, a subsequent parallel task cannot overtake the preceding task even if the execution time of the subsequent task can be more reduced on the accelerator than that of the preceding task. In OpenCL specification, the out-of-order scheduling mode is defined but it is not practically realized because of this constraint. To overcome this situation, an extension of the queuing model of OpenCL is needed to enable multiple commands to be enqueued at one API function call. Moreover, a new out-of-order scheduling algorithm is required for the extended queuing model.

# Bibliography

[1] Satoshi Matsuoka. The TSUBAME Cluster Experience a Year Later, and onto Petascale TSUBAME 2.0. In Franck Cappello, Thomas Herault, and Jack Dongarra, editors, Recent Advances in Parallel Virtual Machine and Message Passing Interface, volume 4757 of *Lecture Notes in Computer Science*, pp. 8–9. Springer Berlin / Heidelberg, 2007.

[2] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the Petaflop Era: The Architecture and Performance of Roadrunner. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing, pp. 1–11, 2008.

[3] NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture programming guide version 4.0, 2011.

[4] Khronos OpenCL Working Group . The OpenCL Specification version 1.0 . `http://www.khronos.org/opencl/`, 2012/1/16.

[5] NVIDIA Corporation . Cg Toolkit . `http://developer.nvidia.com/cg-toolkit`, 2012/1/16.

[6] Microsoft corporation . Programming Guide for HLSL . `http://msdn.microsoft.com/en-us/library/windows/desktop/bb509635(v=vs.85).aspx`, 2012/1/16.

[7] Khronos OpenGL Working Group . The OpenGL Shading Language . `http://www.opengl.org/registry/doc/GLSLangSpec.4.20.8.clean.pdf`, 2012/1/16.

[8] GPGPU.org . GPGPU General-Purpose Computation on Graphics Hardware . `http://gpgpu.org`, 2012/1/16.

[9] J.L. Hennessy and D.A. Patterson. *Computer Architecture*. Morgan Kaufmann Publishers, fourth edition, 2007.

[10] Jayanth Gummaraju and Mendel Rosenblum. Stream Programming on General-Purpose Processors. In MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture, pp. 343–354. IEEE Computer Society, 2005.

[11] Xin David Zhang, Qiuyuan J. Li, Rodric Rabbah, and Saman Amarasinghe. A Lightweight Streaming Layer for Multicore Execution. In Workshop on Design, Architecture and Simulation of Chip Multi-Processors, 2007.

[12] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. Computer Graphics Forum, 26(1):80–113, 2007.

[13] Ian Buck et al. Brook for GPUs: Stream Computing on Graphics Hardware. ACM Trans. Graph., 23(3):777–786, 2004.

[14] Michael D. McCool et al. Performance Evaluation of GPUs Using the RapidMind Development Platform. In poster reception at the ACM/IEEE SC06, 2006.

[15] Matthew Papakipos. SC06 GPGPU Course: PeakStream Platform. In the ACM/IEEE SC06 tutorial, 2006.

[16] AMD Corporation. ATI STREAM ATI stream computing user guide version 1.4 beta, April 2009.

[17] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Software Pipelined Execution of Stream Programs on GPUs. In Proceedings of the International Symposium on Code Generation and Optimization (CGO2009), pp. 200–209, 2009.

[18] Michael Wolfe. Design and Implementation of a High Level Programming Model for GPUs. In Proceedings of the Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPHAM2009), 2009.

[19] Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP: A hybrid multi-core parallel programming environment. In Proceedings of the First Workshop on General Purpose Processing on Graphics Processing Units, 2007.

[20] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. Computers, IEEE Transactions on, C-21(9):948 –960, sept. 1972.

[21] Vasily Volkov and James W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In Proceedings of the ACM/IEEE SC08 Conference, November 2008.

[22] Ian Buck, Kayvon Fatahalian, and Pat Hanrahan. GPUBench: Evaluating GPU Performance for Numerical and Scientific Applications. In Proceedings of the 2004 ACM workshop on general-purpose computing on graphics processors (GP2), pp. C–20, August 2004.

[23] S. Che, J. Meng, J. Sheaffer, and K. Skadron. A Performance Study of General Purpose Applications on Graphics Processors. In Proceedings of the First Workshop on General Purpose Processing on Graphics Processing Units, 2007.

[24] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 73–82, New York, NY, USA, 2008. ACM.

[25] Naga K. Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha. A Memory Model for Scientific Algorithms on Graphics Processors. In Proceedings of the ACM/IEEE SC06 Conference, 2006.

[26] Owen Harrison and John Waldron. Optimising Data Movement Rates For Parallel Processing Applications On Graphics Processors. In Proceedings of Parallel and Distributed Computing and Networks (PDCN 2007), February 2007.

[27] Shingo Ito, Fumihiko Ino, and Kenichi Hagihara. A Performance Model for Assisting Development of GPGPU Applications. IPSJ Transactions on Advanced Computing Systems, 48(SIG 13(ACS19)), 2007. (in Japanese).

[28] Bingsheng He et al. Efficient Gather and Scatter Operations on Graphics Processors. In the ACM/IEEE SC07, November 2007.

[29] Pedro Trancoso and Maria Charalambous. Exploring Graphics Processor Performance for General Purpose Applications. In Proceedings of the 8th Euromicro Conference

on Digital System Design, pp. 306–313, Washington, DC, USA, 2005. IEEE Computer Society.

[30] Sara S. Baghsorkhi and Wei mei Hwu. Analutical Performance Prediction for Evaluation and Tuning of GPGPU applications. In Proceedings of the Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPHAM2009), 2009.

[31] Dominik Göddeke et al. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. Parallel Computing, 33:685–699, 2007.

[32] Hiroyuki Takizawa, Hiroki Shiratori, Katsuto Sato, and Hiroaki Kobayashi. SPRAT: A Stream Programming Language with Runtime Auto-tuning. IPSJ Transactions on Advanced Computing Systems (ACS), 1(2):207–220, Aug. 2008.

[33] Kentaro Sano, Takanori Iizuka, and Satoru Yamamoto. Systolic Architecture for Computational Fluid Dynamics on FPGAs. In 2007 International Symposium on Field-Programmable Custom Computing Machines, pp. 107–116, 2007.

[34] HIOKI E.E. CORPORATION. AC/DC POWER HiTESTER 3334 product page. http://www.hioki.com/product/3334/index.html.

[35] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. In Proceedings of the ACM/IEEE SC05 Conference, 2005.

[36] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro, 28:39–55, 2008.

[37] Tianyi David Han and Tarek 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, pp. 52–61, New York, NY, USA, 2009. ACM.

[38] Naruse Akira, Sumimoto Shinji, and Kumon Kouichi. Acceleration Technique of Computational Fluid Dynanics on GPGPU. In IPSJ High Performance Computing Symposium (HPCS2009), pp. 115–122, 2009.

[39] Ryutaro Himeno. Himeno Benchmark . http://accc.riken.jp/HPC/HimenoBMT/, 2012/1/16.

[40] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded Sparc processor. Micro, IEEE, 25(2):21–29, March-April 2005.

[41] Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-Mei W. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. In Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers, pp. 1–15, Berlin, Heidelberg, 2008. Springer-Verlag.

[42] The OpenMP Architecture Review Board. OpenMP Specifications version 3.1. `http://openmp.org/wp/openmp-specifications/`, 2012/1/16.

[43] Satoshi Ohshima, Shoichi Hirasawa, and Hiroki Honda. OMPCUDA : Implementation of OpenMP for GPU. In IPSJ High Performance Computing Symposium (HPCS2009), pp. 131–138, 2009.

[44] Hiroyuki Takizawa, Katuto Sato, and Hiroaki Kobayashi. SPRAT: Runtime processor selection for energy-aware computing. Cluster Computing, 2008 IEEE International Conference on, pp. 386–393, 29 2008-Oct. 1 2008.

[45] NVIDIA Corporation. CUDA C best practices guid version 4.0, 2011.

[46] Ronald L. Rivest Thomas H.Cormen, Charles E.Leiserson and Clifford Stein. *INTRODUCTION TO ALGORITHMS*, pp. 762–766. The MIT Press, Cambridge, Massachusetts 02142, 2 edition, 2001.

[47] NVIDIA Corporation. OpenCL Programming Guide for the CUDA Architecture, 2011. `http://developer.nvidia.com/opencl`.

[48] AMD Corporation. AMD accelerated parallel processing (app) sdk version 2.6, December 2011. `http://developer.amd.com/sdks/AMDAPPSDK/Pages/default.aspx`.

[49] IBM Corporation. OpenCL Development Kit for Linux on Power version 0.3, 2011.

[50] Intel Corporation. Intel OpenCL SDK 1.5, 2011. `http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/`, 2012/1/16.

[51] Amnon Barak and Amnon Shiloh. Virtual OpenCL (VCL) Cluster Platform. `http://www.MOSIX.org`, 2012/1/16.

[52] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for GPU architectures. In PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 105–114, New York, NY, USA, 2010. ACM.

[53] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of PTX kernels. In Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, pp. 3–12, Oct. 2009.

[54] M. Maheswaran, S. Ali, H.J. Siegal, D. Hensgen, and R.F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth, pp. 30 –44, 1999.

[55] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. SIGARCH Comput. Archit. News, 37:152–163, June 2009.

[56] William T. Vetterling William H. Press, Saul A. Teukolsky and Brian P. Flannery. *Numerical Recipes in C++ The Art of Scientific Computing*, pp. 676–694. Cambridge University Press, 40 West 20th Street, New York, NY 10011-4211, USA, 2nd edition, 2005.

[57] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[58] Hironori Kasahara, Motoki Obata, and Kazuhisa Ishizaka. Automatic Coarse Grain Task Parallel Processing on SMP Using OpenMP. In Samuel Midkiff, Jose Moreira, Manish Gupta, Siddhartha Chatterjee, Jeanne Ferrante, Jan Prins, William Pugh, and Chau-Wen Tseng, editors, Languages and Compilers for Parallel Computing, volume 2017 of *Lecture Notes in Computer Science*, pp. 189–207. Springer Berlin / Heidelberg, 2001.

[59] Daisuke Inaishi, Keiji Kimura, Kensaku Fujimoto, Wataru Ogata, Masami Okamoto, and Hironori Kasahara. A Cache Optimization with Earliest Executable Condition Analysis. IPSJ SIG Technical Reports. 計算機アーキテクチャ研究会報告, 98(70):31–36, 1998-08-05.

[60] G. Diamos and S. Yalamanchili. Speculative execution on multi-GPU systems. In Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pp. 1 –12, april 2010.

[61] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Task Scheduling Algorithms for Heterogeneous Processors. Heterogeneous Computing Workshop, 0:3, 1999.

[62] P. Thambidurai E. Ilavarasan. Low Complexity Performance Effective Task Scheduling Algorithm for Heterogeneous Computing Environments. Journal of Computer Science, 3:94–103, 2007.

[63] Dominik Grewe and Michael F.P. O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In CC '11: Proceedings of the 20th International Conference on Compiler Construction. Springer, 2011.

[64] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, and Gordon Woodhull. Graphviz - Open Source Graph Drawing Tools. In Petra Mutzel, Michael Junger, and Sebastian Leipert, editors, Graph Drawing, volume 2265 of *Lecture Notes in Computer Science*, pp. 594–597. Springer Berlin / Heidelberg, 2002.

[65] AMD Corporation. ATI STREAM software development kit version 2.3, April 2010. `http://developer.amd.com/sdks/AMDAPPSDK/downloads/pages/AMDAPPArchive.aspx`.

[66] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU '10, pp. 63–74, New York, NY, USA, 2010. ACM.

[67] Parboil benchmark suites . `http://impact.crhc.illinois.edu/parboil.php`.

[68] Kazuhiro Nakahashi. High-density mesh flow computations with pre-/post-data compressions. In AIAA paper: 17th AIAA Computational Fluid Dynamics Conference, pp. 2005–4876, 2005.

[69] Shun Takahashi, Takashi Ishida, Kazuhiro Nakahashi, Hiroaki Kobayashi, Koki Okabe, Youichi Shimomura, Takashi Soga, and Akihiro Musa. Study of High Resolution Incompressible Flow Simulation Based on Cartesian Mesh. In AIAA paper: 47th AIAA Aerospace Sciences Meeting, pp. 2009–563, January 2009.

[70] Mamoru Miuraa, Kinya Fudano, Koichi Ito, Takafumi Aoki, Hiroyuki Takizawa, and Hiroaki Kobayashi. GPU Implementation of Phase-Based Image Correspondence Matching and Its Evaluation. In GTC Workshop, 2011.