A Computational Model and Algorithms to Utilize GPUs for Discrete Problems

by

Atsushi Koike

Dissertation

submitted to the Department of Informatics

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

SOKENDAI (The Graduate University for Advanced Studies)

September 2015

Committee

Zhenjiang Hu (Chair)	National Institute of Informatics / Sokendai
Kunihiko Sadakane	The University of Tokyo
Kento Aida	National Institute of Informatics / Sokendai
Takayuki Aoki	Tokyo Institute of Technology
Ken-ichi Kawarabayashi	National Institute of Informatics / Sokendai
Takeaki Uno	National Institute of Informatics / Sokendai

Acknowledgments

I would like to thank my Ph.D. advisor in the University of Tokyo, Kunihiko Sadakane for providing a wonderful research environment and sharing his wisdom with me. I really enjoyed doing my ph.D. research and had a magnificent time in Sokendai and the University of Tokyo. I am very grateful to the staff and the students of Mathematical Informatics 2nd and 7th Laboratories in the University of Tokyo for sharing advanced research topics. Their talks are always logical and well-structured, and I learned much from them.

I would like to thank my Ph.D. advisor in Sokendai, Zhenjiang Hu for his enthusiastic support. I had many opportunities to give a talk and obtained various insights through the inspiring discussions. I would like to thank the members of the IPL Seminar, which is organized by Hu's Laboratory, for their helpful support and encouragement. In particular, Kiminori Matsuzaki and Akimasa Morihata taught me many things about Algebra of Programming, and Soichiro Hidaka, Akimasa Morihata, and Shigeyuki Sato gave me insightful and essential comments for my ph.D. research. I am grateful to Le-Duc Tung for taking the ph.D. journey together.

I would like to thank the members of my dissertation committee, Kento Aida, Takayuki Aoki, Ken-ichi Kawarabayashi, and Takeaki Uno for insightful advice.

I would also like to thank Asanobu Kitamoto, Kenichi Miura, Kento Aida, and Michihiro Koibuchi for providing well-prepared classes in Sokendai.

Finally, I would like to thank my parents for always being supportive.

謝辞

まず，東京大学での指導教員である定兼邦彦教授に感謝いたします．素晴らしい研究環境を提供していただき，また，多くのことを教えていただき，本当にありがとうございました．先生のおかげで有意義で楽しい研究生活を送れました．また，東京大学大学院情報理工学系研究科の数理第 2 研究室，数理第 7 研究室の皆様に感謝いたします．様々な最先端の話題を論理的かつわかりやすく話していただき，多くのことを学ばさせていただきました．

総合研究大学院大学での指導教員である胡振江教授に感謝いたします．常に熱心に指導いただき，本当にありがとうございました．研究発表の機会を多く作っていただき，そこで有意義な議論を行えたことにより，多くの知見を得ることができました．また，胡教授が開催されているIPLセミナーのメンバーの皆様からも多くの有意義な助言をいただきました．特に松崎公紀准教授，森畑明昌講師からはプログラムの数理について多くのことを教えていただきました．また，日高宗一郎助教，森畑明昌講師，佐藤重幸博士からは博士論文に関し多くの貴重なコメントをいただきました．ありがとうございました．博士課程の厳しい道のりを共に歩んできたLe-Duc Tungにも感謝します．

本論文をまとめるにあたり，有益なご助言をいただいた審査委員の合田憲人教授，青木尊之教授，宇野毅明教授，河原林健一教授に感謝いたします．

総研大において素晴らしい授業を提供していただいた北本朝展准教授，三浦謙一名誉教授，合田憲人教授，鯉渕道紘准教授に感謝いたします．

最後にいつも自分を暖かく見守ってくれる両親に感謝します．いつも心配をかけてばかりでごめんなさい．

Abstract

Graphics Processing Units (GPUs) were originally designed for efficient processing of graphics, but nowadays, they are widely used for a variety of computation applications due to their powerful computational capabilities. Nevertheless, it is becoming more difficult to extract the optimal performance out of GPUs because it is necessary to take the complicated architectures called SIMT into account to develop fast algorithms. To tackle the issue, we propose a GPU-based computational model and develop some basic algorithms.

Asymptotic complexity analysis is very useful to develop efficient discrete algorithms. We propose a GPU-based computational model that focuses on the analysis. Our model called *AGPU* abstracts the GPU's SIMT architecture using only three parameters and takes account of many factors affecting the performance such as coalescing, bank conflict, multithreading. Moreover we reveal the relations between the AGPU model and other computation models including BSP model and I/O model. Therefore, we can utilize findings on other models for GPU-based algorithms.

We next analyze and develop some basic algorithms for lists including reduction, prefix scan, and comparison sorting. We not only provide efficient GPU-based algorithms, but also show how to develop the algorithms using the AGPU model. We analyze the complexity of existing algorithms and the computational lower bound, and find the performance bottlenecks of the algorithms from the results. Then, we develop new algorithms by removing the bottlenecks. We propose a time and I/O-optimal reduction algorithm for non-commutative operators and an I/O-optimal comparison sorting algorithm. Using our reduction algorithm, we can solve the maximum segment sum problem up to 3.9 times faster than the existing algorithm and our comparison sorting algorithm runs up to 1.9 times faster than the existing algorithm.

概要

GPUは元々はグラフィック処理のための専用プロセッサとして開発されたが，今日では，その高い並列計算能力を活かし，グラフィック処理以外の多くの処理にGPUが活用されている．しかし，アルゴリズムを設計する際，SIMTと呼ばれる独特なアーキテクチャを考慮する必要があるため，最適なパフォーマンスを得ることは非常に難しい．この問題に対処するため，本論文では，GPU向けの計算モデルおよび幾つかの基本アルゴリズムを提案する．

離散アルゴリズムの開発では，アルゴリズムの有用性を見積もるために，一般に計算複雑度の漸近解析が行われる．本論文ではGPUアルゴリズムの漸近解析を行うための計算モデルとしてAGPUモデルを提案する．本モデルは3つのパラメータのみを用いて，GPUのSIMTアーキテクチャを適切に考慮できる．また，AGPUモデルとPRAMモデル，BSPモデル，I/Oモデルとの関係を明らかにする．これにより，GPUアルゴリズムを設計する際に他の計算モデルでの知見を活用することができる．

また，本論文では，リストに対する基本処理であるリダクション，プレフィックススキャン，ソートの計算量解析と新規アルゴリズムの提案を行う．効率的なアルゴリズムを提案するだけでなく，どのように効率的なアルゴリズムを開発するかについても示す．アルゴリズムの計算量および計算量下界を求めることで，アルゴリズムのボトルネックを発見し，ボトルネックを取り除くことにより新しいアルゴリズムを提案する．具体的には，時間計算量およびI/O計算量が最適な非可換演算子でのリダクションアルゴリズム，およびI/O計算量が最適な比較ソートアルゴリズムを提案する．提案されたリダクションアルゴリズムで最大部分和問題を解くと，既存アルゴリズムよりも最大で3.9倍高速であり，また，提案された比較ソートアルゴリズムは既存アルゴリズムと比較し，最大で1.9倍高速である．

Contents

List of Figures xv

List of Tables xix

List of Algorithms xxi

1 Introduction 1

1.1 Background 2
1.2 Our Contributions 3
1.3 Outline 5

2 Preliminaries 7

2.1 Introduction 7
2.2 Computer Architectures 7
2.2.1 Von Neumann Architectures 7
2.2.2 Threads 11
2.2.3 Memory 11
2.2.4 The Classic Performance Equation 13
2.2.5 Parallel Processing for Sequential Programs 14
2.3 Parallel Programming 16
2.3.1 Multithreading 16
2.3.2 Multiprocessors 17
2.3.3 Flynn's Taxonomy 17
2.3.4 Costs of Parallel Computation 19

2.4 GPUs . . . 20
2.4.1 Architectures . . . 21
2.4.2 Multiprocessors . . . 23
2.4.3 CUDA . . . 25
2.5 Parallel Computational Models . . . 28
2.5.1 PRAM Model . . . 28
2.5.2 BSP Model . . . 29
2.6 GPU Models . . . 30
2.6.1 DMM and UMM . . . 30
2.6.2 PEM Model . . . 32
2.7 I/O Model . . . 32
2.8 Short Summary . . . 32

3 Parallel Computation Models for GPUs **35**
3.1 Introduction . . . 35
3.2 AGPU Model . . . 36
3.2.1 Architecture . . . 36
3.2.2 Metrics . . . 40
3.2.3 The Effect of Multithreading . . . 40
3.2.4 Discrepancy Between the AGPU Model and Real Architectures 41
3.2.5 Notation for Pseudo-codes . . . 43
3.3 Relations between the AGPU Model and Other Computational Models 44
3.3.1 PRAM Model . . . 44
3.3.2 Bulk Synchronous Parallel Model . . . 45
3.3.3 I/O Model . . . 45
3.3.4 Multithreading in AGPU . . . 49
3.4 Guideline for Developing Efficient Algorithms Using the AGPU Model 50
3.4.1 Implementations Using CUDA . . . 50
3.5 Short Summary . . . 51

4 Reduction Algorithms **53**
4.1 Introduction . . . 53
4.2 Definition . . . 54

4.3 Reduction as a Programming Framework 54
4.3.1 Lists 54
4.3.2 List Homomorphism 56
4.3.3 Higher Order Functions on Lists 57
4.3.4 List Homomorphism Lemma 58
4.4 Examples of Reduction 58
4.4.1 Maximum Segment Sum Problem 59
4.5 Reduction with Commutative Operators 61
4.5.1 Tree-based Algorithm 62
4.5.2 Cascading Algorithm 66
4.6 Reduction with Non-commutative Operators 69
4.6.1 Tree-based Algorithm for Non-commutative Operators 69
4.6.2 Matrix-based Algorithm 70
4.6.3 Pipeline Algorithm 75
4.7 Summary of Complexities and Multiplicity 80
4.8 Experimental Evaluation 80
4.8.1 Running Time 80
4.8.2 Maximum Segment Sum 81
4.9 Short Summary 82

5 Prefix Scan Algorithms **85**
5.1 Introduction 85
5.2 Definition 86
5.3 Applications of Prefix Scan 86
5.3.1 Stream Compaction 87
5.4 Prefix Scan Algorithms 88
5.4.1 Matrix-based Algorithm 88
5.5 Experimental Evaluation 94
5.6 Short Summary 95

6 Sorting Algorithms **97**
6.1 Introduction 97
6.2 Analyses of Known Parallel Sorting Algorithms 98

6.2.1 Bitonic Sort . . . 98
6.2.2 GPU-Warpsort . . . 100
6.3 Sorting Lower Bound on the AGPU Model . . . 104
6.4 I/O-optimal Sorting Algorithms . . . 105
6.4.1 Overview of the Algorithm . . . 105
6.4.2 Initialization . . . 107
6.4.3 Column-wise Merge . . . 108
6.4.4 Subarray Partition . . . 110
6.4.5 Row-wise Merge . . . 111
6.4.6 The Complexities and the Amount of Memory Used . . . 113
6.4.7 Effect of Multiplicity . . . 114
6.5 Experimental Evaluation . . . 114
6.5.1 Parameter Tuning . . . 114
6.5.2 Comparison with Thrust . . . 116
6.6 Short Summary . . . 116

7 Conclusion **119**
7.1 Summary of the Dissertation . . . 119
7.2 Future Work . . . 120

Bibliography **123**

List of Figures

2.1 Block diagram of a simple computer architecture 8
2.2 Work flow of the the instruction that loads $a[2]$ in the memory unit. Some modules are omitted. Dotted boxes represent registers or memory units and (1), (2), . . . , (5) represent the stage numbers. 10
2.3 An example of pipelining. 15
2.4 An example of superscalar. 15
2.5 Examples of UMA architectures. 18
2.6 Examples of NUMA architectures. 18
2.7 Brief sketch of GPU architectures 22
2.8 Brief sketch of the multiprocessor 22
2.9 An example of shared memory banks 23
2.10 SIMT 24
2.11 An example of multithreading 25
2.12 CUDA overview 26
2.13 CUDA's hierarchy of threads 27
2.14 How to emulate the grid using a GPU 27
2.15 PRAM model 28
2.16 BSP model 29
2.17 DMM and UMM 31
2.18 Memory access on the DMM and the UMM. When four cores access addresses 0, 1, 5, and 10, the DMM takes $\ell + 1$ unit times and the UMM takes $\ell + 2$ unit times, where ℓ is the latency of the memory access. . . 31
2.19 I/O model 33

3.1 The architecture of AGPU model 37
3.2 Examples of global memory accesses. Each block stores four words. (a) All instructions coalesce. (b) These instructions do not coalesce because the words are spread out among four blocks. 38
3.3 Examples of shared memory accesses. Each column of the shared memory unit represents a memory bank, and each cell can store one word. (a) Bank conflicts do not occur since no banks are accessed by multiple cores. (b) The instruction is divided into two instructions since the second column is accessed by two cores (bank conflict). . . . 39
3.4 BSP model simulated by the AGPU model 46

4.1 An example of the merge operation for the maximum segment problem 60
4.2 Outline of tree-based reduction algorithm. 62
4.3 The procedure of each block for tree-based reduction algorithm. 63
4.4 Input sequence arranged as a matrix with p columns. 67
4.5 The procedure of each block for modified tree-based reduction algorithm. 70
4.6 Input sequence arranged as a matrix with b columns. 72
4.7 Input sequence arranged as a matrix with b rows. 73
4.8 Examples of memory allocation for the matrix-based algorithm; (a) four-way bank conflicts occur, (b) no bank conflicts occur. 75
4.9 Memory assignment of the pipeline algorithm at $b = 4$. 76
4.10 An example of the procedure of the pipeline algorithm. 76
4.11 An example of the shared memory layout to avoid bank conflicts. . . . 77
4.12 Running time of several reduction algorithms. 81
4.13 Running time of the MSS algorithms. 82

5.1 An example of stream compaction 87
5.2 The process to calculate stream compaction 88
5.3 The outline of prefix scan algorithm 89
5.4 An example of the data alignment 90
5.5 Bandwidth of the prefix scan algorithm with varying number of row in the matrix 95

6.1 The bitonic sorting network for 8 elements 99

6.2 Explanatory diagram showing how to merge two sorted sequences A, B 101
6.3 An example of the ranks for 8 distinct elements 101
6.4 The proof of the correctness of the process that merges two sorted sequences. The gray-colored regions represent elements in the regions are already copied to the shared memory. 102
6.5 Global memory accesses to merge eight sorted sequences; (a) GPU-Warpsort, (b) our algorithm. In this example, our algorithm merges eight sequences at a time. In the GPU-Warpsort, the number of the global memory accesses for each element is six, whereas the number is two in our algorithm. 106
6.6 Column-wise and Row-wise merge 107
6.7 A heap used in the merge process 108
6.8 The number of the global memory accesses for each value of d 115
6.9 Sorting rate for each value of d 115
6.10 Sorting rate for our algorithm and Thrust comparison-based sorting . . 116
6.11 The number of global memory accesses for our algorithm and Thrust . 117

List of Tables

2.1 Comparison of each generation of GPUs. 21

4.1 Complexities and multiplicity of reduction algorithms on AGPU(p, b, M). Here n is the number of elements to be reduced. We assume $p = o(n)$. Cascading cannot be used with non-commutative operator, whereas the others can. 80

5.1 Complexities and multiplicity of the prefix scan algorithm adopted by Thrust on AGPU(p, b, M). Here n is the number of input elements. We assume n is much larger than the number of cores p. 91

6.1 Complexities of comparison-based sorting algorithms on the AGPU model. Here n is the number of elements to be sorted, p is the number of total cores, b is the number of cores in a multiprocessor, M is the size of the shared memory in a multiprocessor. We assume $n = \Omega(b^2)$. . 98

List of Algorithms

4.1 Calculate reduction using the tree-based algorithm 64
4.2 Calculate reduction using the cascading algorithm 68
4.3 Calculate reduction using the modified tree-based algorithm 71
4.4 Calculate reduction using the matrix-based algorithm 74
4.5 Calculate reduction using the pipeline algorithm 78
4.6 Calculate local reduction using the pipeline algorithm 79
5.1 Calculate prefix scan 92
5.2 Calculate block prefix scan 93
5.3 Calculate block prefix scan (Continued) 94

1 Introduction

Graphics Processing Units (GPUs) were originally designed for efficient processing of graphics, but nowadays, they are widely used for a variety of computation applications due to their powerful computational capabilities. This approach is known as General-purpose GPU (GPGPU) or GPU computing. Nevertheless, it is becoming more difficult to extract the optimal performance out of GPUs because it is necessary to take the complicated architectures into account to develop fast algorithms. It is a serious problem especially for discrete algorithms because the memory access of the algorithms tends to be done irregularly and it prevents the algorithms from utilizing the computational capabilities of GPUs.

This dissertation aims to make it possible to easily develop discrete algorithms that fully utilize the computational capability of GPUs. To reach the goals, we first propose computational models that abstract the essence of GPU architectures. The model makes it easy to evaluate the efficiency of GPU-based algorithms. Moreover, we can figure out the computational lower bound of various problems using the model, which is useful to find the bottleneck of algorithms. After that, we develop efficient algorithms for some basic problems using our model. These algorithms are useful to design effective discrete algorithms.

1.1 Background

Parallel architectures are becoming more important as processor clock speeds are beginning to reach a limit. In the near future, performance growth will be driven more and more by increased parallelization [1]. If the performance growth is driven by clock speeds, most applications benefit from the growth without any changes. However, in order to exploit the improved parallel architectures, new parallel algorithms are required.

GPUs are used for a variety of parallel computation applications because they are equipped with high memory bandwidth and high parallelism. The memory bandwidth has reached 337GB/s and the number of cores per device has reached 3072 in the latest commercial model from NVIDIA. GPUs are also utilized by a lot of supercomputers, including Titan, which was named No.1 on the Top500 list in November 2012. Moreover, GPUs operate with significantly low power consumption. On the Green500 list [2] in November 2014, nine of top 10 supercomputers operate with GPUs.

Since GPUs have special architectures for efficient processing with many cores, we have to consider the architectures carefully to develop fast algorithms. NVIDIA [3] provides a parallel computing platform and programming model called Compute Unified Device Architecture (CUDA). Although it enables us to develop programs that can be executed on various GPU architectures, this model is less useful to obtain the optimal performance.

In order to estimate the efficiency of sequential discrete algorithms, we often analyze asymptotic complexities of the algorithms on Random-Access Machine (RAM) model. Although we cannot estimate the actual running time from the complexities, it is very useful for some reasons. Firstly, the complexities are independent from the specification of actual machines because the RAM is a unifying abstracted machine for all sequential machines. Moreover, this indicates that each problem has the computational lower bound. Based on comparison of complexities with the lower bound, we can find performance bottlenecks of algorithms. Secondly, we can analyze the efficiency of algorithms without executing programs. Thirdly, we can predict how parameters affect the performance. Consequently, we can focus on designing algorithms separately from implementing and tuning the codes.

On the other hand, no unifying machines for all parallel machines exist because parallel machines have a wide variety of architectures. Parallel Random-Access Machine (PRAM) [4] model, which consists of multiple cores and a single shared memory unit, is the standard computational model for the design of parallel algorithms. However, algorithms developed on the model do not always show good performance on GPUs because the PRAM is substantially different from actual GPU architectures. For example, b GPU-cores in a multiprocessor can read contiguous b elements in the shared memory in parallel, namely, the complexity is 1 as is the case with the PRAM models. However, if b elements are not stored contiguously, the cores cannot always read them in parallel, namely, the complexity varies from 1 to b. Thus, parallel computational models specialized in GPUs are required.

Some computational models for GPUs have been proposed [5, 6]. However, existing models assume that a GPU is a SIMD machine. Therefore, we cannot take account of multithreading, which is one of the most important features of GPUs. Moreover, the relations between the models and other computational models are not clear.

On the other hand, some models [7, 8] aim to predicate the actual running time on GPUs. Using the models, we can evaluate the running time without executing programs on real GPUs. It is useful to tune the codes.

1.2 Our Contributions

In this dissertation, we propose a novel parallel computational model called AGPU. The AGPU model abstracts the GPU's SIMT architecture using only three parameters. The AGPU model focuses on analyzing asymptotic computational complexities of GPU-based algorithms. The purpose of the analyses using the AGPU model is to grasp where the bottleneck for performance is. Therefore the AGPU model is simple and able to take account of a lot of factors affecting the performance such as coalescing, bank conflict, multithreading. Complexities on GPUs depend on device specifications, but they are within a constant factor of complexities on the AGPU model. Moreover, the AGPU model has a lot of relations to other existing models, which is useful for designing efficient algorithms.

We next analyze and develop some basic algorithms for lists including reduction, prefix scan, and comparison sorting. We not only provide efficient GPU-based algorithms, but also show how to develop the algorithms using the AGPU model. We analyze the complexities of existing algorithms and the computational lower bound in order to find the performance bottlenecks of the algorithms. Then, we develop new algorithms by removing the bottlenecks.

First, we analyze two standard reduction algorithms [9]; the tree-based algorithm and the cascading algorithm, and give the evidence that the cascading algorithm runs faster than the tree-based algorithm. Namely, the time complexity of the tree-based algorithm is larger than that of the cascading algorithm. Though the cascading algorithm has the optimal time and I/O complexities, it cannot be used for non-commutative operators. We therefore proposed the pipeline algorithm. It also has the optimal time and I/O complexities. Many problems can be calculated as reduction with a non-commutative operator. Maximum segment sum problem is one of the problems and it is widely used for practical applications. The pipeline algorithm solves the maximum segment sum problem up to 3.9 times faster than the tree-based algorithm and up to 29 times faster than the sequential algorithm on CPU. In addition, we analyze the matrix-based algorithm to show the power of the AGPU model. Although the algorithm has the optimal time and I/O complexities on SIMD architectures, the algorithm is slow on GPUs. We can easily find the reason using the AGPU model; the reason is the small multiplicity. Then, we measure the actual running time of the algorithms and check we can obtain the expected performance.

Next, we analyze the time and I/O complexities of the fastest prefix scan algorithm [10] using the AGPU model. The algorithm has a tuning parameter α. We show that the parameter makes tradeoff between the time complexity and the multiplicity. Then, we measure the actual running time of the algorithm with various parameter values on the real GPUs and check we can obtain the expected performance.

Lastly, we discuss comparison sorting algorithms. Though GPU-Warpsort [11] is one of the fastest comparison sort algorithms, the I/O complexity is larger than the lower bound. We therefore propose an I/O optimal comparison sorting algorithm. We show our algorithm is fast not only in theory but also in practice. Our comparison sorting algorithm runs up to 1.9 times faster than GPU-Warpsort.

1.3 Outline

Chapter 2 presents some basic preliminaries on computer architectures, parallel programing, and GPUs. In Chapter 3, we describe our computational model AGPU. We explain the architecture and the metrics to evaluate GPU-based algorithms. We also discuss the relations between the AGPU model and other computational models. Chapter 4 deals with reduction algorithms. After analyzing the complexities of existing algorithms, we give a novel and efficient algorithm for reduction with non-commutative operators. Chapter 5 deals with prefix scan algorithms. We prove the algorithm has a tradeoff between the time complexity and the effect of multithreading on the AGPU model, and check that the actual running time shows the same tendency. Chapter 6 deals with comparison sorting algorithms. We show the I/O complexity of existing algorithms are not optimal and develop a new algorithm that has the optimal I/O complexity. Chapter 7 concludes the dissertation.

2 Preliminaries

2.1 Introduction

In this chapter, we provide some preliminaries, which are necessary for later discussion. After giving a brief introduction to computer architectures, we explain parallel programming. Then we present GPU architectures. After that, we explain some parallel computation models and GPU models. Finally, we introduce I/O model [12], which is useful to evaluate the number of I/Os.

2.2 Computer Architectures

2.2.1 Von Neumann Architectures

Digital computers are often characterized by the ability to execute programs that are stored in memory. These computers can execute various user programs without any reconstruction of the hardware. They are called the *Von Neumann architectures* because

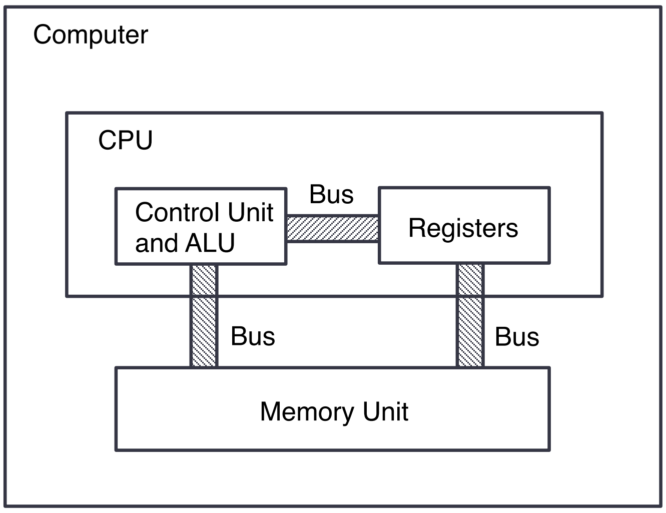


Figure 2.1: Block diagram of a simple computer architecture

this concept was summarized by Von Neumann [13] during development of EDVAC.

We first introduce some technical terms of the Von Neumann architectures. For more details, please refer to the standard textbook by Patterson and Hennessy [14]. In order to introduce some terminologies, we consider a simple computer that consists of one *central processing unit (CPU)* and one *memory unit*. Each byte in the memory unit has a distinct address to access it. Programs and data are stored in the memory unit. We do not consider how to copy them to the memory unit. The CPU consists of an *arithmetic logic unit (ALU)*, *registers*, and a *control unit*. The ALU carries out arithmetic and logic operations. The registers stores data used by the control unit and the ALU. The control unit sends control signals to the ALU, the registers, and the memory unit. Figure 2.1 illustrates the architecture.

Transmission paths between components are called *buses*. Buses fall into three types: *data buses*, *address buses*, and *control buses*. The data buses transfer data to which the ALU carries out the arithmetic and logic operations. The address buses transfer memory addresses in the memory unit. The control buses transfer signals to control the ALU, the registers, and the memory unit. In general, if all data buses and address buses consist of at least 64 parallel electrical wires, the *word* size of the computer is 64-bit. In other words, a 64-bit computer can always transfer a word of 64 bits in parallel. The voltage of a wire represents a bit value. If the voltage is larger than some threshold, it represents 1, otherwise, it represents 0.

A program is a sequence of instructions. Each instruction fetched by the CPU consists of a *operation code* and *operands*. The operation code indicates an operation the CPU carries out, and the operands indicate the addresses of the words to which the operation is performed. After the computer is booted, the CPU first reads a predetermined address to fetch the first instruction, and executes the instruction. After that, it fetches the next adjacent instruction unless the current instruction is the *jump instruction*. If the current instruction is the jump instruction, the CPU fetches the instruction whose address is specified by the operand of the jump instruction. The CPU has a dedicated register to store the address of the current instruction. It is called a *program counter (PC)*. If the size of instructions is the same as the word size (64 bits = 8 bytes), the value of PC is incremented by eight after executing every instruction except the jump instruction.

The computer runs in synchronization with a clock. Each bit in the registers stores a value by keeping the output voltage of a circuit called flip-flop, and the voltages can be updated once at every tick of the clock. We can consider the voltages represent a state of the computer, and the execution of programs proceeds by repeatedly updating the voltages.

Each instruction takes several clock cycles to execute. For example, we consider an implementation of "*load*" instruction. Suppose the computer loads 64-bit element $a[2]$ in the memory unit to register R1, where the address of the array ($\&a[0]$) is stored at register R2 and the value $16 = 8(bytes) \times 2$ is stored at register CONST16. The execution is divided into five stages, each of which is executed in one clock cycle.

1. Fetch The instruction pointed by the PC is transferred to register IR.

2. Decode The control unit reads register IR and calculates control signals to execute the load instruction. After that, it writes the values to registers. Namely, the control signal to the ALU is set to "addition", two operands of the ALU are set to R2 and CONST16, and the final destination address of $a[2]$ is set to R1.

3. Execute Due to the register values above, the ALU outputs the addition of R2 and CONST16, and the output value ($\&a[2]$) is written to register ALUOUT.

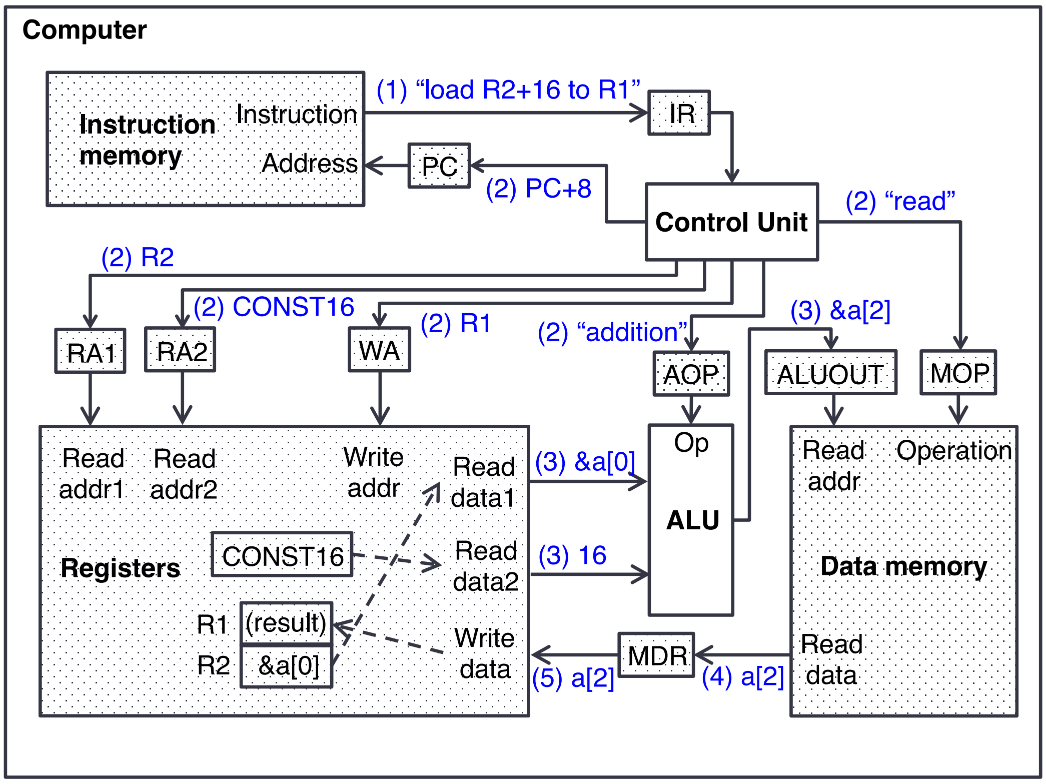


Figure 2.2: Work flow of the the instruction that loads $a[2]$ in the memory unit. Some modules are omitted. Dotted boxes represent registers or memory units and $(1), (2), \ldots, (5)$ represent the stage numbers.

4. Data Access The value stored in the address pointed by register ALUOUT is transferred to register MDR.

5. Write Back The value stored at MDR is transferred to the final destination address R1.

Figure 2.2 illustrates the work flow. In this figure, some modules are omitted. Dotted boxes represent registers or memory units. Note that this implementation takes much time to execute the "Data Access" stage in practice. Since each stage basically needs to be done in one clock cycle, the clock cycle becomes large.

Other instructions can also be executed in at most five clock cycles. A set of instructions supported by a computer is called an *instruction set*, or an *instruction* *set architecture (ISA)*. The set includes arithmetic and logic operations and memory accesses in general.

When the computer sequentially executes many instructions, a technique called pipelining improves the performance. We will discuss it later.

2.2.2 Threads

As can be seen from the previous section, the state of the computer is determined by the values of the registers including the program counter. The computer can execute another program by evacuating the state of the current program and loading the state of the other program. This is called a *context switch*, and each state to be switched is called a *context* in this dissertation. Computers usually support the context switch.

In many cases, one context is assigned to a program called *operating system (OS)*, and the OS manages the other contexts in cooperation with the computer hardware. Contexts managed by the OS are called *threads*. *Multithreading* is defined as dividing a program into multiple threads. *A Sequential program* is defined as a program that consists of a single thread. If the CPU executes multiple threads utilizing the context switch, we say that the threads are *concurrently* executed.

In some architectures, the contexts are managed by the hardware. In this case, the mechanism is called *hardware multithreading*.

Using multiple threads, the computer can utilize the hardware resource more efficiently in general. For example, when a thread waits for something, the computer can execute another thread.

2.2.3 Memory

Since memory access instructions take more time than arithmetic instructions in general, using appropriate memory leads to high performance of computation. Since memory systems have a tradeoff between the access speed and the amount, the registers and the memory unit use different types of memory. In many cases, the register is implemented from SRAM to achieve fast memory access, and the memory unit is implemented from DRAM to obtain large amount of memory. We first explain SRAM and DRAM, and then, we introduce caches to improve the performance.

SRAM

SRAM is memory that uses a circuit called flip-flop to store one bit. It provides fast memory access, but low capacity. It is a volatile memory, that is, data in the memory is lost when the computer shuts down. SRAM is mainly used for the registers.

DRAM

DRAM is memory that uses capacitors, each of which stores one bit. It provides larger capacity than SRAM, but the memory access is slow. DRAM is also a volatile memory. Usually, DRAM works with different frequency from the CPU. During the past decade, DDR SDRAM is mainly used. DDR SDRAM can transfer a set of bits twice per clock cycle. For example, when the clock frequency is 2.6GHz and the bandwidth is 320 bits, the transfer rate is $2.6 \times 320 \times 2 = 1664$(Gbits/sec) = 208(GB/sec). DRAM is mainly used for the memory unit. To obtain large bandwidth, a DRAM unit often consists of several memory banks.

When a CPU accesses the DRAM, the CPU sends the control signal to DRAM. DRAM requires time to start the data transfer. This waiting time is called *latency*, and it significantly affects the performance.

A DRAM bank consists of multiple rows. When a CPU accesses a row, the CPU has to activate the row in advance. The read instruction comprises the following three steps:

1. Activate The row that the data belong to is activated,

2. Read The data are transferred to the CPU,

3. Precharge The row that the data belong to is deactivated.

The activation takes time. Moreover, unless the row is precharged, other rows cannot be activated. The precharge also takes time.

To reduce the latency, DRAM usually supports *burst access*. Thanks to the burst access, several contiguous words in the same row can be transferred by an instruction, where the activation and the precharge are done only once.

Caches

Many architectures equip *caches* to reduce the memory access latency. Caches can store small amount of data originally stored in the memory unit. Caches are usually implemented from SRAM. The current computer systems equip a variety of caches. When we emphasize the fact that the cache is for the main memory, it is called an *L1 cache*. A computer often has two separate caches for instructions and data. A cache for instructions is called an *instruction cache*, and a cache for data is called a *data cache*.

Experientially, programs have two types of localities:

Temporal locality if an element is accessed, it tends to be accessed again soon.

Spatial locality If an element is accessed, elements close to the element tend to be accessed soon.

Caches take advantage of the property. The accessed element and elements close to the element are stored in the cache. Namely, the addresses of the memory unit are divided into blocks with contiguous elements, and the block that the element belongs to is transferred to the cache. This transfer can be effectively done using burst access of DRAM. Moreover, usually, the CPU does not control this transfer. Instead, the dedicated module called a *DMA controller* controls it. In other words, the CPU does not have to wait for the transfer to be complete.

2.2.4 The Classic Performance Equation

The running time of a program is written as follows:

$$\text{Running time} = \frac{\#(\text{Clock cycles})}{\text{Clock frequency}},$$

where #(Clock cycles) represents the number of clock cycles required to execute the program. We define *Clock cycles per instruction (CPI)* as an average number of clock cycles that a computer requires to execute one instruction. Then, we have

$$\#(\text{Clock cycles}) = \#\text{Instructions} \times \text{CPI},$$

where #Instructions represents the number of instructions fetched by the computer. Note that the CPI depends on both the computer and the program. Therefore, we have

$$\text{Running time} = \frac{\#\text{Instructions} \times \text{CPI}}{\text{Clock frequency}}.$$

Conventionally, software developers focus on reducing the number of instructions of sequential programs, and hardware developers focus on reducing the CPI and increasing the clock frequency. In spite of the fact that parallel processing is useful to reduce the CPI especially for multithreaded programs, hardware developers have applied parallel processing only to sequential programs. However, this approach is beginning to reach a limit. Nowadays, software developers have to provide multithreaded programs, to which hardware developers apply parallel processing.

We first explain parallel processing for sequential programs, and then, introduce some techniques for multithreaded programs. From now on, when the CPI is reduced to $1/x$, we say the improvement of the CPI is x.

2.2.5 Parallel Processing for Sequential Programs

Pipelining

If each stage of the instructions is computed by a distinct module, the stages can be executed in parallel. This technique is called *pipelining*. Figure 2.3 shows an example. If the execution of a instruction is divided into five stages, the improvement of the CPI is up to five.

Each stage always needs to be executed in one clock cycle to obtain the best performance. The situation when a stage is not ready to execute is called a *hazard*. For example, when an instruction refers an element written back by the previous instruction, the CPU has to wait for completion of the previous instruction. Thus, a hazard occurs. This is called a *data hazard*. Another example is a *control hazard*, which occurs when the address of the instruction fetched next depends on the result of the current instruction. When a hazard occurs, the CPU has to wait until the stage becomes ready to execute. It is called a *pipeline stall*.

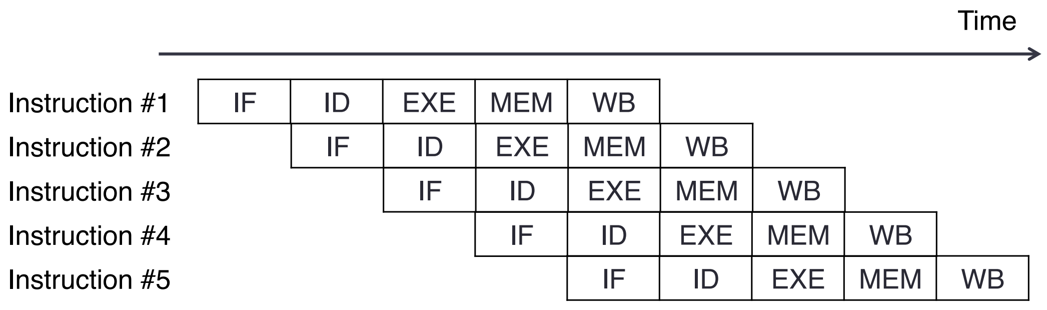


Figure 2.3: An example of pipelining.

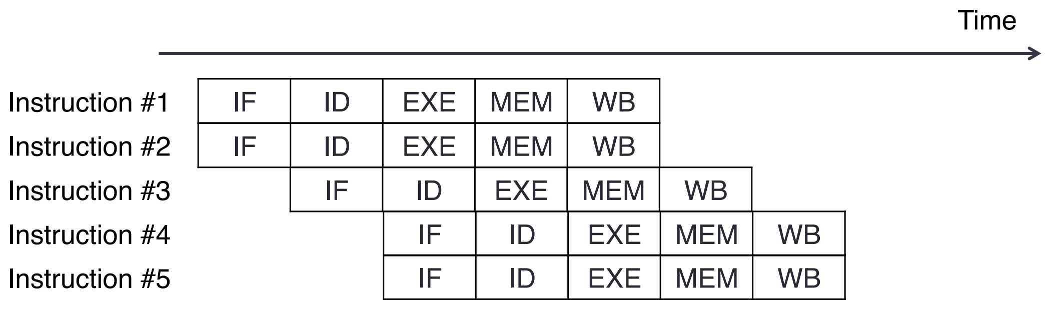


Figure 2.4: An example of superscalar.

Superscalar

If the word size of the computer is 64 bits and the length of a instruction is 32 bits, the CPU can fetch two instructions in parallel. Similarly, if the ALU can carry out operations to two 64 bits elements and the data size of the input elements is 32 bits, the ALU can carry out two operations in parallel with a slight modification of the ALU. Thus, in some cases, the CPU can execute multiple instructions in parallel by taking advantage of the long bus width. This technique is called a *superscalar*. Figure 2.4 shows an example. In this example, a pair of the instructions #1 and #2, and a pair of instructions #4 and #5 are executed in parallel. If the CPU can execute s instructions in parallel, the improvement of the CPI is s. However, the number of elements executed in parallel heavily depends on the program. When the maximum number of instructions executed in parallel is s, we say the CPU has s slots.

SIMD

When we deal with vectors, the same operation is often applied to many distinct elements. If the CPU has a wide data bus and equips many ALUs, the operation can be processed in parallel. This technique is called *SIMD*, which stands for single instruction stream, multiple data stream. This is slightly similar to the superscalar, but in this case, one instruction is applied to many elements. These instructions are called *SIMD operations*. When the maximum number of elements executed in parallel is s, we say the CPU has s SIMD lanes. The improvement of the CPI is up to s. As explained later, GPU architectures also utilize SIMD operations.

SIMD operations that access the memory unit are efficient in general. If the CPU accesses contiguous elements in the memory unit, the CPU can utilize the burst access of the DRAM, which reduces the latency of the memory access. In GPU programming, it is called *coalesced memory access*.

2.3 Parallel Programming

In this dissertation, multithreaded programs are called *parallel programs*. In this section, we introduce some techniques to improve the performance of parallel programs. We first explain how a single CPU executes parallel programs. Then we explain computers with multiple cores. After that, we categorize computer architectures by some criteria.

If parallel programs are executed by computers that connect with each other over a network, it is called *distributed computing*. This dissertation does not deal with distributed computing.

From now on, CPUs are called *cores* because "CPU" is an ambiguous word. For example, "CPU" also represents the device with multiple CPUs.

2.3.1 Multithreading

Multithreading is a technique to increase utilization of a core by executing multiple threads concurrently. When a thread waits for something, the core executes another thread. Thus, the core keeps active. This technique reduces the CPI. A concrete example will be explained in Section 2.4.2.

Multithreading is categorized into two main types: coarse-grained multithreading and fine-grained multithreading. In coarse-grained multithreading, the core switches between threads only when the active thread waits for a long time due to some reasons such as communicating with external modules. When another thread is executed, the current pipeline is emptied. It is easy to implement, but the performance improvement is limited. In fine-grained multithreading, the core can switch between threads at every clock cycle. In other words, the core executes one of the active threads at every clock cycle. Since the core has to maintain the states of all threads, it is difficult to implement, while it significantly improves the performance because It can hide many stalls. As explained later, GPU architectures also utilize fine-grained multithreading.

Multithreading can work with superscalar. It is called *simultaneous multithreading (SMT)*. In the SMT, each thread is assigned to a slot.

2.3.2 Multiprocessors

Computers composed of multiple cores are called *multiprocessors*. Usually, the cores in a multiprocessor share the same address space. This kind of multiprocessors are called *Shared memory multiprocessors (SMPs)*.

The SMPs are categorized into two types. If memory access time does not depend on cores or addresses, the architectures are called *uniform memory access (UMA)*. Otherwise, the architectures are called *nonuniform memory access (NUMA)*. Figure 2.5 (a) and (b) show examples of UMA, and Figure 2.6 (a) and (b) show examples of NUMA. Note that caches are not considered as memory.

If multiple cores access the same element in the memory unit, we have to make sure the cores access the element in the correct order. *Thread synchronization* or *synchronization* is a mechanism to make some cores stop until other cores execute a certain instruction. If each core waits at a predetermined point of the program until all other cores reach their predetermined points, it is called *barrier synchronization*.

2.3.3 Flynn's Taxonomy

Flynn's taxonomy [15] is a well-known classification of computer architectures. Computer architectures are classified according to the following two criteria:

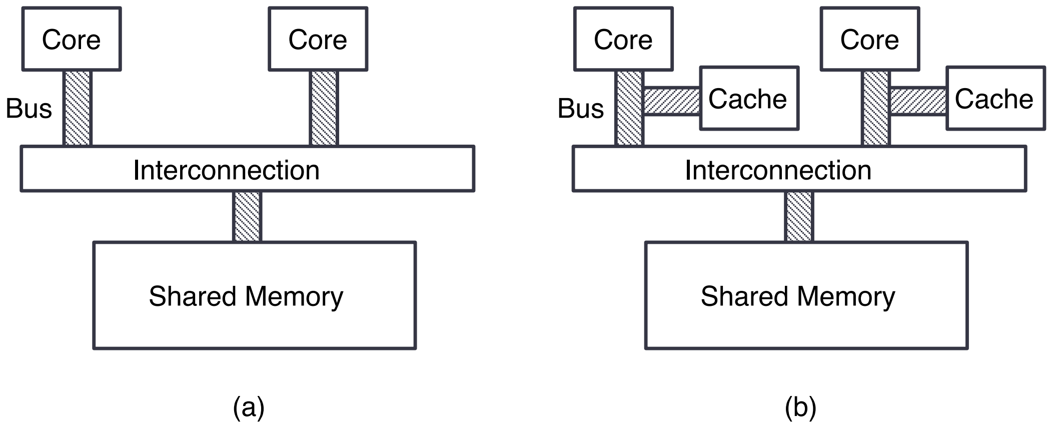


Figure 2.5: Examples of UMA architectures.

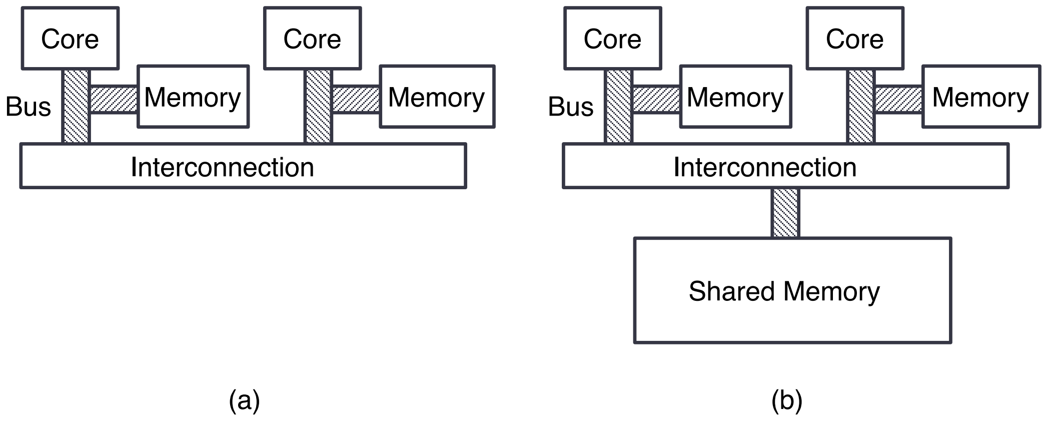


Figure 2.6: Examples of NUMA architectures.

1. Does a program consist of a single thread or multiple threads? The former is called “Single Instruction stream”, and the latter is called “Multiple Instruction stream”.

2. Does a program apply instructions to a single element or multiple elements? The former is called “Single Data stream”, and the latter is called “Multiple Data stream”.

Thus, we have the following four classifications:

- Single Instruction stream, Single Data stream (SISD)
- Single Instruction stream, Multiple Data stream (SIMD)
- Multiple Instructions stream, Single Data stream (MISD)
- Multiple Instructions stream, Multiple Data stream (MIMD)

The advantage of MIMD is that it is easy to increase the number of cores. On the other hand, the disadvantage of MIMD is that it is difficult to develop efficient programs. For example, programs for MIMD require a lot of synchronization in general. Moreover, it is difficult to utilize the burst access of DRAM.

Single program, Multiple Data stream (SPMD) is one implementation of MIMD and all threads execute the same program. It is a common style of parallel programming.

2.3.4 Costs of Parallel Computation

With respect to the cost of computation using multiple cores, the following two facts are well known.

Amdahl [16] argued that the sequential portion of programs has large impact on the whole performance. The argument is formalized as follows.

Theorem 2.1 (Amdahl’s Law) *Suppose we execute a program using P processors and α represents the fraction of the task that is inherently sequential. Let the speedup be the running time in the case where it is executed in serial (with one processor) divided by the running time in the case where it is executed in parallel (with P processors). Then, the speedup S is*

$$S = \frac{1}{\alpha + (1-\alpha)/P}.$$

Proof. Let t_1 be the running time in the case where it is executed with one processor, and t_P be the running time in the case where it is executed with P processors. Then, we have

$$t_P = t_1\alpha + t_1(1-\alpha)/P = t_1\left(\alpha + (1-\alpha)/P\right).$$

Therefore, we have

$$S = \frac{t_1}{t_p} = \frac{1}{\alpha + (1-\alpha)/P}.$$

□

This gives the computational lower bound of parallel computation.

On the other hand, if we know the computation times with both a single core and sufficiently large number of cores, we can obtain the upper bound of the computation time with p cores.

Theorem 2.2 (Brent's theorem [17]) *If a computation C can be performed in time t with q operations and sufficiently many processors which perform arithmetic operations in unit time, then C can be performed in time $t + (q-t)/p$ with p such processors.*

Proof. Suppose that s_i operations are performed at step i, for $i = 1, 2, \cdots, t$. Thus $\sum_{i=1}^{t} s_i = q$. Using p processors, we can simulate step i in time $\lceil s_i/p \rceil$. Hence, the computation C can be performed with p processors in time

$$\sum_{i=1}^{t} \lceil s_i/p \rceil \leq \sum_{i=1}^{t} (p - 1 + s_i)/p = (1 - 1/p)t + (1/p)\sum_{i=1}^{t} s_i = t + (q-t)/p.$$

□

The value q is called a *work*, and the value t is called a *depth* or a *span*.

2.4 GPUs

Graphics Processing Units (GPUs) were originally designed for efficient processing of graphics. In 2006, NVIDIA released GeForce 8800, which is the first GPU that supports C language. This GPU executes multiple threads efficiently due to the SIMT technique explained later. This architecture is called Tesla [18] or G80 [19]. Since then, GPU

Table 2.1: Comparison of each generation of GPUs.

Model	GeForce 8800	C2070	k40	TITAN X[1]
Architecture	G80 [18]	Fermi [19]	Kepler [20]	Maxwell
Cores	128	448	2880	3072
Multiprocessors	16	14	15	24
Cores / Multiprocessors	8	32	192	128
Amount of shared memory (max)	16 KB	48 KB	48 KB	96 KB
Memory bandwidth	104 GB/sec	144 GB/sec	288 GB/sec	336.5 GB/sec

architectures have been improved significantly. Figure 2.1 shows a comparison of each generation of GPUs.

In this dissertation, we explain Fermi architecture [19, 3] proposed by NVIDIA. A lot of GPU architectures are proposed recently, but most architectures including *Graphics Core Next (GCN)*, which is an architecture developed by AMD, have similar characteristics.

2.4.1 Architectures

The Fermi architecture is a hybrid system of CPU and GPU devices. Figure 2.7 shows a brief sketch of GPU architectures. The GPU device comprises multiple cores called *streaming processors (SPs)* organized as multiprocessors called *streaming multiprocessors (SMs)*. Figure 2.8 shows a brief sketch of the multiprocessor. For instance, C2070 model, which is Fermi based, consists of 448 cores organized as 14 multiprocessors. Each multiprocessor individually executes programs and it does not have communication means with other multiprocessors. Multiprocessors have the advantage of MIMD, that is, it is easy to change the number of multiprocessors. The CPU invokes GPU programs

[1]This model is not for High Performance Computing (HPC). Since the specifications are not officially published, it is based on the information from the following sites:

- https://developer.nvidia.com/cuda-gpus
- http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities
- http://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/
- http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications

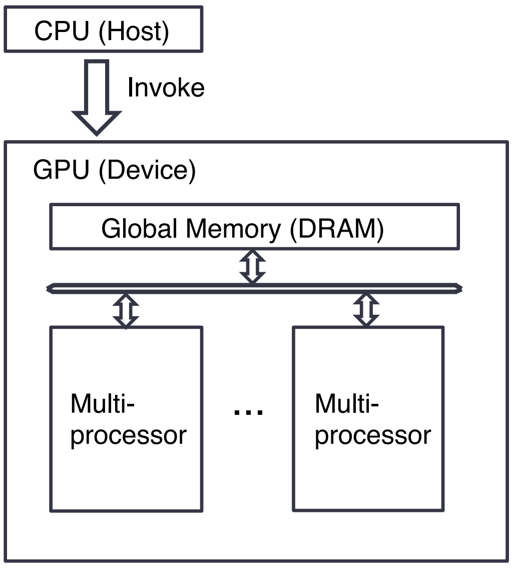


Figure 2.7: Brief sketch of GPU architectures

and only the CPU can synchronize multiprocessors by waiting for all multiprocessors in the device to complete executing programs. All cores in a multiprocessor execute the same instruction at the same time, which is similar to SIMD.

The Fermi architecture has mainly three types of memory. The first one is *global memory*, which can be accessed from all multiprocessors and the CPU. The global memory is composed of DRAM. Therefore accesses to the global memory take longer time than arithmetic operations. The device has a common cache for the global memory, which is shared with all multiprocessors. It is called *L2 cache*. The second one

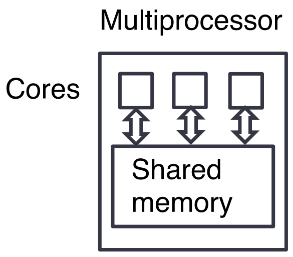


Figure 2.8: Brief sketch of the multiprocessor

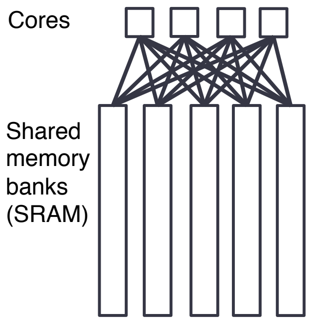


Figure 2.9: An example of shared memory banks

is *shared memory*, which can be accessed only from cores in a multiprocessor. The shared memory is composed of SRAM and each multiprocessor has shared memory inside. It consists of multiple banks, and cores can access distinct banks in parallel. However, the number of cores that can access the same bank in parallel is limited. Figure 2.9 shows an example. In Fermi architecture, a bank cannot be accessed from multiple cores simultaneously unless cores access the same address. The third one is *registers* in cores. The Fermi architecture has a large set of registers that can be used for storage space for internal process. The device has several types of other memory, but we do not explain them.

2.4.2 Multiprocessors

In Fermi architecture, each multiprocessor can handle up to 1536 threads concurrently. The threads can be synchronized using barrier synchronization. The threads are divided into groups with 32 threads called *warps*. The multiprocessor always applies the same instruction to all threads in a warp. In other words, the multiprocessor issues SIMD instructions to the threads in a warp. Moreover, the multiprocessor can switch between warps similar to fine-grained multithreading. In other words, the multiprocessor always selects an active warp. This method is called *Single Instruction stream, Multiple Threads (SIMT)*, and it has the advantages of both SIMD and fine-grained multithreading. Figure 2.10 shows the process.

Figure 2.11 shows an example of multithreading. When a warp waits for data from the global memory, the multiprocessor executes another warp. Thus, the multiprocessor

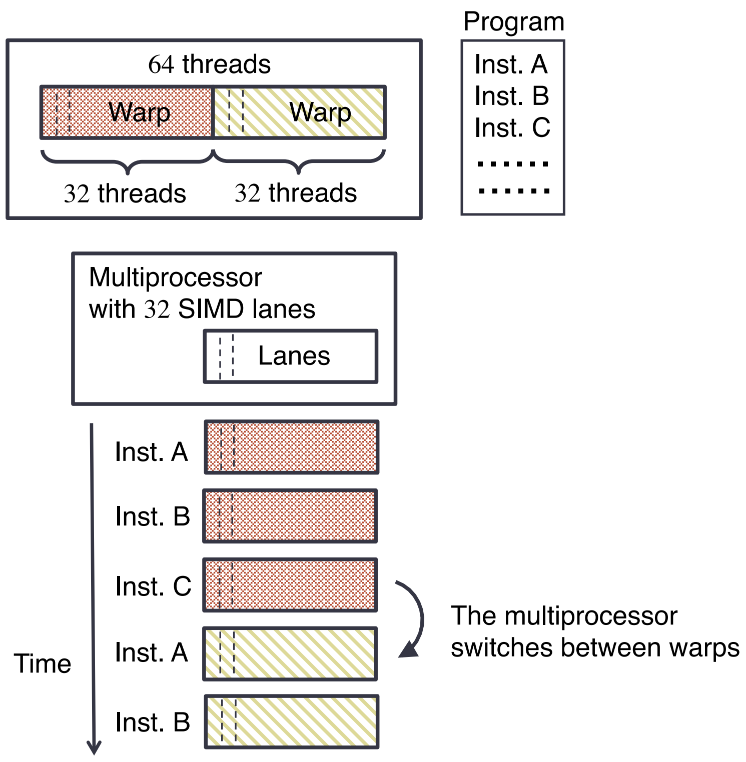


Figure 2.10: SIMT

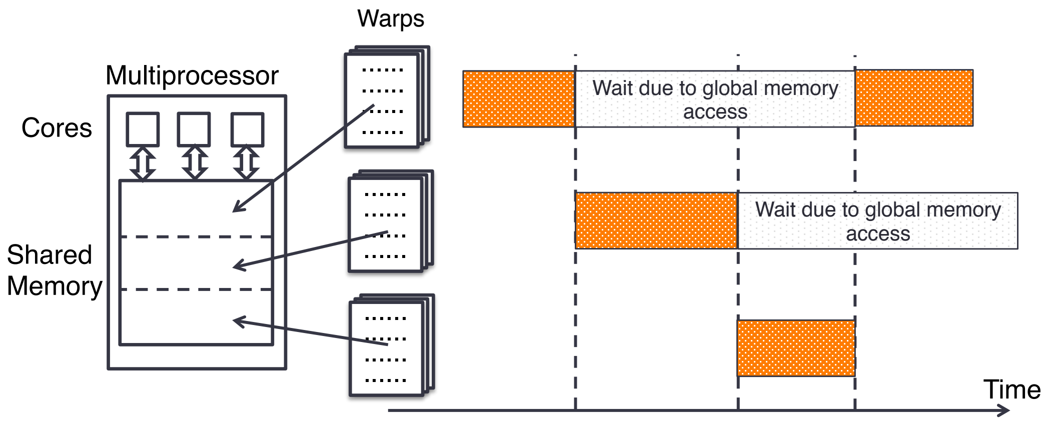

Warps
Multiprocessor
Cores
Shared Memory
Wait due to global memory access
Wait due to global memory access
Time

Figure 2.11: An example of multithreading

keeps active. Moreover, the global memory access is efficiently done if the threads in the warp access the contiguous elements in the global memory. Note that the maximum number of warps executed concurrently is constrained by the amount of shared memory and registers used by a warp, because all warps in a multiprocessor share the same memory.

In order to obtain high performance, it is important to design programs with consideration for warps. For example, when threads in a warp diverge via a data-dependent conditional branch, the multiprocessor serially executes each branch path, which is inefficient. It is also important to consider memory access patterns including bank conflicts and coalescing, which will be discussed in Section 3.2.1.

2.4.3 CUDA

NVIDIA provides a programming environment called *CUDA*. CUDA provides programming language, development tools, and the software architecture. Programmers write codes based on the software architecture, and GPU hardware emulates the software architecture. Thanks to the architecture, programs implemented using CUDA can be executed on all NVIDIA GPUs. Figure 2.12 illustrates it. We first explain the software architecture, and then, explain how GPU hardware emulates the software architecture.

A CUDA program consists of a CPU program and GPU programs. GPU programs are called *kernels*, and the CPU invokes the GPU to execute a kernel. A kernel has

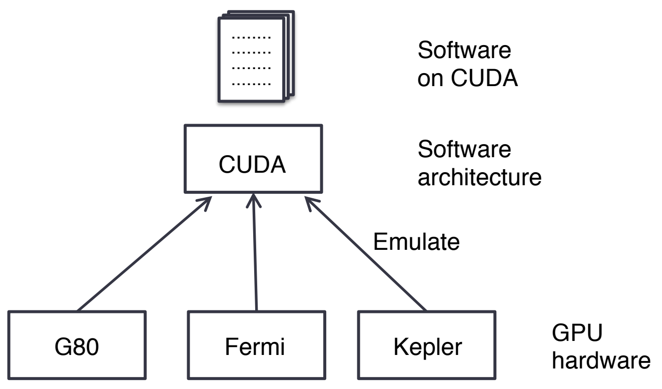
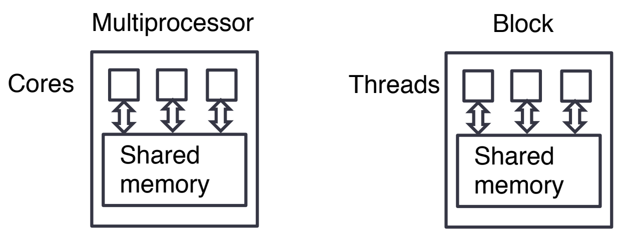


Figure 2.12: CUDA overview

a hierarchy of sets of threads: grids, blocks, and threads. One grid is assigned for a kernel, that is, the grid contains all threads in a kernel. A grid consists of blocks, and a block consists of threads. The hierarchy is mapped to the hierarchy of the hardware. The correspondence between the hardware and the kernel is shown in Figure 2.13. A block is a set of threads that are executed by a single multiprocessor. A multiprocessor may be assigned more than one blocks. Programmers are unconscious of the assignment of blocks to multiprocessors. One thread is executed by a single core, and each core concurrently executes multiple threads.

Figure 2.14 shows how to emulate the grid using GPU hardware. Suppose the number of blocks in a grid is eight. If the number of multiprocessors in a GPU is two, then, each block is basically assigned four blocks. If the number of multiprocessors in a GPU is four, then, each block is basically assigned two blocks. Note that several blocks in a multiprocessor can be executed concurrently, which is different from the figure.

Next, we explain how to emulate a block using a multiprocessor. As mentioned in Section 2.4.2, multiprocessors support SIMT. If the number of cores in a multiprocessor is larger than 32, then, cores are divided into groups with 32 cores, and each group issues SIMD instructions to warps. If the number of cores is smaller than 32, the cores execute 32 SIMD lanes in a warp over multiple clock cycles. In Fermi architecture, 32 cores in a multiprocessor are divided into two groups with 16 cores, and each group executes SIMD instructions basically over two clocks.



GPU Hardware	Kernel (software)
Core	Thread
Multiprocessor	Block (set of threads)
GPU device	Grid (set of blocks)

Figure 2.13: CUDA's hierarchy of threads

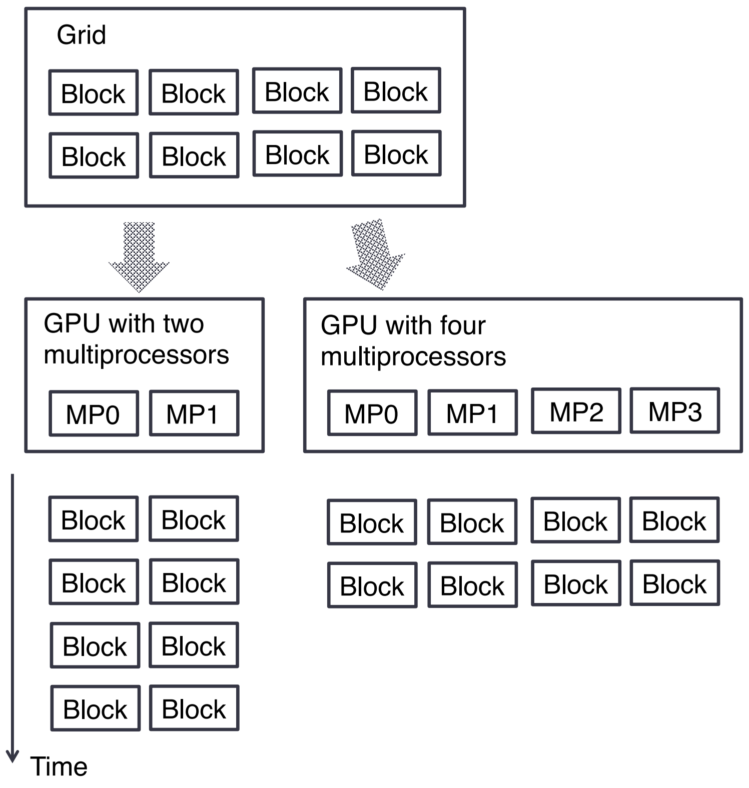


Figure 2.14: How to emulate the grid using a GPU

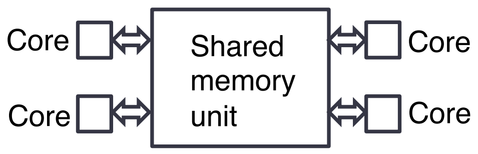


Figure 2.15: PRAM model

2.5 Parallel Computational Models

2.5.1 PRAM Model

Parallel random access machine (PRAM) model [4, 21, 22] has p cores that can execute arbitrary instructions with a constant number of operands simultaneously. Figure 2.15 shows an example. The p cores have a shared memory unit of M words. Each core can read/write from/to an arbitrary address in parallel.

PRAM models are classified according to whether multiple cores can simultaneously access the same address of the memory:

- Exclusive-Read, Exclusive-Write (EREW): multiple cores cannot read or write the same address,
- Concurrent-Read, Exclusive-Write (CREW): multiple cores can read the same address, but cannot read the same address,
- Exclusive-Read, Concurrent-Write (ERCW): multiple cores cannot read the same address, but can write the same address (this case is not used),
- Concurrent-Read, Concurrent-Write (CRCW): multiple core can read and write the same address.

When multiple cores attempt to write values to the same address at the same time in "Concurrent-Write", the model writes a value according to a predetermined rule. For example, the model writes a value only when all cores attempt to write the same value.

In this dissertation, we consider EREW (exclusive-read, exclusive-write) PRAM model unless otherwise stated. We denote the model by PRAM(p, M).

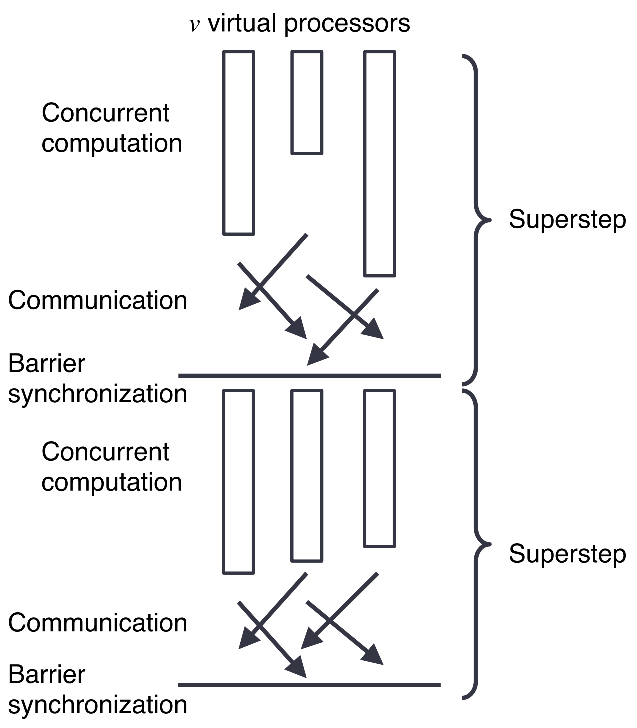


Figure 2.16: BSP model

2.5.2 BSP Model

Bulk Synchronous Parallel (BSP) model [23] is one of the parallel programming models to make it possible to write programs without conscious of physical processors. The number v of virtual processors in the BSP model is larger than the number p of physical processors, and users need not take processor assignment into account. This helps users write general parallel programs. Figure 2.16 shows an example. In the BSP model, a computation proceeds in a series of supersteps, each of which consists of concurrent computation, communication, and barrier synchronization steps. The cost of a superstep is determined by the maximum computation time by the p processors, the maximum size of sent and received messages among the processors, and the time for synchronization.

2.6 GPU Models

GPU models are basically categorized into performance prediction models and computational models. Performance prediction models aim to estimate running time of programs without executing the programs on GPUs, and many models are proposed [7, 8, 24].

Computational models aim to provide asymptotic computational complexities of algorithms. Nakano [25, 5] proposed memory machine models designed for GPUs. Sitchinava and Weichert [6] also proposed a computational model for GPUs. These models assume that a GPU is a SIMD machine. Therefore, we cannot take account of multithreading, which is one of the most important features of GPUs. Moreover, Ma et al. [26] proposed a memory access model for highly-threaded many-core architectures. This model can analyze the asymptotic behavior of the global memory access using a work and a span explained in Section 2.3.4.

Some researchers take a different approach. Hou et al. [27] proposed a programming model BSGP for GPUs. This model is based on the BSP model [23]. Programmers can easily write BSGP-based codes and the BSGP compiler automatically translates the codes to GPU-based codes.

In the following sections, we explain some GPU-based computational models.

2.6.1 DMM and UMM

Discrete Memory Machine (DMM) and Unified Memory Machine (UMM) [25] are memory machine models designed for GPUs. The DMM captures the features of the shared memory of GPUs, while the UMM captures the features of the global memory.

The DMM consists of w memory banks, and word address i is located in the (i mod w)-th bank. Figure 2.17 (a) shows an example. Each bank can be accessed from one core at a time. We assume that it takes ℓ unit times to obtain the first word since a core issues a memory access instruction, where ℓ is the latency of the memory access. Let p be the number of cores in the machine. When core i ($0 \leq i < p$) accesses address i, it takes $\ell - 1 + \lceil p/w \rceil$ unit times. We can analyze the effect of bank conflicts. Figure 2.18 (a) shows an example. When four cores access addresses 0, 1, 5, and 10, addresses 1 and 5 cannot be accessed at a time due to the bank conflict. Therefore, it

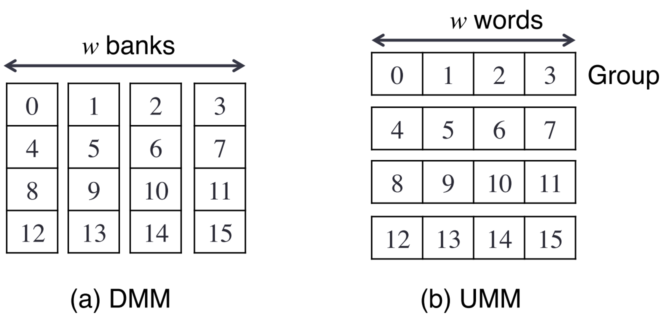


Figure 2.17: DMM and UMM

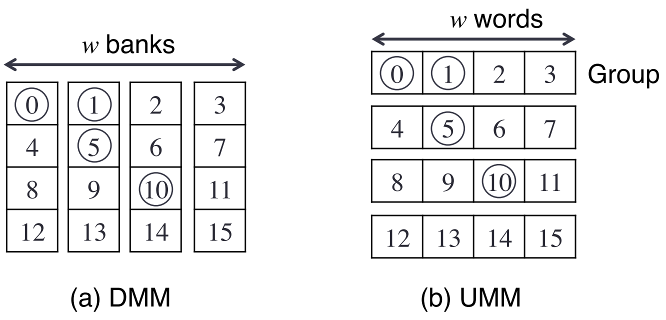


Figure 2.18: Memory access on the DMM and the UMM. When four cores access addresses 0, 1, 5, and 10, the DMM takes $\ell + 1$ unit times and the UMM takes $\ell + 2$ unit times, where ℓ is the latency of the memory access.

takes $\ell - 1 + 2 = \ell + 1$ unit times.

The UMM consists of multiple address groups, each of which consists of w words and word address i is located in the $\lfloor i/w \rfloor$-th group. Figure 2.17 (b) shows an example. All words in a group can be accessed simultaneously. When core i ($0 \le i < p$) accesses address i, it takes $\ell - 1 + \lceil p/w \rceil$ unit times as is the case with the DMM. We can analyze the effect of coalescing. Figure 2.18 (b) shows an example. When four cores access addresses 0, 1, 5, and 10, the machine has to access three groups. Therefore, it takes $\ell - 1 + 3 = \ell + 2$ unit times.

In our GPU model explained in Chapter 3, the latency for the shared memory access is not considered because it will be small enough. Moreover, instead of considering the latency for the global memory access, our model takes the effect of multithreading into account.

2.6.2 PEM Model

Parallel external memory (PEM) model [6] is a computational model proposed by Sitchinava and Weichert. This model assumes the multiprocessors are SIMD machines. A program is divided into some rounds by synchronizations, and the running time $T(n)$ of a round is upper bounded as follows:

$$T(n) = t(n) + \lambda q(n) + \sigma,$$

where $t(n)$ is the time complexity of the round, $q(n)$ is the I/O complexity of the round, λ is the latency for the global memory access, and σ is the upper bound on time for a barrier synchronization.

Again, in our GPU model explained in Chapter 3, instead of considering the latency for the global memory access, we takes the effect of multithreading into account.

2.7 I/O Model

In GPU programming, the number of I/Os significantly affects the performance. In order to evaluate the I/Os, the following model is very useful. In Section 3.3.3, we reveal the relation to our GPU model.

The standard *I/O model* or *external memory model* [12] consists of a single processor, an internal memory that can hold M words, and an external memory (a disk). The external memory is divided into blocks with b words, and the processor can read/write a single block per unit time. Algorithms are evaluated by only the sum of read and write instructions. We call the number I/O complexity and denote this model by I/O(b, M). Figure 2.19 shows the model.

2.8 Short Summary

In this chapter, we provided some preliminaries, which are necessary for later discussion. As explained in Section 2.4, GPUs utilize MIMD to increase the scalability, and utilize

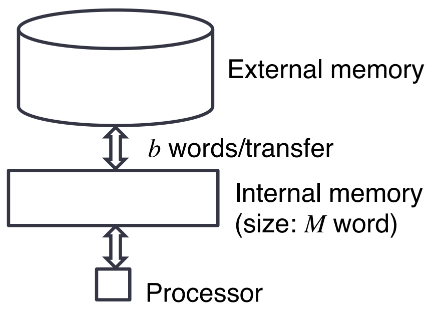


Figure 2.19: I/O model

SIMT to improve the performance of multiprocessors. SIMT is implemented by the combination of SIMD and fine-grained multithreading. Instead of the good efficiency of SIMT, it is difficult to extract the optimal performance out of GPUs. Therefore, appropriate computation models are required.

Finally, we introduced some existing models. We can utilize the models by revealing the relation to our GPU model, which is discussed in Section 3.3.

3 Parallel Computation Models for GPUs

3.1 Introduction

We introduce a new computational model in order to analyze asymptotic computational complexities of GPU-based algorithms. Complexity analyses are useful to design efficient algorithms. For example, we can compare the efficiency of several algorithms without executing the programs. Based on comparisons with the lower bound, we can find what we should refine.

Actually, actual running time is often used to evaluate GPU-based algorithms. However, in this case, we have to implement the algorithms and do code optimization. If one does not know how to optimize the codes, one cannot show the efficiency of the algorithms. Since new architectures are proposed one after another, one needs to spend much time to follow them.

We would like to separate designing algorithms from optimizing the codes. We design our GPU model AGPU for this purpose. The model is simple and able to take account of a lot of factors affecting performance such as coalescing, bank conflicts, multithreading. Complexities on GPUs depend on device specifications, but they are within a constant factor of complexities on the AGPU model.

In this chapter, we explain the architecture and the metrics of our model. After that, we explain the relation between our model and other computational models. It is useful to analyze the complexities of algorithms and find the lower bound.

3.2 AGPU Model

3.2.1 Architecture

We propose a new computational model *Abstract GPU (AGPU)*. It is an abstracted computational model that captures the essence of common GPU architectures. We focus on features affecting performance and make the model as simple as possible. Figure 3.1 shows the architecture of the AGPU model. AGPU consists of a *host* (CPU) and a *device* (GPU). The device consists of p cores and one *global memory unit*. In this model, each core handles a single thread and executes one instruction per unit time. The word length of the device is w bits. A group of b cores forms a *multiprocessor*. The device has k multiprocessors, that is, $p = kb$. Each multiprocessor has its own *shared memory unit* with M words and individually executes programs invoked by the host. We assume $M \geq b^2$. We also assume b can be represented as powers of two in order to omit the process of rounding of fractions.

The multiprocessors have no means of communicating with each other. The host can synchronize the multiprocessors by waiting for all multiprocessors in the device to complete executing programs.

Each core can apply basic operations to words stored in the shared memory. The basic operations include addition, subtraction, multiplication, division, and shift. Each operation can be executed in one unit time unless bank conflicts occur. Cores cannot apply operations to words in the global memory. If we need them, we first read them from the global memory to the shared memory, then apply the operations to them. Cores on real GPUs have large amount of registers, while cores on the AGPU model do not have registers in order to make the model as simple as possible. Though memory access to registers is several times faster than memory access to the shared memory in general, we consider they are within a constant factor.

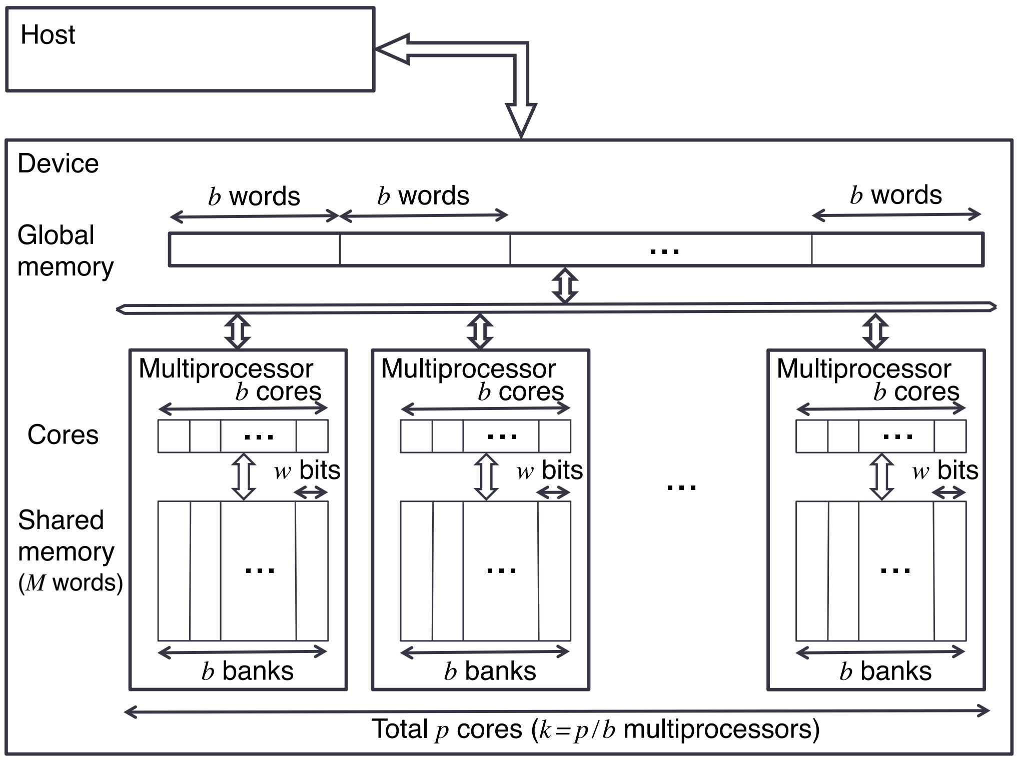


Figure 3.1: The architecture of AGPU model

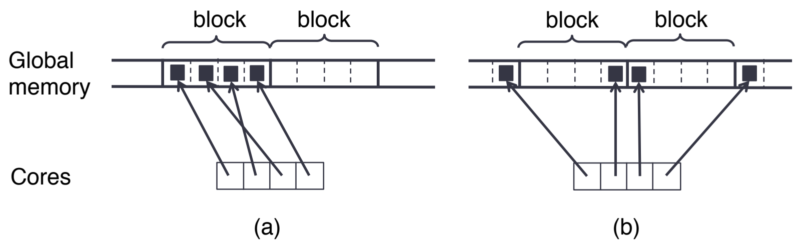


Figure 3.2: Examples of global memory accesses. Each block stores four words. (a) All instructions coalesce. (b) These instructions do not coalesce because the words are spread out among four blocks.

Each multiprocessor always makes all cores in it fetch the same operation, but data addresses are arbitrary, namely, each core in a multiprocessor applies the same operation to distinct words. In other words, all cores in a multiprocessor must take the same execution path. When cores diverge via a data-dependent conditional branch, the multiprocessor serially executes each branch path.

The global memory unit is high-capacity, low-speed and can be accessed by the host and all multiprocessors in the device, whereas the shared memory units are low-capacity, high-speed and can be accessed by only cores in the multiprocessor. The global memory unit is divided into blocks with b words. The AGPU model has only two instructions to access the global memory unit; one is a read instruction that copies all words in a block to a shared memory unit, and the other is a write instruction that copies b words in a shared memory unit to a block. Real GPU devices have the same mechanism as these instructions, which are called *coalescing* or *coalesced access.* Figure 3.2 shows examples of global memory accesses. In Figure 3.2(a), all memory access instructions coalesce into one instruction and it is executed in a unit time. On the other hand, in Figure 3.2(b), the instructions are executed in four unit times because the words are spread out among four blocks. Note that if the b cores in a multiprocessor access b consecutive words in the global memory, the instructions are executed in at most two unit times. This is called *contiguous access to the global memory.*

The shared memory unit in each multiprocessor is divided into b banks. All b cores in a multiprocessor can access b distinct banks simultaneously. If multiple cores are accessing the same bank, the accesses are serialized, which is called *bank conflict.*

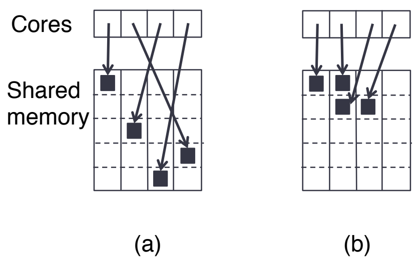


Figure 3.3: Examples of shared memory accesses. Each column of the shared memory unit represents a memory bank, and each cell can store one word. (a) Bank conflicts do not occur since no banks are accessed by multiple cores. (b) The instruction is divided into two instructions since the second column is accessed by two cores (bank conflict).

Figure 3.3 shows examples of shared memory accesses. The columns of the shared memory unit represent the banks. The addresses are allocated in the order of increasing row index, that is, the first b addresses are allocated in the first row. In Figure 3.3(a), all cores access distinct banks, therefore bank conflicts do not occur. On the other hand, in Figure 3.3(b), the second column is accessed by two cores, therefore the instruction is divided into two instructions. Note that if the b cores in a multiprocessor access b consecutive words in the shared memory, bank conflicts do not occur. This is called *contiguous access to the shared memory.*

We denote this model by AGPU(p, b, M, w). We may omit the parameters w and M if they do not affect the performance of algorithms. In this case we denote the model by AGPU(p, b, M) or AGPU(p, b).

Finally, we consider a variation of the model: volatile AGPU and non-volatile AGPU. In the volatile AGPU, all data in the shared memory units are erased when the host synchronizes multiprocessors, while in the non-volatile AGPU, all data in the shared memory units are kept after synchronization. In the volatile model, if some variables in a shared memory unit are necessary for the following process, the multiprocessor has to write the variables to the global memory unit at the end of the process. The CUDA environment uses the volatile model. We denote the volatile model as AGPU, and the non-volatile model as AGPU$'$.

3.2.2 Metrics

To evaluate the performance of algorithms, we use five metrics: the *time complexity*, the *I/O complexity*, the *multiplicity*, the *amount of the global memory used* and the *amount of the shared memory used*. The multiplicity will be explained in the next section. The time and I/O complexities are used to evaluate the running time of algorithms. The time complexity is the number of instructions each multiprocessor executes. When cores in a multiprocessor diverge, we count the instructions in all branch paths. If the time complexity varies by multiprocessors, the largest complexity is adopted. The I/O complexity is the total number of the global memory access instructions issued by all multiprocessors. The reason why we analyze the I/O complexity separately from the time complexity is that the execution time of the instructions to access the global memory is quite larger than the time for other instructions, and this may be a bottleneck. Since the number of multiprocessors accessing the global memory simultaneously is limited by the bandwidth of the global memory, the I/O complexity is defined as the summation of the number of global memory access instructions issued by each multiprocessor.

The amounts of the global and shared memory used are used to evaluate the memory usage of algorithms. If the amount of the shared memory used varies according to the multiprocessors, the largest amount is adopted. If the amount of the shared memory used is larger than M words, the algorithm cannot be implemented on GPU. As we discuss in the next section, a large amount of the shared memory used makes multithreading less effective. Moreover, it is important to reduce the amount of the global memory used, especially if the input size is large.

3.2.3 The Effect of Multithreading

As mentioned in Section 2.4.2, real GPU devices have a mechanism called multithreading. Although it has a huge impact for the efficiency of global memory accesses, the I/O complexity is useless to estimate the effect of multithreading because multithreading does not change the value of the I/O complexity. Therefore, we need a new metric. In this section, we define *multiplicity* to evaluate the effect. Since we assume that each core executes a single thread in the AGPU model, we cannot directly evaluate the number of threads each real GPU core executes concurrently. However, we provide a simple method to evaluate the efficiency of multithreading of programs on the AGPU model.

Each core in the real GPU devices handles multiple threads concurrently. The way to make multithreading more effective is to increase the number of threads per core. The number is limited by device specifications. When the maximum number of threads is assigned, multithreading is most effective.

Supposing m is the amount of the shared memory used by a multiprocessor on the AGPU model, the *multiplicity* $\mathcal{M}$ is defined as $\mathcal{M} := M/m$. As mentioned in Section 2.4.2, the number of threads assigned to a core is limited by the amount of memory used. When m is small, the multiplicity becomes large. In CUDA terms, *occupancy* is defined as the number of assigned threads divided by the maximum number of threads. The multiplicity corresponds to the occupancy, but it is simplified and can be calculated only using other AGPU metrics.

Finally, we discuss the value of the multiplicity. In real GPUs, the number of warps each multiprocessor concurrently executes is limited by device specifications. In NVIDIA GPUs, the maximum value is strictly larger than M/b^2, but at most M/b. Namely, when $m = O(b)$, we can assign the maximum number of warps to the multiprocessors. Therefore, when $m = O(b)$, we say the multiplicity is optimal. On the other hand, when $m = \Omega(b^2)$, we consider the multiplicity is not optimal.

3.2.4 Discrepancy Between the AGPU Model and Real Architectures

Since our model is designed for analyzing time and I/O complexities, it is rather simplified. We discuss the discrepancy between our model and real GPU architectures.

Firstly, we do not take memory caches into account. Though many GPU devices have caches, their specifications differ a lot and it is difficult to analyze cache behavior. The aim of using the AGPU model is not to predict the actual running time of programs but to analyze the asymptotic behavior of algorithms when the input size grows. Therefore, we do not consider caches in the AGPU model. This makes it easy to analyze the I/O complexities of algorithms. In the RAM model, it is common that memory caches are not considered to analyze the asymptotic complexities of sequential algorithms. Nevertheless, the obtained complexities are very useful for designing algorithms. We therefore consider that complexity analyses using the AGPU model are also useful for designing GPU-based algorithms. With respect to the L2 caches, we can consider the AGPU has user-managed caches instead of the L2 caches. If we can analyze the behavior of the L2 caches, we can change the algorithm so that it has user-managed caches on the shared memory. Therefore the AGPU model does not need to have the L2 caches.

Secondly, the memory organization of the multiprocessor in the AGPU model is different from that in real GPUs. In real GPUs, each core has a large amount of registers and we can store data in the registers, while in the AGPU model, we cannot store data in the registers. The AGPU model uses the shared memory instead. Since we assume that each core can access the shared memory in unit time, this does not change the time complexity. The amount of the shared memory in AGPU corresponds to the sum of the amount of the shared memory and the registers in real GPUs.

Thirdly, real GPU architectures have many parameters such as the number of cores in a multiprocessor, the number of banks on the shared memory unit, the block size of the global memory, while these are fixed to b in the AGPU model. However, it dose not affect asymptotic behavior of algorithms because we can consider that the differences of the parameters are within a constant factor of b. The time and I/O complexities in the AGPU model are therefore within a constant factor of those in real GPU devices.

Finally, the AGPU model does not consider synchronization of threads in a multiprocessor. In real GPU devices, cores execute multiple threads concurrently to improve the efficiency of global memory accesses, which is called multithreading. CUDA supports synchronization in a multiprocessor, while the AGPU model does not support it to simplify the model. We consider that this is not a severe restriction. The reason that CUDA has the mechanism of synchronization in a multiprocessor is that the number of cores is normally smaller than that of executed threads. In the AGPU model, we analyze the complexities of algorithms by assuming that the number of cores is equal to that of threads. However, using Theorem 3.9 in Section 3.3.4, we can easily obtain the complexities when algorithms are executed on multiprocessors with fewer cores. We can also estimate the effect of multithreading using the AGPU model as discussed in Section 3.2.3. Therefore it is not necessary to use synchronization in a multiprocessor.

3.2.5 Notation for Pseudo-codes

We explain notation for pseudo-codes on AGPU(p, b, M). Let MP[$0..k-1$] be an array of multiprocessors, where $k = p/b$. Let Core[$0..b-1$] be an array of cores in a multiprocessor. When multiple multiprocessors execute a program in parallel, we write as follows:

```
1: **for all** $\rho \in$ MP[$x..y$] **in parallel do**
2:     carry out some processing
3: **end for**
```

where $x..y$ represents the range of multiprocessors that execute a program. "for all" loops are launched by a host. Namely, codes outside of "for all" loops are executed by the host. All multiprocessors are synchronized at the end of "for all" loops by the host. Although a real multiprocessor may concurrently execute multiple blocks (corresponding to multiprocessors on AGPU(p, b, M)), programmers do not need to care the assignment.

When all cores in a multiprocessor execute a program in parallel, we write as follows:

```
1: **for all** $\epsilon \in$ Core[$0..b-1$] **in parallel do**
2:     carry out some processing
3: **end for**
```

We cannot specify the range of cores because all cores in the multiprocessor must execute the same instruction.

Next, we explain the instructions to access the global memory; the symbols "$\Rightarrow$" and "$\Leftarrow$" represent the global memory access instructions. We can access at most b consecutive words in the global memory per instruction. Since a multiprocessor in the AGPU model can access one block of the global memory in a unit time, the multiprocessor may access the global memory twice to obtain b consecutive words. However, it does not change the asymptotic I/O complexity.

The symbols "$\rightarrow$" and "$\leftarrow$" represent the shared memory access instructions. The symbol ":=" represents an assignment of a pointer. Variable names begin with a capital letter if they are in the global memory. Otherwise, they begin with a lower-case letter.

3.3 Relations between the AGPU Model and Other Computational Models

In this section we discuss relations between the AGPU model and other computational models in order to evaluate the power and limitation of the models. First we give a notation.

Definition 3.1 *Let X, Y be computational models. If for any algorithm A_Y on Y, there exists an algorithm A_X for the same problem on X such that the time (I/O) complexity of A_X is equal to or less than the value that is α times the time (I/O) complexity of A_Y, we denote this by $X \leq \alpha Y$ ($X_{IO} \leq \alpha Y_{IO}$). If it holds that $X \leq O(1)Y$ and $Y \leq O(1)X$, we denote this $X = Y$. We define $X_{IO} = Y_{IO}$ analogously.*

3.3.1 PRAM Model

The difference between the PRAM model and the AGPU model is the following. First, in the PRAM model, p processors can execute different instructions at the same time (MIMD), while in the AGPU model, processors in a multiprocessor execute an identical instruction (SIMD). Secondly, the PRAM model does not have memory hierarchy; the PRAM model has only a shared memory. Thirdly, memory access of the AGPU model is more restrictive than that of the PRAM model. Unless all cores inside a multiprocessor access consecutive elements in the global memory, the accesses do not coalesce. Additionally, when the cores inside the multiprocessor access the shared memory, bank conflicts occur unless the cores access distinct banks.

For any algorithm on AGPU(p, b, M) using g-word global memory, there is a corresponding PRAM$(p, g + pM/b)$ algorithm running in the same time complexity, that is, PRAM$(p, g + pM/b) \leq O(1)$AGPU(p, b, M). This means that a lower bound on the time complexity in EREW PRAM model also holds in AGPU. This is useful for algorithm analyses.

On the other hand, for any algorithm on the PRAM model, the following theorem holds:

Theorem 3.2 *Consider any algorithm on the EREW PRAM(p, M) model that has an instruction set of a constant number of instructions. Then it holds that AGPU$(p, p, M) \leq O(M/p)$PRAM(p, M).*

Proof. We simulate the EREW PRAM model by AGPU(p, p, M), that is, all the p cores belong to a single multiprocessor, and they use the same shared memory. An algorithm on PRAM model, in which cores can execute different instructions at the same time, can be converted to that on AGPU(p, p, M) by sequentially executing all types of instruction in each cycle. The time complexity increases, but is multiplied by only a constant factor. We also have to solve the bank conflict problem. Since the PRAM algorithm uses M contiguous words of the shared memory, at most $\lceil M/p \rceil$ words belong to the same bank. Therefore, the degree of bank conflict is at most $\lceil M/p \rceil$ in each memory access. Then the running time of the AGPU algorithm is bounded by $\lceil M/p \rceil$. □

If M is linear to p, then it holds that AGPU(p, p, M) $\leq O(1)$PRAM(p, M). The theorem indicates that we can use known PRAM algorithms to design efficient algorithms executed inside a multiprocessor.

3.3.2 Bulk Synchronous Parallel Model

We can implement a BSP algorithm using AGPU′(p, 1), that is, each multiprocessor has only one core. Each multiprocessor in the AGPU model corresponds to a processor of the BSP model, which is shown in Figure 3.4.

Communication between processors in the BSP model is done by using the global memory of the non-volatile AGPU model. However the I/O complexity on the AGPU model is higher than the communication cost on the BSP model because the I/O complexity on the AGPU model is defined as the total number of global memory access instructions.

3.3.3 I/O Model

Lemma 3.3 *For the volatile and the non-volatile models,*

$$I/O_{IO}(b, M) = AGPU_{IO}(b, b, M) = AGPU'_{IO}(b, b, M).$$

Proof. A multiprocessor in AGPU(b, b, M) corresponds to a processor in I/O(b, M) and shared memory in AGPU(b, b, M) corresponds to internal memory in I/O(b, M).

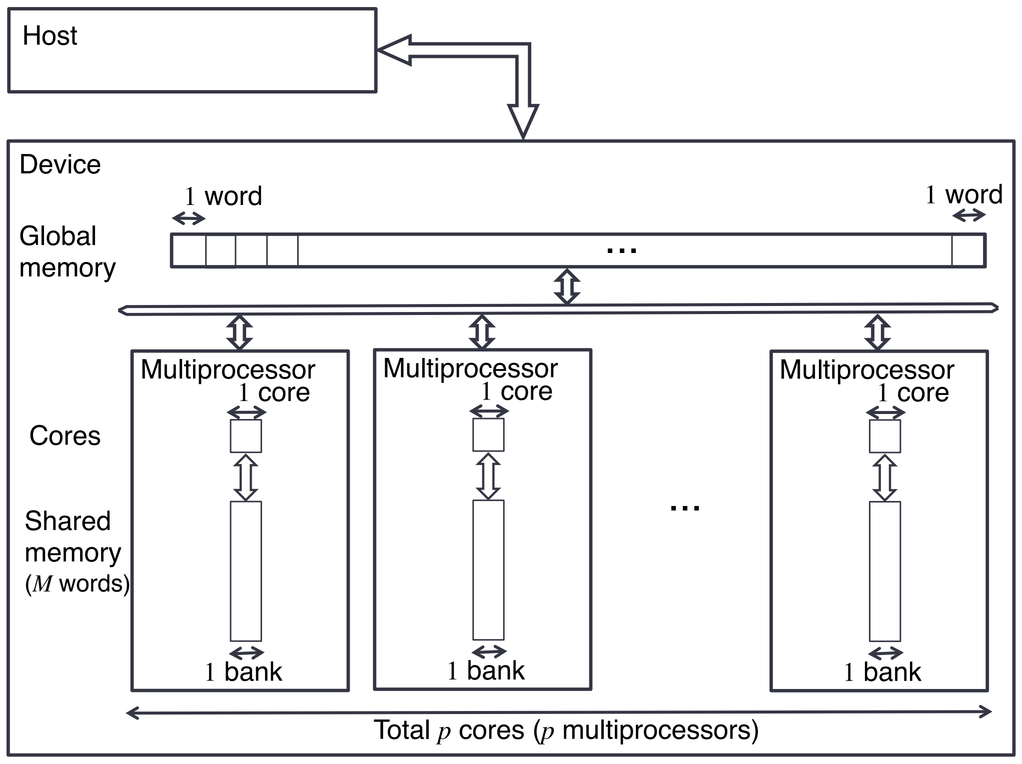


Figure 3.4: BSP model simulated by the AGPU model

The both memory can keep M words. The global memory access instructions in AGPU(b, b, M) correspond to block transfers in I/O(b, M). Therefore, I/O$_{IO}(b, M)$ = AGPU$_{IO}(b, b, M)$. Because AGPU(b, b, M) has only one multiprocessor, it is not necessary to synchronize. Therefore the claim holds for both volatile and non-volatile models. □

Lemma 3.4 *For the volatile AGPU model,*

$$AGPU_{IO}(p, b, M) = AGPU_{IO}(b, b, M).$$

Proof. AGPU$_{IO}(b, b, M)$ comprises only one multiprocessor equipped with a shared memory unit of M words while AGPU$_{IO}(p, b, M)$ comprises p/b multiprocessors, each of which has a shared memory unit of M words. It is trivial that for any algorithm on AGPU(b, b, M) there exists an algorithm on AGPU(p, b, M) that has same I/O complexity as the algorithms on AGPU(b, b, M). Then we consider simulating any AGPU(p, b, M) algorithm on AGPU(b, b, M). As mentioned in Section 3.2.1, the host can synchronize multiprocessors. Let a phase be duration from a synchronization to the next synchronization. If there is no synchronization in a program, the program has a single phase. Suppose one multiprocessor in AGPU(b, b, M) sequentially executes tasks that p/b multiprocessors in AGPU(p, b, M) are supposed to execute in parallel in a phase. Since multiprocessors have no means of communication with each others and all data in shared memory units are deleted at the time of synchronization, the multiprocessor in AGPU(b, b, M) can always refer the same data as a multiprocessor in AGPU(p, b, M). Therefore, it can execute any instructions the multiprocessor in AGPU(p, b, M) executes. Since I/O complexity is defined as the total number of global memory access instructions issued by all multiprocessors, the tasks the multiprocessor in AGPU(b, b, M) executes has the same I/O complexity as the tasks all multiprocessors in AGPU(p, b, M) execute. It holds for any phase. Therefore, for any algorithm on AGPU(p, b, M) there exists an algorithm on AGPU(b, b, M) that has same I/O complexity as the algorithms on AGPU(p, b, M). □

From Lemmas 3.3 and 3.4, it is obvious that:

Theorem 3.5 *For the volatile model,*

$$I/O_{IO}(b, M) = AGPU_{IO}(p, b, M).$$

Next we consider the case of the non-volatile AGPU model. Lemma 3.4 does not hold in this case.

Lemma 3.6 *For the non-volatile AGPU model, it holds*

$$AGPU'_{IO}(b, b, \frac{pM}{b}) \leq AGPU'_{IO}(p, b, M).$$

Proof. We consider one phase as is the case with Lemma 3.4. Suppose one multiprocessor in AGPU$(b, b, (p/b)M)$ sequentially executes tasks that $k = p/b$ multiprocessors in AGPU(p, b, M) are supposed to execute in parallel in a phase. Since data in the shared memory can be used in the next phase, the multiprocessor has to keep all data in the shared memory for the following phase. A multiprocessor in AGPU$'(b, b, (p/b)M)$ can keep all data that k multiprocessors in AGPU(p, b, M) store in the shared memory. As with Lemma 3.4, the multiprocessor in AGPU$(b, b, (p/b)M)$ can execute any instructions the multiprocessor in AGPU(p, b, M) executes. □

Note that it is not always true that for any algorithm on AGPU$'(b, b, (p/b)M)$ there exists an algorithm on AGPU$'(p, b, M)$ that has the same I/O complexity as the algorithms on AGPU$(b, b, (p/b)M)$.

From Lemmas 3.3 and 3.6, we have:

Theorem 3.7 *For the non-volatile AGPU model, it holds*

$$I/O_{IO}(b, \frac{pM}{b}) \leq AGPU'_{IO}(p, b, M).$$

We can also relate the volatile and non-volatile AGPU models. It is obvious that AGPU$'_{IO}(p, b, M) \leq$ AGPU$_{IO}(p, b, M)$, and we also obtain:

Theorem 3.8 *For any algorithm on the non-volatile AGPU using s synchronizations,*

$$AGPU_{IO}(p, b, M) \leq AGPU'_{IO}(p, b, M) + O\left(\frac{spM}{b}\right).$$

Proof. We can simulate any non-volatile AGPU algorithm on the volatile AGPU as follows. At each synchronization, we save all the contents of shared memory to the global memory, and before executing a program in a multiprocessor, the contents of its shared memory are restored. Therefore extra $O(spM/b)$ I/Os are enough for the simulation. □

3.3.4 Multithreading in AGPU

Finally, we discuss the complexities in the case that algorithms designed with AGPU(v, b, M) is executed on AGPU(p, b, M).

Theorem 3.9 *Supposing $v > p$, for the volatile AGPU model,*

$$\begin{aligned} AGPU_{IO}(p, b, M) &= AGPU_{IO}(v, b, M) \\ AGPU(p, b, M) &\leq \left\lceil \frac{v}{p} \right\rceil AGPU(v, b, M) \end{aligned}$$

Proof. Due to Lemma 3.4, AGPU$_{IO}(p, b, M)$ = AGPU$_{IO}(v, b, M)$ = AGPU$_{IO}(b, b, M)$. We consider the time complexity. In AGPU(v, b, M), the number of multiprocessors used by the algorithm is v/b, whereas, in AGPU(p, b, M), the number of multiprocessors is p/b. Therefore, the ratio of the number of multiprocessors is at most $\lceil v/p \rceil$. Suppose the multiprocessors on AGPU(p, b, M) simulate the multiprocessors on AGPU(v, b, M). Since the multiprocessors on the AGPU model have no means of communication with each others and all data in the shared memory are deleted at the time of synchronization, a multiprocessor on AGPU(p, b, M) can always execute the same instructions as a multiprocessor on AGPU(v, b, M). Therefore, the multiprocessors on AGPU(p, b, M) can simulate the multiprocessors on AGPU(v, b, M) by simulating a multiprocessor on AGPU(v, b, M) at most $\lceil v/p \rceil$ times. If the time complexity varies by multiprocessors, the largest complexity is adopted. Therefore, the time complexity on AGPU(p, b, M) can be smaller than $\lceil v/p \rceil$ factor of the time complexity on AGPU(v, b, M). □

When we develop the algorithms taking multithreading into account, the number of threads in the algorithms must be larger than the number of cores. We can estimate the time and I/O complexities of the algorithms by applying Theorem 3.9. For example, if the time complexity of an algorithm on AGPU(v, b, M) is $O(n/v + \log v)$, the time complexity in case that the algorithm is executed on AGPU(p, b, M) is $O(n/p+v \log v/p)$.

3.4 Guideline for Developing Efficient Algorithms Using the AGPU Model

We provide a guideline to design efficient algorithms using the AGPU model. First of all, the I/O complexity (the number of global memory accesses) should be reduced as much as possible because the execution time of a global memory access instruction is much larger than others. As shown in Section 3.3.3, lower bounds on the I/O model also give lower bounds on the I/O complexity in the AGPU model. Therefore efficient I/O model based algorithms will be bases of efficient AGPU-based algorithms. Next, we should make the multiplicity as large as possible by reducing the amount of shared memory used. It makes multithreading effective. To design efficient algorithms executed on a multiprocessor, known PRAM algorithms can be used (see Section 3.3.1).

3.4.1 Implementations Using CUDA

We discuss the implementation of the algorithms designed with the AGPU model. When we implement programs using CUDA, the followings are effective to make the programs speed up.

1. The data referred from a single core are moved to the register.
2. The number of the warps in a block is increased to more than one.
3. The communication inside a block is done using the shared memory.

If we use the register instead of the shared memory, we can make memory access fast. Moreover, the amount of shared memory used is reduced, which leads to make the multiplicity large. When the number of a block in a multiprocessor is one, the number of warps in a multiprocessor is limited because the number of warps assigned to a block is limited. Therefore, using more than one blocks also makes the multiplicity large. We can reduce the I/O complexity by the last item. It is also effective for performance improvements to adjust the number of threads and blocks depending on hardware architectures.

3.5 Short Summary

In this chapter, we proposed a new computational model AGPU in order to analyze asymptotic computational complexities of GPU-based algorithms. Algorithms on the AGPU model can be evaluated using five metrics, the time complexity, the I/O complexity, the multiplicity, the amount of shared memory used, and the amount of global memory used. The model is simple and able to take account of a lot of factors affecting the performance such as coalescing, bank conflicts, multithreading. Complexities on GPUs depend on device specifications, but they are within a constant factor of complexities on the AGPU model.

After describing the architecture, we revealed the relation between the AGPU model and other models including the PRAM model, the BSP model, and the I/O model. It enables us to utilize some knowledge on the other model for the AGPU model.

Lastly, We provide a guideline to design efficient algorithms using the AGPU model according to the discussion so far.

4 Reduction Algorithms

4.1 Introduction

As the first example of complexity analyses using the AGPU model, we deal with reduction algorithms. Reduction is a basic operation for arrays. Despite the simplicity of the operation, it is not easy to develop fast reduction algorithms on GPUs because it is necessary to take the GPU architectures into account. If one does not know the GPU architecture, one cannot evaluate which kind of algorithms are suitable for GPUs.

Reduction plays a very important role in parallel programming. After giving the formal definition, we explain many problems can be described as reductions especially when we use non-commutative operators. We then analyze the complexities of some reduction algorithms. We deal with both commutative and non-commutative operators. There exist two main algorithms for reduction with commutative operators: tree-based algorithm and cascading algorithm. The latter is faster than the former in practice. We give evidence using the AGPU model; the latter has lower time complexity than the former. We next deal with non-commutative operators. Since we cannot use the cascading algorithm for non-commutative operators, we give a novel and efficient algorithm. Our algorithm has the same complexities as the cascading algorithm. We finally evaluate these algorithms on real GPUs and show that our algorithm is fast not only in theory but also in practice.

4.2 Definition

Given an array $T[0..n-1]$ of n elements, *reduction* $r(T, \oplus)$ is defined as

$$r(T, \oplus) := \bigoplus_{i=0}^{n-1} T[i],$$

where the operator $\oplus$ is associative in this dissertation. The right-hand side of the equation indicates that all elements in the array are reduced to one using operator $\oplus$, informally,

$$\bigoplus_{i=0}^{n-1} T[i] = T[0] \oplus T[1] \oplus \cdots \oplus T[n-1].$$

For instance, $r(T, +)$ represents the summation of all the elements in an array T. We assume the input array is allocated on the global memory of GPUs and each element of the array stores a w bit number. We discuss both commutative and non-commutative operators.

4.3 Reduction as a Programming Framework

Many problems for arrays can be described as reductions. Bird [28] revealed what kind of problems are characterized as reductions. As explained later, a function can be converted to a reduction if and only if the function is a homomorphism with respect to list concatenation. In this section, we explain this property and give some examples.

4.3.1 Lists

Lists are sequences of elements such that all elements are included in the same set. *Arrays* are one implementation of lists whose elements are indexed. Normally, memory addresses of the elements are continuously allocated in order of indices. *List functions* are the functions that take lists as arguments. In this section, we discuss several properties of list functions.

Basic Definition

Let S be a finite set. A function *Singleton* maps elements of S into *singleton lists.* Given $b \in S$, the return value of the function is represented as $[b]$, namely,

$$Singleton(b) = [b].$$

Let $U = \{[b] \mid b \in S\}$, that is, U consists of all singleton lists on S. We next define an associative operation on the singleton lists. The operator $+\!\!+$ represents *concatenation* of two singleton lists. For $a, b \in S$, the result of $[a] +\!\!+ [b]$ is represented as $[a, b]$. The operator is also defined on the results of the operation. Since we assume the operator is associative, we have $([a] +\!\!+ [b]) +\!\!+ [c] = [a] +\!\!+ ([b] +\!\!+ [c])$ for $a, b, c \in S$. Thus, we can use the following notation:

$$[a] +\!\!+ [b] +\!\!+ [c] = [a, b, c].$$

Let $[S]^+$ be the closure of U on the operator $+\!\!+$. In other words, $[S]^+$ consists of all elements generated by repeatedly applying the operator $+\!\!+$ to singleton lists. Let $[\,]$ be the identity of the operator $+\!\!+$. It is called the *empty list.* In other words, for all $x \in [S]^+$, we have

$$x +\!\!+ [\,] = [\,] +\!\!+ x = x.$$

We define $[S]$ as follows:

$$[S] = [S]^+ \cup \{\ [\,]\ \}.$$

For example, assuming $S = \{a, b\}$, we have

$$[S]^+ = \{[a], [b], [a, a], [a, b], [b, a], [b, b], [a, a, a], \cdots\},$$

$$[S] = \{[\,], [a], [b], [a, a], [a, b], [b, a], [b, b], [a, a, a], \cdots\}.$$

The elements of $[S]$ are called *lists* on S.

By this definition, the pair $([S], +\!\!+)$ is a *monoid*, that is, the following properties hold:

Associativity: For all $x, y, z \in [S]$, $(x +\!\!+ y) +\!\!+ z = x +\!\!+ (y +\!\!+ z)$,

Identity: For all $x \in [S]$, $[\,] +\!\!+ x = x +\!\!+ [\,] = x$.

4.3.2 List Homomorphism

Let $h(x)$ be a function from $([S], +\!\!+)$ to another monoid $(T, \oplus)$ such that the identity is $id_\oplus$ and the operator $\oplus$ is associative. The function h is called a *list homomorphism* if there exists a function f, and the function h can be specified by the following equations:

$$h(x) = \begin{cases} id_\oplus, & \text{if } x = [\,], \\ f(a), & \text{if } x \text{ is a singleton list represented as } [a], \\ h(y) \oplus h(z), & \text{if } x = y +\!\!+ z \text{ for some } y, z \in [S]^+. \end{cases}$$

Note that the argument $x \in [S]$ always satisfies one of these three conditions. As explained later in Section 4.3.4, these equations specify a unique function regardless of the way to determine y, z due to the associativity of the operators $+\!\!+, \oplus$. For example, the function $\mathrm{count}(x)\colon [S] \to \mathbb{Z}^+$ can be specified as follows:

$$count(x) = \begin{cases} 0, & \text{if } x = [\,], \\ 1, & \text{if } x \text{ is a singleton list}, \\ count(y) + count(z), & \text{if } x = y +\!\!+ z \text{ for some } y, z \in [S]^+. \end{cases}$$

Given a list, this function counts the number of elements in the list.

The list homomorphism indicates that the function can be calculated using a divide-and-conquer method. Accordingly, it is well known that many functions are list homomorphisms [28, 29, 30]. For example, the merge sort can be described as a list homomorphism. In that case, the output of $h(x)$ is a list. In this dissertation, we only deal with the case where the output of $h(x)$ is not a list but a value. We still have many examples, some of which we will introduce later.

4.3.3 Higher Order Functions on Lists

If a function takes functions as arguments or returns functions, it is called a *higher order function*. In this section, we introduce some higher order functions on lists.

Map

The function $map(f, x)$ takes a function f and a list in $[S]$ as arguments and returns a list in a monoid $([R], +\!\!+)$, where $[R]$ is the lists on a set R, and f is a function from S to R. It is specified as follows:

$$map(f, x) = \begin{cases} [\,], & \text{if } x = [\,], \\ [f(a)], & \text{if } x \text{ is a singleton list represented as } [a], \\ map(f, y) +\!\!+ map(f, z), & \text{if } x = y +\!\!+ z \text{ for some } y, z \in [S]^+. \end{cases}$$

Informally, we have

$$map(f, [a_1, a_2, a_3, \cdots, a_n]) = [f(a_1), f(a_2), f(a_3), \cdots, f(a_n)].$$

Again, the return value is independent of the way to determine y, z due to the associativity of the operator $+\!\!+$.

Reduce

The function $reduce(\oplus, x)$ takes a function $\oplus$ and a list x as arguments and returns a value in S, where $\oplus$ is an associative function from $(S \times S)$ to S. It is specified as follows:

$$reduce(\oplus, x) = \begin{cases} id_{\oplus}, & \text{if } x = [\,], \\ a, & \text{if } x \text{ is a singleton list represented as } [a], \\ reduce(\oplus, y) \oplus reduce(\oplus, z), & \text{if } x = y +\!\!+ z \text{ for some } y, z \in [S]^+. \end{cases}$$

Informally, we have

$$reduce(\oplus, [a_1, a_2, a_3, \cdots, a_n]) = a_1 \oplus a_2 \oplus a_3 \oplus \cdots \oplus a_n.$$

The return value is also independent of the way to determine y, z due to the associativity of the operator $\oplus$.

4.3.4 List Homomorphism Lemma

The following lemma shows that the list homomorphisms can be calculated using maps and reductions.

Lemma 4.1 (List homomorphism lemma [31]) *A function h that takes a list x as an argument is a list homomorphism if and only if there exist a operator $\oplus$ and a function f such that*

$$h(x) = \mathit{reduce}(\oplus, \mathit{map}(f, x)).$$

Informally, we have

$$h([a_1, a_2, a_3, \cdots, a_n]) = f(a_1) \oplus f(a_2) \oplus f(a_3) \oplus \cdots \oplus f(a_n).$$

We discuss the parallel computation of the list homomorphisms. The map function has a trivial data parallelism because each element of a list can be executed independently. Therefore, it is important to develop efficient parallel algorithms for the reduction function.

4.4 Examples of Reduction

We give some examples of the reduction. For commutative operators, the examples include sum of list, maximum element in the list, and the size of the list.

We next give some examples for non-commutative operators. The first example is matrix multiplication. Given a list of square matrices of the same size, the reduction multiplies all matrices and outputs the resulting matrix. This operation is used for many applications. For example, Hongo et al. [32] utilized this operation for random value generations. In the next section, we give an example such that a divide-and-conquer algorithm to solve the problem is well known.

4.4.1 Maximum Segment Sum Problem

In this section, we deal with the maximum segment sum (MSS) problem.

Problem 4.2 (Maximum Segment Sum) *Given a list of real numbers, compute the maximum sum found in any contiguous segment of it.*

For example, given a list $[3, -1, -4, 1, 5, -9, 2]$, the answer is 6. The segment contributing to the answer is $[1, 5]$. If all elements in the list are negative, the answer is 0. In this case, the corresponding segment is empty. This problem appears in many applications. In the field of data mining, this problem is used to calculate one dimensional association rules [33]. Bentley [34] introduced a linear time algorithm for MSS. We introduce a slightly modified version introduced by Cole [29] in order to fit the algorithm to the reduction framework.

We give a divide-and-conquer algorithm for this problem. Instead of calculating the maximum segment sum directly, we calculate a tuple including it. We first generate a tuple for each input element and then repeat merging the tuples until the number of tuples becomes one.

Given a list $x[0..n-1]$, $x[\ell..r]$ denotes the segment (sub-list) that ranges from index ℓ to index r, and $sum(x[\ell..r])$ denotes $\Sigma_{k=\ell}^{r} x[k]$. The function $mss'(x)$ that calculates the tuple is represented by

$$mss'(x) = (mss(x), sum(x), mts(x), mis(x)) .$$

The function $mss(x)$ returns the maximum segment sum, which is defined as follows:

$$mss(x) = max2\left(\max_{\ell \le r} sum(x[\ell..r]), 0\right),$$

where the function $max2(a, b)$ returns maximum value out of two arguments a, b. The function $mts(x)$ is called the maximum tail segment sum and defined as follows:

$$mts(x) = max2\left(\max_{i} sum(x[i..n-1]), 0\right).$$

The function $mis(x)$ is called the maximum initial segment sum and defined as follows:

$$mis(x) = max2\left(\max_{i} sum(x[0..i]), 0\right).$$

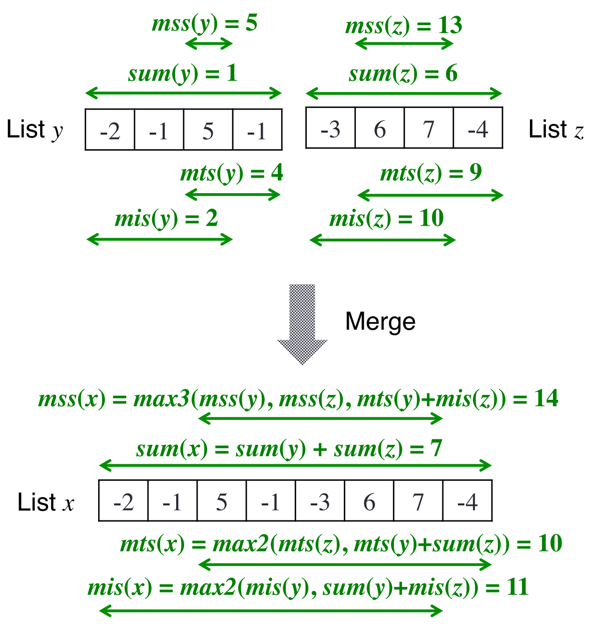

mss(y) = 5
mss(z) = 13
sum(y) = 1
sum(z) = 6
List y
-2 -1 5 -1
-3 6 7 -4
List z
mts(y) = 4
mts(z) = 9
mis(y) = 2
mis(z) = 10
Merge
mss(x) = max3(mss(y), mss(z), mts(y)+mis(z)) = 14
sum(x) = sum(y) + sum(z) = 7
List x
-2 -1 5 -1 -3 6 7 -4
mts(x) = max2(mts(z), mts(y)+sum(z)) = 10
mis(x) = max2(mis(y), sum(y)+mis(z)) = 11

Figure 4.1: An example of the merge operation for the maximum segment problem

For a singleton list $[b]$, we can calculate the tuple as follows:

$$\begin{aligned} mss'([b]) &= (mss([b]), sum([b]), mts([b]), mis([b])) \\ &= (max2(b, 0), b, max2(b, 0), max2(b, 0))\,. \end{aligned}$$

Let f be the function that generates a tuple from an element b in a list. Then, we have

$$f(b) = (max2(b, 0), b, max2(b, 0), max2(b, 0))\,.$$

Next we define the function $\oplus$ that merges two tuples. Figure 4.1 shows an example. Given two lists y, z, and the corresponding tuples $mss'(y)$, $mss'(z)$, the operator $mss'(y) \oplus mss'(z)$ is defined as follows:

$$\begin{aligned} mss'(y) \oplus mss'(z) = (&max3\,(mss(y), mss(z), mts(y) + mis(z))\,, \\ &sum(y) + sum(z), \\ &max2\,(mts(z), mts(y) + sum(z))\,, \\ &max2\,(mis(y), sum(y) + mis(z))), \end{aligned}$$

where the function $max3(a, b, c)$ returns maximum value out of three arguments a, b, c. Note that this operator is not commutative, namely, $y \oplus z \neq z \oplus y$ in general. We can see that this tuple corresponds to $mss'(y +\!\!+ z)$, namely,

$$mss'(y +\!\!+ z) = mss'(y) \oplus mss'(z).$$

Thus, the function $mss'(x)$ is a list homomorphism, and we can solve the maximum segment sum problem using a divide-and-conquer algorithm.

According to the Lemma 4.1, the function $mss'(x)$ is represented as follows:

$$mss'(x) = reduce\,(\oplus, map\,(f, x))\,.$$

Therefore, we can utilize the reduction framework to calculate the function $mss'(x)$. We will show the experimental result on GPUs in Section 4.8.2.

4.5 Reduction with Commutative Operators

We now discuss GPU-based reduction algorithms using the AGPU model. In this section, we assume the operator $\oplus$ is commutative. We will deal with non-commutative operators in the next section. We describe two standard algorithms suggested by Harris [9] and analyze the time and I/O complexities and the amount of memory used. Harris introduced seven algorithms for reduction [9]. We can divide them into two types. The first six algorithms are called tree-based algorithms, and the last one is called a cascading algorithm. The cascading algorithm is faster than the six tree-based algorithms in a real GPU. In this section we show that the cascading algorithm has lower time complexity than the tree-based algorithms on the AGPU model.

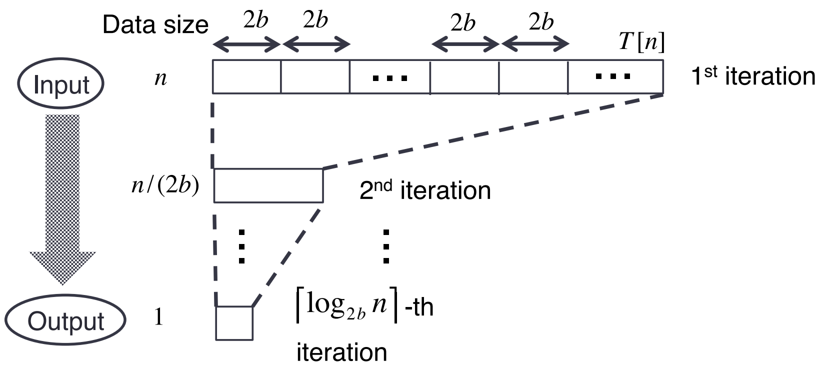


Figure 4.2: Outline of tree-based reduction algorithm.

We only explain the case where n is equal to or larger than p. When $n < p$, some cores are not used. In particular, when $n \leq b$, we always use a single multiprocessor. However, we do not go into the detail of this case because we cannot take full advantage of GPUs. We often consider the cases where the input size n is much larger than p. In this section, if $p = o(n)$ holds, we say that the input size is sufficiently larger than the number of cores.

4.5.1 Tree-based Algorithm

We describe the tree-based algorithm proposed by Harris [9]. He applied five optimization techniques to a naive algorithm step by step, and obtained six algorithms. We analyze the fastest algorithm among the six algorithms using AGPU(p, b, M). This algorithm contains all optimization techniques proposed by Harris.

Figure 4.2 shows the outline of tree-based algorithm. The input $T[0..n-1]$ is divided into blocks with $2b$ words and each block is reduced to one element using a single multiprocessor. After reducing all blocks, we obtain $n/2b$ elements. The same calculation is repeatedly done to the resulting values until the size of the elements becomes one. The result is the reduction value of the input.

We next explain the procedure for calculating each block. Figure 4.3 shows an example for the case where the operator is "addition" and b is equal to four. A multiprocessor reads the first half of a block from the global memory and stores them in the shared memory, and next, reads and stores the second half of a block

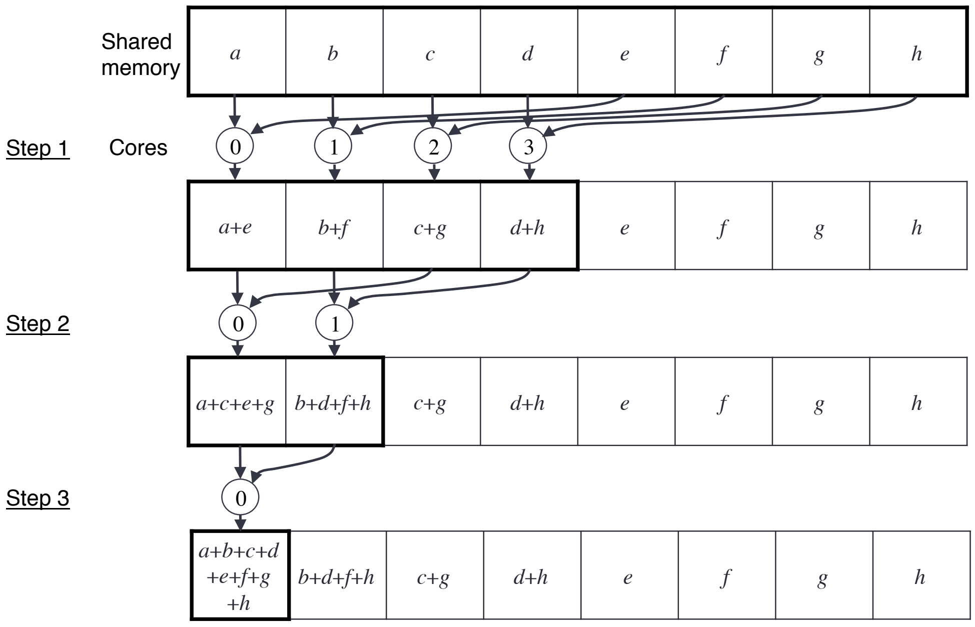


Figure 4.3: The procedure of each block for tree-based reduction algorithm.

similarly. Thus, $2b$ elements are stored in the shared memory. After that, b cores in the multiprocessor apply the operator to two elements in parallel (see Step 1 in Figure 4.3). The cores access contiguous elements in each step. At the first step, the i-th core carries out the operation to the i-th element and the $(i + b)$-th element in the block. Some cores repeat carrying out the operation to the resulting values until only one element remains. Since the cores access distinct banks, bank conflicts do not occur. Note that we can use this procedure only if the operator is commutative. In Figure 4.3, we actually calculate $a \oplus e \oplus c \oplus g \oplus b \oplus f \oplus d \oplus h$ instead of $a \oplus b \oplus c \oplus d \oplus e \oplus f \oplus g \oplus h$. If the operator is not commutative, these two values are not the same in general. A pseudo code for tree-based algorithm is shown in Algorithm 4.1.

We analyze the time complexity of this tree-based algorithm using the AGPU model. The loop on lines 14-17 runs $\log b$ times. So it takes $O(\log b)$ times to calculate the reduction for one block. The loop on lines 5-27 runs $\lceil \log_{2b} n \rceil$ times. Let k be the number of multiprocessors and $s(i)$ represent how many times the loop in lines 7-23

Algorithm 4.1 Calculate reduction using the tree-based algorithm

```
1:  procedure CalculateReductionUsingTreeBased($T$, $n$)    ▷ Given an array $T[n]$
2:      $Q := \&T[0]$
3:      $\Omega := \&W[0]$                    ▷ Buffer to store temporary reduction values
4:      $d \leftarrow n$
5:      while $d > 1$ do
6:          $s \leftarrow \lceil d/2kb \rceil$                    ▷ The number of serialization
7:          for $j \leftarrow 0$ to $s - 1$ do
8:              for all $\rho \in \mathrm{MP}[0..k-1]$ in parallel do
9:                  for all $\epsilon \in \mathrm{Core}[0..b-1]$ in parallel do
10:                     $d_1[\epsilon] \Leftarrow Q[2b(jk+\rho)+\epsilon]$
                            ▷ Each multiprocessor reads the first half of the $2b$ elements
11:                     $d_2[\epsilon] \Leftarrow Q[2b(jk+\rho)+b+\epsilon]$
                            ▷ Each multiprocessor reads the second half of the $2b$ elements
12:                     $x[\epsilon] \leftarrow d_1[\epsilon] \oplus d_2[\epsilon]$
13:                     $\delta \leftarrow b/2$
14:                     while $\delta > 0$ do
15:                         $x[\epsilon] \leftarrow x[\epsilon] \oplus x[\epsilon+\delta]$
16:                         $\delta \leftarrow \delta/2$
17:                     end while
18:                     if $\epsilon = 0$ then
19:                         $\Omega[jk+\rho] \Leftarrow x[0]$
                    ▷ Each multiprocessor writes the reduction value of the $2b$ elements
20:                     end if
21:                 end for
22:             end for
23:         end for
24:         $Q := \Omega$    ▷ The same calculation is repeatedly done to the resulting values
25:         $d \leftarrow \lceil d/2b \rceil$                ▷ The number of elements is reduced to $1/2b$
26:         $\Omega := \&\Omega[d]$
27:     end while
28:     return $Q[0]$                                   ▷ The reduction value of $T[n]$
29: end procedure
```

are executed at the i-th iteration in lines 5-27. We have

$$\begin{aligned} s(i) &= \left\lceil \frac{n}{(2b)^i} \frac{1}{k} \right\rceil \\ &= \left\lceil \frac{n}{2p(2b)^{i-1}} \right\rceil . \end{aligned}$$

Therefore, the time complexity of the algorithm is

$$\begin{aligned} \sum_{i=1}^{\lceil \log_{2b} n \rceil} s(i) \log b &< \sum_{i=1}^{\lceil \log_{2b} n \rceil} \log b \left(\frac{n}{2p(2b)^{i-1}} + 1 \right) \\ &= \frac{n \log b}{2p} \sum_{i=1}^{\lceil \log_{2b} n \rceil} \left(\frac{1}{2b} \right)^{i-1} + \log b \left\lceil \log_{2b} n \right\rceil . \end{aligned}$$

Since $b \geq 1$, we have

$$\begin{aligned} \sum_{i=1}^{\lceil \log_{2b} n \rceil} \left(\frac{1}{2b} \right)^{i-1} &\leq \sum_{i=1}^{\infty} \left(\frac{1}{2b} \right)^{i-1} \\ &= \frac{1}{1 - \frac{1}{2b}} \\ &\leq 2. \end{aligned}$$

Thus, the time complexity is $O((n \log b)/p + \log n)$. If the data size is sufficiently larger than the number of cores, the time complexity is $O((n \log b)/p)$.

We next analyze the I/O complexity. Since each block is accessed three times in the loop of lines 9-21 and the number of blocks is $\left\lceil n/(2b)^i \right\rceil$, the multiprocessors access the global memory $3 \left\lceil n/(2b)^i \right\rceil$ times at the i-th iteration. Therefore, the I/O complexity is

$$\begin{aligned} \sum_{i=1}^{\lceil \log_{2b} n \rceil} 3 \left\lceil \frac{n}{(2b)^i} \right\rceil &\leq \sum_{i=1}^{\lceil \log_{2b} n \rceil} 3 \left(\frac{n}{(2b)^i} + 1 \right) \\ &= \sum_{i=1}^{\lceil \log_{2b} n \rceil} \left(\frac{3n}{2b} \left(\frac{1}{2b} \right)^{i-1} + 3 \right) \\ &= O\left(\frac{n}{b} + \frac{\log n}{\log b} \right) . \end{aligned}$$

If the data size is sufficiently larger than the number of cores, the I/O complexity is $O(n/b)$.

Next, we analyze the amount of memory used in the algorithm. The shared memory is only used to store elements of a block. The amount of the shared memory used is therefore $2b$ words. The global memory is used to store the input, the output, and the temporary results. The input consists of n elements. We use the same buffer for output and temporary results. We have to store $n/(2b)$ elements to the global memory as a temporary result after the first iteration. The amount of the global memory used is therefore $n + n/(2b)$ words.

The multiplicity is immediately calculated with the amount of shared memory used. Since the amount of shared memory used is $2b$ words, The multiplicity is $M/(2b)$.

4.5.2 Cascading Algorithm

We can improve the tree-based algorithm. The tree-based algorithm repeatedly calculates a reduction value of a block, but it is not efficient because some cores are not used in the later steps of the calculation of each block. The cascading algorithm makes cores keep active until the final part of the algorithm.

We describe the cascading algorithm using AGPU(p, b, M). We consider the input sequence as a matrix with p columns. We use the row-major order, that is, the first p elements in the input array are stored in the first row. Each of p cores is assigned to one of the columns in the matrix. Figure 4.4 shows an example for $b = 4$ and $p = 12$.

Each core calculates the reduction of one column sequentially. After that, cores in a multiprocessor calculate the reduction of b resulting values in a multiprocessor and write the result to the global memory. We call this step "local reduction". After that, we calculate the reduction of p/b resulting values using the tree-based algorithm. We call this step "global reduction". As a result, we obtain the overall reduction value. Note that we can use this procedure only if the operator is commutative. Algorithm 4.2 shows a pseudo code for the cascading algorithm.

The details of the local reduction is shown in Procedure CalculateLocalReductionUsingCascading. Each multiprocessor repeats handling one row with b elements. In each iteration, it reads b contiguous elements from the global memory by one coalesced memory access instruction and each core applies the operator to

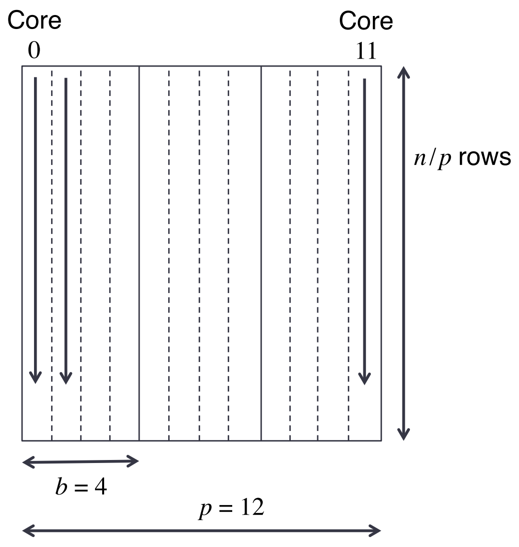


Figure 4.4: Input sequence arranged as a matrix with p columns.

an element and the resulting value so far. After handling all assigned elements, a multiprocessor has b resulting values. Finally, it reduces the b elements to one using the tree-based algorithm.

We analyze the time complexity. The local reduction takes $O(n/p + \log b)$ times because each multiprocessor handles all assigned elements in $O(n/p)$ times and reduces b resulting elements to one in $O(\log b)$ times. The global reduction can be computed in $O(\log k)$ times using the tree-based algorithm for the k resulting values. The time complexity is therefore $O(n/p + \log p)$. If the data size is sufficiently larger than the number of cores, the time complexity is $O(n/p)$.

Next, we analyze the I/O complexity. Since each multiprocessor always reads b elements per instruction, the local reduction accesses the global memory $O(n/b)$ times. The global reduction accesses the global memory $O(k/b)$ times using the tree-based algorithm. Since we assume $n \geq p$, the I/O complexity is $O(n/b)$.

Finally, we analyze the amount of memory used and the multiplicity. The amount of the shared memory used is $O(b)$ words because each core stores two elements in the shared memory. The amount of the global memory used is $n + O(p/b)$ words because each multiprocessor has to store one word to the global memory as a temporary result. The multiplicity is therefore $M/(2b)$.

Algorithm 4.2 Calculate reduction using the cascading algorithm

```
1: procedure CalculateReductionUsingCascading($T$,$n$)    ▷ Given an array $T[n]$
2:     $W[0..p/b-1]$ = CalculateLocalReductionUsingCascading($T, n$)
                                                              ▷ Local reduction
3:     return CalculateReductionUsingTreeBased($W, p/b$)      ▷ Global reduction
4: end procedure

5: procedure CalculateLocalReductionUsingCascading($T$,$n$)
                                                         ▷ Given an array $T[n]$
6:     for all $\rho \in$ MP$[0..k-1]$ in parallel do
7:         for all $\epsilon \in$ Core$[0..b-1]$ in parallel do
8:             $x[\epsilon] \leftarrow 0$                  ▷ Reduction value so far
9:             for $i \leftarrow 0$ to $n/p-1$ do
10:                $d[\epsilon] \Leftarrow T[ip+b\rho+\epsilon]$   ▷ Each multiprocessor reads the $i$-th row
11:                $x[\epsilon] \leftarrow x[\epsilon] \oplus d[\epsilon]$
12:            end for
13:            $\delta \leftarrow b/2$
14:            while $\delta > 0$ do
15:                $x[\epsilon] \leftarrow x[\epsilon] \oplus x[\epsilon+\delta]$
16:                $\delta \leftarrow \delta/2$
17:            end while
18:            if $\epsilon = 0$ then
19:                $W[\rho] \Leftarrow x[0]$
                       ▷ Each multiprocessor writes the reduction value so far
20:            end if
21:        end for
22:    end for
23:    return $W[0..p/b-1]$
       ▷ The local reduction values each of which is calculated by a multiprocessor
24: end procedure
```

To sum it up, we confirmed that the cascading algorithm is theoretically faster than the tree-based algorithm on the AGPU model. If the data size is sufficiently larger than the number of cores, the cascading algorithm is $O(\log b)$ times faster than the tree-based algorithm, while these algorithms have the same I/O complexity. In Section 4.8, we will show that the cascading algorithm is faster than the tree-based algorithm in practice.

4.6 Reduction with Non-commutative Operators

The cascading algorithm utilizes the commutativity of the reduction operators for efficient memory access. Therefore, the algorithms do not work for non-commutative operators. In this section, we propose a novel reduction algorithm for non-commutative operators. The basic idea is that the reduction procedure is divided into several pipeline stages and each core in a multiprocessor processes one pipeline stage. We call this *pipeline algorithm*.

We first modify the tree-based algorithm so that it works for non-commutative operators for later use. Although the algorithm causes two-way bank conflicts, the complexities do not increase. Next we try to modify the cascading algorithm such that it can work for non-commutative operators. The algorithm is called *matrix-based algorithm*. Despite the optimal time and I/O complexities of the algorithm, the algorithm is very slow on real GPUs. We analyze the algorithm using the AGPU model, and reveal that the performance bottleneck is low multiplicity. Then, we describe the detail of the pipeline algorithm.

4.6.1 Tree-based Algorithm for Non-commutative Operators

If the operators are not commutative, the tree-based algorithm explained in Section 4.5.1 do not return the correct answer in general. For example, in Figure 4.3, the algorithm actually calculates $a \oplus e \oplus c \oplus g \oplus b \oplus f \oplus d \oplus h$ instead of $a \oplus b \oplus c \oplus d \oplus e \oplus f \oplus g \oplus h$. We therefore change the data assignment to cores. The new data assignment is shown in Figure 4.5 and the pseudo code for the algorithm is shown in Algorithm 4.3. Since the number of elements in a block is $2b$, two-way bank conflicts occur at the step 1. For example, since b is equal to four in Figure 4.5, the elements “a” and “e” in the first line

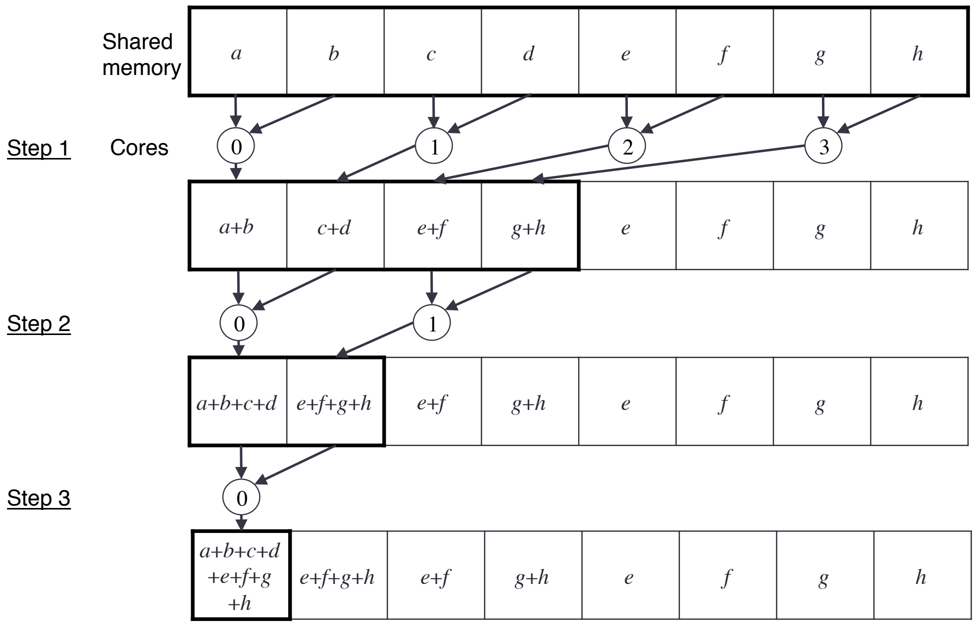


Figure 4.5: The procedure of each block for modified tree-based reduction algorithm.

are in the same bank, that is, Core 0 and Core 2 access the same bank at the same time. Similarly, Core 1 and Core 3 access the same bank at the same time. Although it makes the number of shared memory access twice, it does not change the asymptotic complexities. Accordingly, the time and I/O complexities and the multiplicity are the same as the original tree-based algorithm.

4.6.2 Matrix-based Algorithm

The cascading algorithm does not work for non-commutative operators. The matrix-based algorithm can be obtained by improving the cascading algorithm. Although the algorithm has the optimal time and I/O complexities, it is practically slow.

A typical way to develop GPU-based algorithms is to improve PRAM-based algorithms for GPUs. As mentioned in Section 3.3.1, PRAM-based algorithms are useful to develop algorithms executed inside a multiprocessor. In this section, we first develop the matrix-based algorithm in this manner. After that, we analyze the performance bottleneck.

As shown in Figure 4.6, we consider n input elements as a matrix with b columns

Algorithm 4.3 Calculate reduction using the modified tree-based algorithm

```
procedure CalculateReductionUsingTreeBased2($T$,$n$)   ▷ Given an array $T[n]$
    $Q := \&T[1]$
    $\Omega := \&W[1]$   ▷ Buffer to store temporary reduction values
    $d \leftarrow n$
    while $d > 1$ do
        $s \leftarrow \lceil d/2kb \rceil$   ▷ The number of serialization
        for $j \leftarrow 0$ to $s - 1$ do
            for all $\rho \in$ MP[0..$k - 1$] in parallel do
                for all $\epsilon \in$ Core[0..$b - 1$] in parallel do
                    $x[\epsilon] \Leftarrow Q[b(jk + \rho) + \epsilon]$ ▷ Each multiprocessor reads $b$ elements
                    for $j \leftarrow 0$ to $\log b - 1$ do
                        if $\epsilon < b/2^{j+1}$ then
                            $x[\epsilon] \leftarrow x[2\epsilon] \oplus x[2\epsilon + 1]$
                        end if
                    end for
                    if $\epsilon = 0$ then
                        $\Omega[jk + \rho] \Leftarrow x[0]$
                ▷ Each multiprocessor writes the reduction value of the $b$ elements
                    end if
                end for
            end for
        end for
        $Q := \Omega$   ▷ The same calculation is repeatedly done to the resulting values
        $d \leftarrow \lceil d/2b \rceil$   ▷ The number of elements is reduced to $1/2b$
        $\Omega := \&\Omega[d]$
    end while
    return $Q[0]$   ▷ The reduction value of $T[n]$
end procedure
```

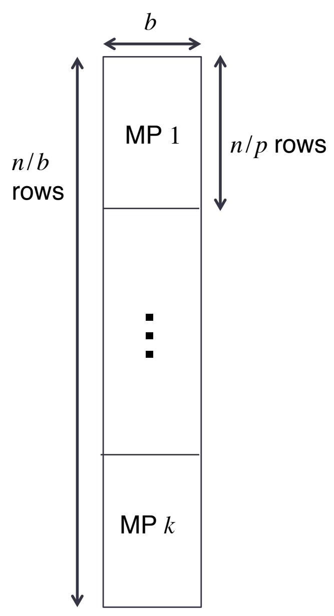


Figure 4.6: Input sequence arranged as a matrix with b columns.

and n/b rows in the row-major order. The n/b rows are divided into $k(= p/b)$ groups, each of which has n/p rows. Each multiprocessor calculates the reduction of one group.

In order to develop an efficient algorithm executed inside a multiprocessor on the AGPU model, we first develop a PRAM-based algorithm. In the cascading algorithm, each core handles one column. We cannot adopt it due to non-commutativity. Instead, we can design a simple algorithm on the PRAM model. As shown in Figure 4.7, we consider the input elements as a matrix with n/b columns and b rows in the row-major order. Each core handles one row and after that we reduce b resulting values to one. This algorithm works for non-commutative operators. The algorithm has the optimal time complexity $O(n/p)$ on the PRAM model.

Next, we improve this algorithm for the AGPU model. We first discuss a naive algorithm. A multiprocessor repeatedly accesses one column of the matrix shown in Figure 4.7 and each core in the multiprocessor calculates reduction of one row. The I/O complexity of this algorithm on the AGPU model is n, which is b times larger than the optimal value. This inefficiency is caused by non-coalesced global memory access, namely, we need b memory accesses to obtain b input elements. We next improve

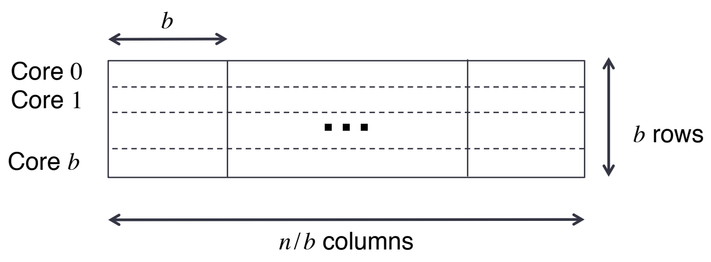


Figure 4.7: Input sequence arranged as a matrix with b rows.

the I/O complexity of the algorithm. In order to obtain b input elements per global memory access, we repeat executing a square matrix with b columns and b rows. For each iteration, we first copy all elements in a square matrix from the global memory to the shared memory. Then, each core handles one row with b elements. In this algorithm, we can obtain b elements per global memory access because we can utilize coalesced memory access. Therefore, the I/O complexity of the algorithm is $O(n/b)$.

We furthermore improve this algorithm. As mentioned in Section 3.3.1, An algorithm on the AGPU model may have a larger time complexity than the algorithm on the PRAM model for the same problem due to the bank conflicts. In the above algorithm, b-way bank conflicts occur because all cores in a multiprocessor access the same bank at the same time. We explain the detail using an example. Figure 4.8(a) shows the memory allocation at $b = 4$. The data addresses are allocated in the row-major order, namely, the addresses 0, 1, 2, and 3 are in the first row. In this figure, the columns represent banks. For example, the addresses 0, 4, 8, 12 are in the same bank. In the above algorithm, the i-th element is allocated to the address i. In this case, four elements in the same bank are accessed at the same time, that is, four-way bank conflicts occur. We can avoid the bank conflicts by changing the memory allocation. Figure 4.8(b) shows the improved memory allocation at $b = 4$. Formally, the i-th element is allocated to the address $\lfloor i/b \rfloor \times b + ((i\%b) + \lfloor i/b \rfloor)\,\%b$. As a result, this algorithm has the optimal time and I/O complexities. The pseudo code for the algorithm is shown in Algorithm 4.4. The global reduction is calculated using Algorithm 4.3.

Next, we analyze the multiplicity of this algorithm. Since this amount of shared memory used is $O(b^2)$, the multiplicity is $O(n/b^2)$. It seems to be difficult to increase this value without increasing the time and I/O complexities. The low multiplicity is

Algorithm 4.4 Calculate reduction using the matrix-based algorithm

```
1: procedure CalculateReductionUsingMatrixBased($T$,$n$)   ▷ Given an array $T[n]$
2:     $W[0..p/b-1]$ = CalculateLocalReductionUsingMatrixBased($T, n$)
                                                          ▷ Local reduction
3:     return CalculateReductionUsingTreeBased2($W, p/b$)    ▷ Global reduction
4: end procedure

5: procedure Conv(i)
6:     return $\lfloor i/b \rfloor \times b + ((i\%b) + \lfloor i/b \rfloor)\,\%b$   ▷ Convert from index to address
7: end procedure

8: procedure CalculateLocalReductionUsingMatrixBased($T$,$n$)
                                                          ▷ Given an array $T[n]$
9:     for all $\rho \in$ MP[0..$k-1$] in parallel do
10:        for all $\epsilon \in$ Core[0..$b-1$] in parallel do
11:            $x[\epsilon] \leftarrow 0$                     ▷ Reduction value so far
12:            for $i \leftarrow 0$ to $\lceil \lceil n/p \rceil /b \rceil - 1$ do
13:                for $j \leftarrow 0$ to $b-1$ do
14:                    $d[\text{Conv}(jb+\epsilon)] \Leftarrow T[\rho \lceil nb/p \rceil + \lceil n/p \rceil \times j + i \times b + \epsilon]$
15:                end for
16:                for $j \leftarrow 0$ to $b-1$ do
17:                    $x[\epsilon] \leftarrow x[\epsilon] \oplus d[\text{Conv}(j+\epsilon b)]$
18:                end for
19:            end for
20:            $\delta \leftarrow b/2$
21:            while $\delta > 0$ do
22:                $x[\epsilon] \leftarrow x[2\epsilon] \oplus x[2\epsilon+1]$
23:                $\delta \leftarrow \delta/2$
24:            end while
25:            if $\epsilon = 0$ then
26:                $W[\rho] \Leftarrow x[0]$
                       ▷ Each multiprocessor writes the reduction value so far
27:            end if
28:        end for
29:    end for
30:    return $W[0..p/b-1]$
       ▷ The local reduction values each of which is calculated by a multiprocessor
31: end procedure
```

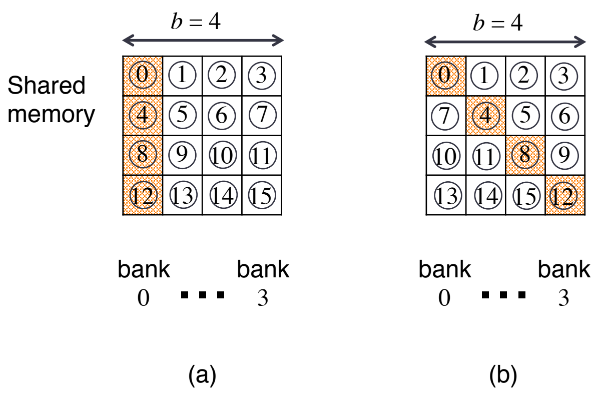


Figure 4.8: Examples of memory allocation for the matrix-based algorithm; (a) four-way bank conflicts occur, (b) no bank conflicts occur.

considered a main reason why the matrix-based algorithm is much slower than the cascading algorithm. We will show the experimental result in Section 4.8.

4.6.3 Pipeline Algorithm

We now describe our pipeline algorithm. As is the case with the matrix-based algorithm, the input array is represented as a matrix with b columns and n/b rows in the row-major order (see Figure 4.6). The n/b rows are divided into k groups with n/p rows. Each multiprocessor calculates the reduction of a group using $2b$ words of the shared memory.

Figure 4.9 shows the memory assignment at $b = 4$. The elements in the shared memory are numbered from 0 to $2b - 1$, which are represented by ⓪, ①, ②, $\cdots$ in Figure 4.9. The first b elements store values copied from the global memory, and the remaining b elements store values in progress.

Figure 4.10 shows the process of the reduction at $b = 4$. We suppose that each core carries out one operation at each timestamp $t = t_1, t_2, \cdots$. When the index of timestamps is odd, the cores read b elements from the global memory. Otherwise, the cores reduce two elements to one in parallel.

In Figure 4.9, the cores first copy four values to ⓪①②③ at $t = t_1$. Then Cores 0 and 1 reduce the four values to two and store the resulting values to ④⑤ at $t = t_2$. At $t = t_4$, Core 2 reduces ④ and ⑤ to one and stores it to ⑥. Finally at $t = t_6$, Core 3

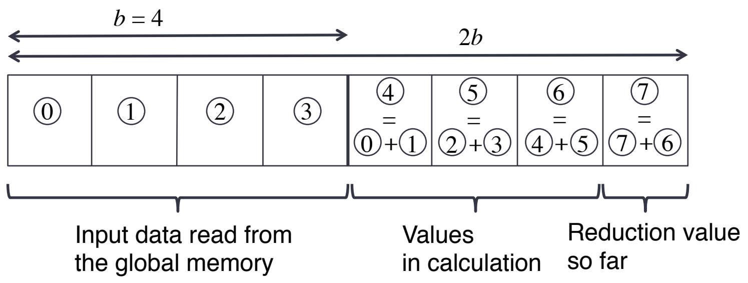
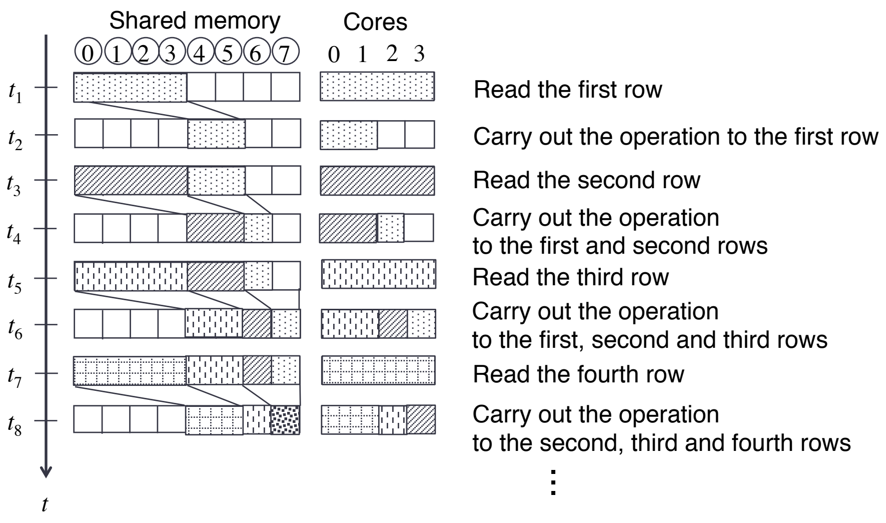


Figure 4.9: Memory assignment of the pipeline algorithm at $b = 4$.



Shared memory
⓪①②③④⑤⑥⑦
Cores
0 1 2 3
t_1 Read the first row
t_2 Carry out the operation to the first row
t_3 Read the second row
t_4 Carry out the operation to the first and second rows
t_5 Read the third row
t_6 Carry out the operation to the first, second and third rows
t_7 Read the fourth row
t_8 Carry out the operation to the second, third and fourth rows
⋮
t

Figure 4.10: An example of the procedure of the pipeline algorithm.

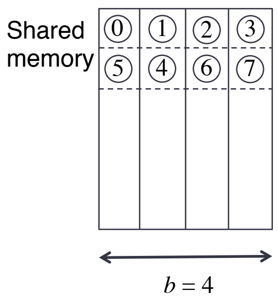


Figure 4.11: An example of the shared memory layout to avoid bank conflicts.

reduces ⑥ and the reduced value so far, which is the identity at first, and stores the resulting value to ⑦. The second row in the input matrix is processed similarly at time t_3, t_4, t_6, t_8, and the third row is processed at time t_5, t_6, t_8, t_{10}. We can see all cores are always active at $t \geq 5$. We can design the algorithm for any b represented as powers of two. On the other hand, when we calculate the reduction of one block using the tree-based algorithm, the number of active cores decreases as the calculation proceeds. Therefore the pipeline algorithm is faster than the tree-based algorithm.

Next, we explain the shared memory layout to avoid bank conflicts. Let $addr(i)$ denote the data address of i-th element. We determine $addr(i)$ as follows:

$$addr(i) = \begin{cases} i, & \text{if } i < b/2 \text{ or } i \geq b-2, \\ i+1, & \text{if } b/2 \leq i < b-2 \text{ and } i \text{ is a even number}, \\ i-1, & \text{otherwise}. \end{cases}$$

Figure 4.11 shows an example of shared memory layout for $b = 4$.

In order to prove this arrangement does not cause bank conflicts, we prove there exists a one-to-one mapping from cores to banks at any time the cores access the shared memory. It is enough to prove the mapping is surjective. The cores access the shared memory with four different patterns.

1. When the cores store the input elements to the shared memory, core i handles the i-th element. For each bank β, the corresponding core is β.

2. When each core loads the first element to reduce, core i handles element $2i$ (at $i < b-1$) or element $2b-1$ (at $i = b-1$). If $i = b-1$, core i accesses address

Algorithm 4.5 Calculate reduction using the pipeline algorithm

```
1: procedure CalculateReductionUsingPipeline($T$,$n$)          ▷ Given an array $T[n]$
2:     $W[0..p/b-1]$ = CalculateLocalReductionUsingPipeline($T, n$)
                                                                  ▷ Local reduction
3:     return CalculateReductionUsingTreeBased2($W, p/b$)       ▷ Global reduction
4: end procedure
```

$addr(2b-1) = 2b-1$, which is stored in bank $b-1$. Therefore, core $b-1$ accesses bank $b-1$. If $i < b/2$, core i accesses address $addr(2i) = 2i$, which is stored in bank $2i$. Therefore, core i accesses bank $2i$. Otherwise (if $b/2 \leq i < b-1$), core i accesses address $addr(2i) = 2i+1$, which is stored in bank $2i+1-b$. Therefore core i accesses bank $2i+1-b$. To sum up, for each core i, if $i < b/2$, the core accesses bank $2i$, otherwise, the core accesses $2i+1-b$. Note that $2i+1-b$ is odd. Inversely, for each bank β, if β is even, the corresponding core is $\beta/2$, otherwise, the corresponding core is $(\beta+b-1)/2$.

3. When each core loads the second element to reduce, core i handles element $2i+1$ (at $i < b-1$) or element $2b-2$ (at $i = b-1$). We can handle this case in the same manner as the above case. For each bank β, if β is odd, the corresponding core is $(\beta-1)/2$, otherwise the corresponding core is $(\beta+b)/2$.

4. When the cores write the resulting value to the shared memory, the cores access discrete addresses, and all cores access the second row in the share memory (see Figure 4.11). Therefore, for each bank, there exists a corresponding core.

Thus, the mapping from the cores to the banks is surjective in all cases. Therefore, bank conflicts do not occur. Algorithm 4.5 shows a pseudo code for the pipeline algorithm. The details of the local reduction is shown in Algorithm 4.6. Algorithm 4.5 uses the tree-based algorithm for the global reduction. A pseudo code for the tree-based reduction algorithm with non-commutative operators is shown in Algorithm 4.3.

Next, we analyze the complexities of the algorithm. Lines 2-30 in Algorithm 4.6 can be computed in $O(n/p + \log b)$ time. The global reduction can be computed in $O(\log k)$ time using the tree-based algorithm for the k resulting values. Thus, the time complexity is $O(n/p + \log p)$. If the data size is sufficiently larger than the number of cores, the time complexity becomes $O(n/p)$. As is the case with the cascading

Algorithm 4.6 Calculate local reduction using the pipeline algorithm

```
procedure CalculateLocalReductionUsingPipeline($T$,$n$)
                                                        ▷ Given an array $T[n]$
    for all $\rho \in$ MP$[0..k-1]$ in parallel do
        for all $\epsilon \in$ Core$[0..b-1]$ in parallel do
            if $\epsilon < b/2$ then ▷ Index for the first and second elements to be reduced
                $i_1 \leftarrow 2 * \epsilon$
                $i_2 \leftarrow 2 * \epsilon + 1$
            else
                $i_1 \leftarrow 2 * \epsilon + 1$
                $i_2 \leftarrow 2 * \epsilon$
            end if
            if $\epsilon \geq b - 2$ then                ▷ Index for the resulting elements
                $i_r \leftarrow \epsilon + b$
            else if $(\epsilon \% 2) \neq 0$ then
                $i_r \leftarrow \epsilon + b - 1$
            else
                $i_r \leftarrow \epsilon + b + 1$
            end if
            for $j \leftarrow 0$ to $n/p - 1$ do
                $y[\epsilon] \Leftarrow T[\rho nb/p + bj + \epsilon]$        ▷ Copy input to $y[\epsilon]$
                $y[i_r] \leftarrow y[i_1] \oplus y[i_2]$         ▷ Reduce two elements to one
            end for
            for $j \leftarrow 0$ to $\log b - 1$ do
                $y[\epsilon] \leftarrow id_{\oplus}$
                $y[i_r] \leftarrow y[i_1] \oplus y[i_2]$
            end for
            if $\epsilon = 0$ then
                $W[\rho] \Leftarrow y[2b-1]$
                        ▷ Each multiprocessor writes the reduction value so far
            end if
        end for
    end for
    return $W[0..p/b-1]$
      ▷ The local reduction values each of which is calculated by a multiprocessor
end procedure
```

Table 4.1: Complexities and multiplicity of reduction algorithms on AGPU(p, b, M). Here n is the number of elements to be reduced. We assume $p = o(n)$. Cascading cannot be used with non-commutative operator, whereas the others can.

Algorithms	Time complexity	I/O complexity	Multiplicity
(Optimal)	$\Theta(n/p)$	$\Theta(n/b)$	–
Tree-based	$O((n \log b)/p)$	$O(n/b)$	$O(M/b)$
Cascading	$O(n/p)$	$O(n/b)$	$O(M/b)$
Pipeline (Ours)	$O(n/p)$	$O(n/b)$	$O(M/b)$
Matrix-based	$O(n/p)$	$O(n/b)$	$O(M/b^2)$

algorithm, the I/O complexity is $O(n/b)$. The amount of the shared memory used is $O(b)$ words and the amount of the global memory used is $n + O(p/b)$ words. Finally, the multiplicity is $O(M/b)$.

4.7 Summary of Complexities and Multiplicity

Table 4.1 summarizes the time and I/O complexities and multiplicity of reduction algorithms. All algorithms have the optimal I/O complexity $O(n/b)$, and all algorithms except the tree-based algorithm have the optimal time complexity $O(n/p)$. The tree-based algorithm has $\log b$ times larger time complexity than other algorithms. In addition, the multiplicity of all algorithms except the matrix-based algorithm is $O(M/b)$, while the multiplicity of the matrix-based algorithm is $O(M/b^2)$.

4.8 Experimental Evaluation

4.8.1 Running Time

We have implemented all reduction algorithms explained in this chapter using CUDA and have measured their running time using NVIDIA k20c GPU. The device consists of 2496 cores organized as 13 multiprocessors. The bandwidth of the global memory is $208 GB/s$. All algorithms use summation of integer as a reduction operator. In order to compare the performance of the algorithms, all algorithms use the same operators. Since the operator is commutative, all algorithms work correctly. An evaluation using

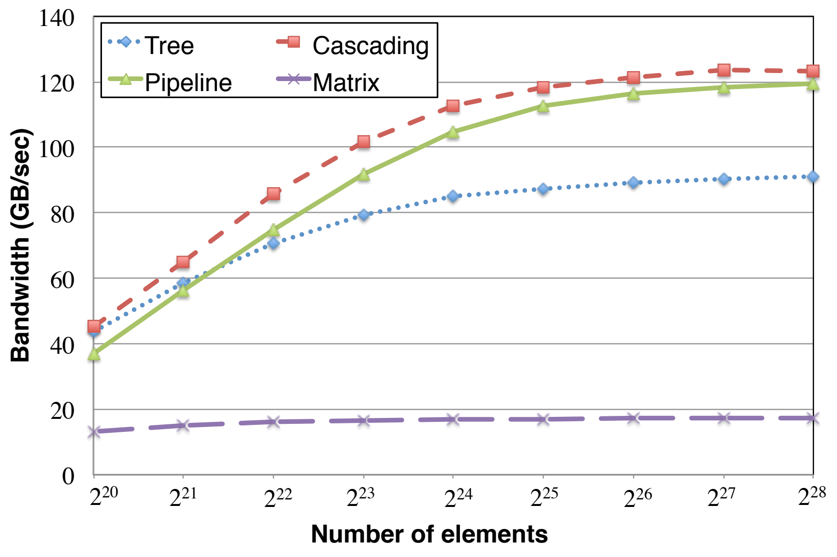


Figure 4.12: Running time of several reduction algorithms.

a non-commutative operator will be shown in the next section.

Figure 4.12 shows the bandwidth of the reduction algorithms, which represents the amount of elements processed per one second. Let "Tree", "Cascading", "Pipeline", "Matrix" denote tree-based algorithm, cascading algorithm, pipeline algorithm, matrix-based algorithm respectively. The bandwidth is limited by the bandwidth of the global memory.

The cascading algorithm is fastest among these algorithms. The pipeline algorithm is slower than the cascading algorithm, but sufficiently fast. The tree-based algorithm is slower than the cascading algorithm and the pipeline algorithm when n is larger than 2^{21}. This is considered due to the large time complexity. The matrix-based algorithm is slowest among these algorithms. This is considered due to the small multiplicity.

4.8.2 Maximum Segment Sum

As explained in Section 4.4.1, the maximum segment sum problem requires a non-commutative operator. We calculated the problem using the pipeline algorithm and the tree-based algorithm for non-commutative operators. The running time were

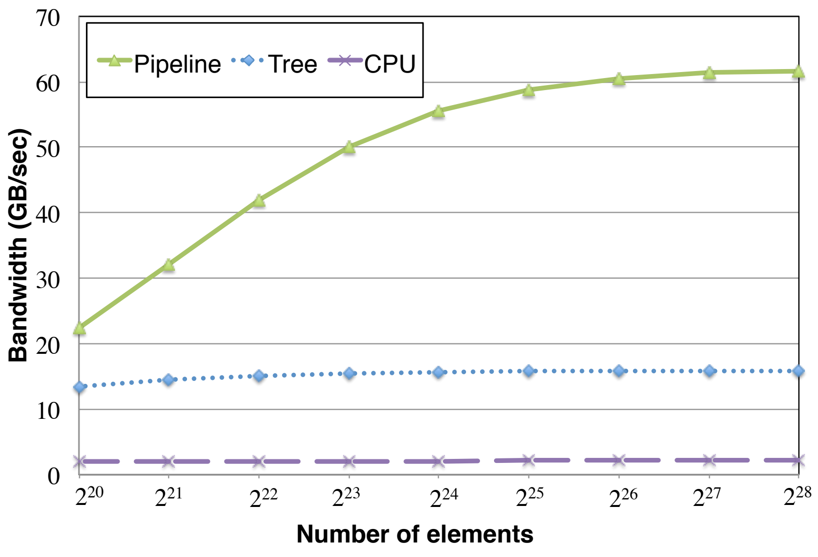


Figure 4.13: Running time of the MSS algorithms.

measured on NVIDIA k40c GPU. The device consists of 2880 cores organized as 15 multiprocessors. The bandwidth of the global memory is $288GB/s$. We also measured the running time on CPU. We used Xeon E5-1620 (3.7GHz) with 252 GB DDR. We utilized the linear-time sequential algorithm proposed by Bentley [34].

Figure 4.13 shows the bandwidth of the MSS algorithms. The pipeline algorithm is 3.9 times faster than the tree-based algorithm and 29 times faster than the sequential algorithm on CPU at $n = 2^{28}$.

4.9 Short Summary

We first explained that reduction plays a very important role in parallel programming. Divide-and-conquer algorithms can be formalized as list homomorphisms, and we can calculate list homomorphisms using reduction algorithms. Since some list homomorphisms require non-commutative operators, we need fast algorithms for it. As an example of list homomorphisms, we dealt with the maximum segment sum (MSS) problem. This algorithm also requires a non-commutative operator.

After that, we evaluated the several reduction algorithms. The cascading algorithm is fastest, but it works for only commutative operators. The tree-based algorithm works for non-commutative operators, but it is slower than the cascading algorithm. For non-commutative operators, we proposed the pipeline algorithm, which has the optimal time and I/O complexities. The algorithm is nearly as fast as the cascading algorithm on the real GPUs. With respect to the MSS problem, the pipeline algorithm is 3.9 times faster than the tree-based algorithm and 29 times faster than the sequential algorithm on CPU at $n = 2^{28}$. Thus, our algorithm is fast not only in theory but also in practice.

In addition, we evaluated the matrix-based algorithm to show the power of the AGPU model. Although the algorithm has the optimal time and I/O complexities on SIMD architectures, the algorithm is slow on GPUs. We can easily find the reason using the AGPU model; the reason is the small multiplicity. In this way, one can evaluate GPU-based algorithms using the AGPU model even if one is not familiar with GPU architectures.

5 Prefix Scan Algorithms

5.1 Introduction

In this chapter, we deal with prefix scan (prefix sum) algorithms. This chapter aims to show that we can predict the performance of the GPU-based algorithms using the AGPU model. As is the case with reduction, prefix scan is utilized by many other algorithms. After defining the prefix scan in Section 5.2, we show the applications of prefix scan in Section 5.3. In Section 5.4, we analyze the asymptotic behavior of prefix scan algorithms and reveal that the performance of the state-of-the-art algorithm heavily depends on a tuning parameter. In Section 5.5, we measure the actual running time of the algorithms with various parameter values and check that we can estimate the performance of the algorithms using the AGPU model.

5.2 Definition

Prefix scan or *prefix sum* is defined as follows. Given an array $T[0..n-1]$ of n elements, prefix scan returns an array $U[0..n-1]$ such that:

$$U[k] = \begin{cases} id_{\oplus}, & \text{if } k = 0, \\ \bigoplus_{i=0}^{k-1} T[i], & \text{otherwise,} \end{cases}$$

where the operator $\oplus$ is associative and commutative and $id_{\oplus}$ is the identity element for the operator. As stated in Chapter 4, the operator $\bigoplus$ indicates that all elements in the array are reduced to one using operator $\oplus$, informally,

$$\bigoplus_{i=0}^{k-1} T[i] = T[0] \oplus T[1] \oplus \cdots \oplus T[k-1].$$

This definition is also known as *exclusive scan* or *prescan*.

We introduce another definition of prefix scan. *Inclusive scan* is defined as follow:

$$U[k] = \bigoplus_{i=0}^{k} T[i].$$

We can generate the inclusive scan from the exclusive scan by shifting all elements of the exclusive scan to the left and inserting the sum of the input elements, which is calculated by adding the last element of the exclusive scan to the last element of the input, at the end. In this dissertation, we deal with exclusive scan unless otherwise noted.

5.3 Applications of Prefix Scan

Prefix scan is utilized by many applications. Blelloch [35] provides the list of the applications. Harris et al. [36] deal with stream compaction, summed-area tables, and radix sort as the applications, and describe the details of them. In this section, we deal with stream compaction. It is commonly used in parallel algorithms.

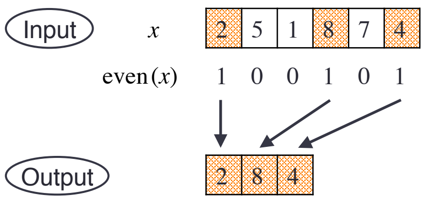


Figure 5.1: An example of stream compaction

5.3.1 Stream Compaction

Stream compaction is a kind of filtering operation.

Problem 5.1 (Stream Compaction) *Given an array of integer numbers and a filter function f that takes a real number and returns a boolean value, output the array that consists of all elements each element x of which satisfies $f(x) = 1$.*

Note that we assume that memory addresses in an array are continuously allocated in order of indices in this dissertation.

Figure 5.1 shows an example. The function *even* returns one for even numbers, and zero for odd numbers. The output is the array that consists of even numbers in the input array. Obviously, the sequential algorithm requires $\Theta(n)$ times to solve the problem. We want to reduce the time complexity using parallel algorithms.

The parallel algorithm consists of three steps. Figure 5.2 shows the process. The first step generates the flag array that stores $even(x)$ for each input element x. The second step calculates prefix scan of the flag array. The resulting array indicates the destination addresses for the input elements that satisfy $even(x) = 1$. The third step moves the input elements to the output array. This operation is called *scatter*.

Obviously, the step 1 and 3 can be calculated in parallel because there is no data dependency. Thus, the performance of the algorithm heavily depends on the step 2, that is, prefix scan. Actually, it is not easy to implement scatter operations efficiently on GPUs because the global memory accesses are not always coalesced. In Chapter 6, we will introduce a good way to avoid non-coalesced accesses.

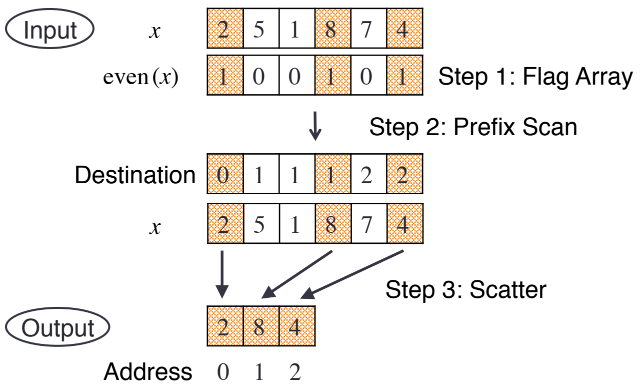


Figure 5.2: The process to calculate stream compaction

5.4 Prefix Scan Algorithms

In this section, we deal with GPU-based prefix scan algorithms. Some basic algorithms are proposed for parallel processors [37, 35]. We call them tree-based algorithms. The GPU-based implementations are provided by Harris et al. [36, 38]. The time complexity is $O(\log b)$ times larger than the trivial lower bound. Matrix-based algorithm is proposed for GPUs by Dotsenko et al. [10]. The algorithm has a tuning parameter α that makes tradeoff between the time complexity and the multiplicity. At $\alpha = 1$, the matrix-based algorithm is the same as the tree-based algorithms, while it is faster than the tree-based algorithms at the optimal value.

5.4.1 Matrix-based Algorithm

The matrix-based algorithm is designed for GPUs. We analyze the algorithm using the AGPU(p, b, M) model. Input data represented as a matrix with b columns are partitioned into matrices with α rows and b columns and each matrix is processed by a multiprocessor. We can choose an arbitrary value of the parameter α. Thrust [39], which is one of the standard CUDA libraries, uses a similar algorithm for prefix scan. We analyze the prefix scan algorithm in Thrust.

A pseudo-code for the prefix scan algorithm used in Thrust is shown in Algorithm 5.1. The code is slightly modified to make the algorithm suitable for the AGPU model. In addition, for simplicity, we omit the process of rounding of fractions. In the

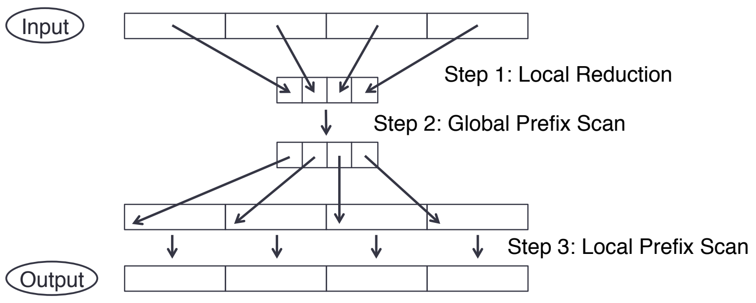


Figure 5.3: The outline of prefix scan algorithm

pseudo-code, parameter k represents the total number of multiprocessors, namely, $k = p/b$. Algorithms 5.2 and 5.3 are a subroutine invoked by Algorithm 5.1.

Figure 5.3 shows the outline of the algorithm. First, input data are partitioned into k blocks and each multiprocessor calculates the reduction of a block. This process is called "local reduction". The algorithm for the local reduction is shown in Algorithm 4.2. The result of reductions is stored in an array C. Then, one multiprocessor calculates the prefix scan of the array C. This process is called "global prefix scan". Finally, each multiprocessor calculates the prefix scan of a block. This process is called "local prefix scan". The value of index i in the global prefix scan array is equal to the final output value of the first element of i-th block. Therefore, each multiprocessor can concurrently calculates the prefix scan of a block using the result of the global prefix scan.

Algorithms 5.2 and 5.3 show the detail of the global and local prefix scan. Each multiprocessor calculates the prefix scan of a single block. At the local prefix scan, the number of blocks is k, whereas the number of blocks is one at the global prefix scan. In order to let the multiple multiprocessors calculate the prefix scan concurrently, we want to know the prefix scan value of the first element of each block. Actually, the output C of the global prefix scan represents the values.

In Algorithms 5.2 and 5.3, each block is divided into subblocks with αb elements. Each subblock is represented as a matrix with α rows and b columns ($\alpha \leq b$). Each multiprocessor repeats calculating the prefix scan of a matrix. We use column-major order, namely, first α elements in a block are stored in the first column. First, all elements in a matrix are transferred from the global memory to the shared memory. At this time, we rearrange the matrix in the shared memory as a matrix with α rows

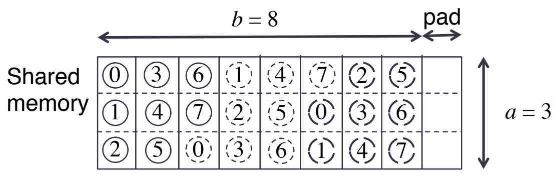


Figure 5.4: An example of the data alignment

and $b + 1$ columns such that the i-th column in the original matrix is stored in the i-th column and $(b + 1)$-th column in the new matrix does not store any elements. Figure 5.4 shows an example for $\alpha = 3$ and $b = 8$. The first α elements are stored in distinct banks because the number of columns is $b+1$. However, if $\alpha < b$, the $(\alpha+1)$-th element is stored in the same bank as the first element. Thus, bank conflicts occur. On the other hand, if α is equal to b, bank conflicts do not occur because the $\alpha = b$ contiguous elements are stored in distinct banks. Next, each core in a multiprocessor calculates the reduction of the α elements in one column in parallel. After that, we calculate the prefix scan of this b resulting values. Finally, each core calculates the prefix scan of one column. Since the prefix scan values of the first element of the columns is represented as the above b resulting values, each core can do it in parallel.

We analyze the complexities of the algorithm. First, we analyze the time complexity. It takes $O(n/p + \log b)$ time to calculate the local reduction because each core calculates n/p rows and it takes $O(\log b)$ time to merge the result of each core. Note that bank conflict does not occur because b cores in a multiprocessor always access b contiguous elements in the shared memory. Next, we analyze the time complexity of global and local prefix scan. Bank conflicts may occur when elements in a matrix are transferred from global memory to shared memory. Each multiprocessor repeats copying b elements α times. Although all elements in the same column are in distinct banks due to padding, elements in different columns can be in the same bank. The degree of conflicts is $\min\{\lceil b/\alpha\rceil, \alpha\}$ because the degree is equal to or smaller than the number of columns used by b elements and it is also equal to or smaller than α. Since a multiprocessor reads b elements α times, time complexity of reading a matrix is $O(\min\{\lceil b/\alpha\rceil, \alpha\} \cdot \alpha)$. Writing the resulting values of a matrix to global memory has the same time complexity. In addition, it takes $O(\alpha + \log b)$ time to calculate the prefix scan of one matrix in the shared memory. Therefore, the time complexity to calculate the prefix scan of a matrix

Table 5.1: Complexities and multiplicity of the prefix scan algorithm adopted by Thrust on AGPU(p, b, M). Here n is the number of input elements. We assume n is much larger than the number of cores p.

Algorithms	Time complexity	I/O complexity	Multiplicity
Matrix-based	$O\left(\left(\frac{n}{p}+\frac{p}{b^2}\right)\left(\min\left\{\left\lceil\frac{b}{\alpha}\right\rceil,\alpha\right\}+\frac{\log b}{\alpha}\right)\right)$	$\frac{3n}{b}+O\left(\frac{p}{b}\right)$	$O\left(\frac{M}{\alpha b}\right)$

is $O(\min\{\lceil b/\alpha\rceil, \alpha\} \cdot \alpha + \log b)$ in total. To calculate the global prefix scan, this process is serially repeated $k/(\alpha b)$ times. To calculate the local prefix scan, this process is serially repeated $n/(k\alpha b)$ times. Therefore, the total time complexity is

$$O\left(\left(\frac{n}{p}+\frac{p}{b^2}\right)\left(\min\left\{\left\lceil\frac{b}{\alpha}\right\rceil,\alpha\right\}+\frac{\log b}{\alpha}\right)\right).$$

Next, we analyze the I/O complexity. The algorithm uses $(n/b + k)$ I/Os to calculate the local reduction, $(2p/b^2)$ I/Os to calculate the global prefix scan, and $(n/b + k)$ I/Os to calculate the local prefix scan. Therefore, the total number of I/Os is $3n/b + O(p/b)$. Next, the amount of the shared memory used is $O(\alpha b)$ words because the algorithm uses αb words for a matrix to calculate the global and local prefix scan. The amount of the global memory used is $(2n+p/b)$ words because $2n$ words are used for input/output and p/b words are used to store the result of the local reduction. Finally, the multiplicity is $O(M/(\alpha b))$.

Table 5.1 shows the I/O and time complexities and multiplicity of the matrix-based algorithm.

Parameter Tuning

We can choose an arbitrary value of the parameter α. First, we discuss how the value of parameter α affects the performance of the algorithm. The parameter α is related to the time complexity. When α is equal to one, the time complexity is $O((n/p + p/b^2)(\log b))$. When $\alpha = \Omega(b)$, the time complexity is $O(n/p + p/b^2)$. Thus, the time complexity depends on α, and it attains the minimum when $\alpha = \Omega(b)$. On the other hand, when α is large, the multiplicity is inversely proportional to α. When α is equal to one, the multiplicity is considered to be optimal since the amount of shared memory used is $O(b)$. On the other hand, When $\alpha = \Omega(b)$, the multiplicity is not considered to be

Algorithm 5.1 Calculate prefix scan

```
1: procedure CalculatePrefixScan($T$,$U$,$n$)
                                        ▷ Given an array $T[n]$, Output $U[n]$
2:     $C[0..p/b-1]$ = CalculateLocalReductionUsingCascading($T, n$)
                                        ▷ Local reduction

3:     for all $\rho \in$ MP[0..0] in parallel do            ▷ Global prefix scan
4:         for all $\epsilon \in$ Core[$0..b-1$] in parallel do
5:             CalculateBlockPrefixScan($C, C, k$, NULL)
6:         end for
7:     end for

8:     for all $\rho \in$ MP[$0..k-1$] in parallel do        ▷ Local prefix scan
9:         for all $\epsilon \in$ Core[$0..b-1$] in parallel do
10:            CalculateBlockPrefixScan($\&T[n*\rho/k], \&U[n*\rho/k], n/k, \&C[\rho]$)
11:        end for
12:    end for

13: end procedure
```

optimal since the amount of shared memory used is $O(b^2)$. To summarize, parameter α must be chosen from between 1 and $\Theta(b)$ with careful consideration of two competing factors: the time complexity and the multiplicity.

If the input size n is much larger than p and $\alpha = \Omega(b)$, the time and the I/O complexities become $O(n/p)$ and $O(n/b)$, respectively, which are asymptotically optimal. However, if the input size n is not sufficiently larger than p, the term p/b^2 affects the time complexity. This is due to the inefficiency of the global prefix scan. The time complexity can be improved by using multiple multiprocessors for the process. It can be reduced to

$$O\left(\frac{n}{p}\left(\min\left(\left\lceil\frac{b}{\alpha}\right\rceil, \alpha\right) + \frac{\log b}{\alpha}\right) + \log p\right)$$

by using the algorithm proposed by Harris et al. [36].

Algorithm 5.2 Calculate block prefix scan

```
1: procedure CalculateBlockPrefixScan($T$,$U$,$n$,$C$)
                                                  ▷ Given an array $T[n]$, Output $U[n]$
2:     Allocate a matrix $m[\alpha][b+1]$ in the shared memory.
3:     for $i \leftarrow 0$ to $n-1$ do
                  ▷ Sequentially calculate the prefix scan of the matrix with $\alpha b$ elements.
4:         for $j \leftarrow 0$ to $\alpha - 1$ do
                  ▷ Read all elements in a matrix and arrange them in column-major order.
5:             $data\_tmp[\epsilon] \Leftarrow T[\alpha bi + jb + \epsilon]$
6:             $m[(jb+\epsilon)\%\alpha][(jb+\epsilon)/\alpha] \leftarrow data\_tmp[\epsilon]$
7:         end for
8:         $carry \Leftarrow C$          ▷ Each core calculates the reduction of one column.
9:         if $((\epsilon = 0)\&\&(C \neq NULL))$ then
10:            $val\_column[\epsilon] \leftarrow C$
11:        else
12:            $val\_column[\epsilon] \leftarrow I_{\oplus}$
13:        end if
14:        for $j \leftarrow 0$ to $\alpha - 1$ do
15:            $val\_column[\epsilon] \leftarrow val\_column[\epsilon] \oplus m[j][\epsilon]$
16:        end for

17:        for $(j = 1;\ j < b;\ j = j * 2)$ do
                              ▷ Cores calculate the prefix scan of $b$ resulting values.
18:            if $(\epsilon \geq j)$ then
19:                $val\_column[\epsilon] \leftarrow val\_column[\epsilon - j] + val\_column[\epsilon]$
20:            end if
21:        end for

22:        if $((\epsilon = 0)\&\&(C \neq NULL))$ then
                              ▷ Each core calculates the prefix scan of one column.
23:            $next \leftarrow C \oplus m[0][\epsilon]$
24:            $m[0][\epsilon] \leftarrow C$
25:        else
26:            $next \leftarrow m[0][\epsilon]$
27:            $m[0][\epsilon] \leftarrow val\_column[p-1]$
28:        end if
```

Algorithm 5.3 Calculate block prefix scan (Continued)

```
29:         for $j \leftarrow 1$ to $\alpha - 1$ do
30:             $tmp \leftarrow m[j][\epsilon]$
31:             $m[j][\epsilon] \leftarrow next$
32:             $next \leftarrow next \oplus tmp$
33:         end for

34:         for $j \leftarrow 0$ to $a - 1$ do
              ▷ Each multiprocessor writes all words in a sub-block to global memory.
35:             $data\_tmp[\epsilon] \leftarrow m[(jb + \epsilon)/a]$
36:             $U[abi + jb + \epsilon] \Leftarrow data\_tmp[\epsilon]$
37:         end for
38:     end for
39: end procedure
```

5.5 Experimental Evaluation

We measured the actual running time of the matrix-based algorithm for various values of α using NVIDIA Tesla C1060 and k20c. We have implemented the program using CUDA and the algorithm is based on the prefix scan algorithm in the Thrust library. The operator is 32-bit integer addition and the size of the input is $2^{27} = 134{,}217{,}728$. The shared memory size per multiprocessor is 16 Kbyte in the C1060, and it is 48Kbyte in the k20c. The maximum number of warps assigned to a multiprocessor is limited to 16 in the C1060 and it is limited to 64 in the k20c.

Figure 5.5 shows the actual running time for various values of α. The horizontal axis represents the value of α, and the vertical axis represents the bandwidth, which is a throughput speed. We compute the bandwidth as $n \times 2 \times sizeof(int)/t$ where t is running time of the algorithm. This value is limited by the bandwidth of the architectures. The bandwidth of the C1060 is 102 GB/s, and the bandwidth of the k20c is 208 GB/s.

The multiplicity has the maximum value when $\alpha < 6$ in the k20c. In this range, the bandwidth decreased with decreasing α. It is considered due to the large time complexity. When α is equal to or larger than 6, the value is affected by two competing factors, the time complexity and the multiplicity. The largest bandwidth was attained at $\alpha = 18$. When $\alpha > 18$, it appears that the small multiplicity strongly affects the value.

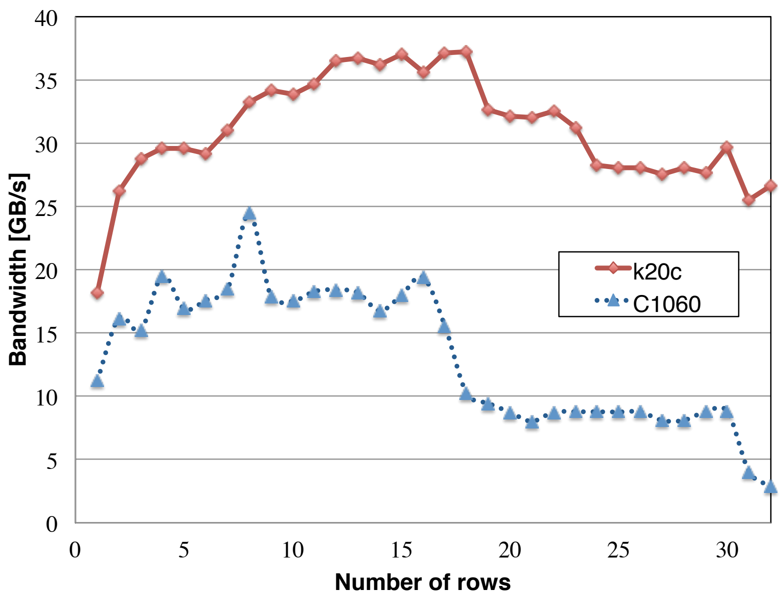


Figure 5.5: Bandwidth of the prefix scan algorithm with varying number of row in the matrix

The line of the C1060 shows the same tendency. The multiplicity has the maximum value at $\alpha < 8$.

5.6 Short Summary

In this chapter, we analyzed the time and I/O complexities of the prefix scan algorithm and measured the performance on real GPUs. The state-of-the-art algorithm has a tuning parameter α. We showed that the parameter makes tradeoff between the time complexity and the multiplicity. Then, we measured the actual running time of the algorithm with various parameter values on the real GPUs and obtained the expected performance. We can conclude that the AGPU model can explain the behavior of the algorithm.

6 Sorting Algorithms

6.1 Introduction

In this chapter, we deal with sorting algorithms. Since sorting is one of the most fundamental operations used for many applications, it is useful to speed it up. Sorting algorithms are grouped into two types, non-comparison sorting and comparison sorting. With respect to non-comparison sorting, many radix sort algorithms [36, 40, 41, 42, 43] are proposed for GPUs, and one of the fastest radix sort algorithms is proposed by Merrill and Grimshaw [44, 45]. Kolonias et al. [46] proposed a GPU-based count sort.

Many algorithms for comparison sorting are also proposed for GPUs. An early implementation of bitonic sort [47] is proposed by Purcell et al. [48, 49], and some algorithms are based on the bitonic sort, including GPUTeraSort [50], GPU-ABiSort [51], IBR bitonic sort [52]. A GPU-based bitonic sort is explained in Section 6.2.1. Kipfer and Westermann [53] showed some basic algorithms based on an odd-even transition sorting network. Satish et al. [42] proposed an algorithm based on merge sort. He et al. [54] and Cederman et al. [55] provided GPU-based quick sort algorithms. Leischner

Table 6.1: Complexities of comparison-based sorting algorithms on the AGPU model. Here n is the number of elements to be sorted, p is the number of total cores, b is the number of cores in a multiprocessor, M is the size of the shared memory in a multiprocessor. We assume $n = \Omega(b^2)$.

Algorithms	I/O complexity	Time complexity
(Lower bound)	$\Omega\left(\frac{n}{b}\log_{\frac{M}{b}}\frac{n}{b}\right)$	$\Omega\left(\frac{n}{p}\log n\right)$
Bitonic sort [47]	$O\left(\frac{n}{b}\log^2\frac{n}{M}\right)$	$O\left(\frac{n}{p}\log^2 n\right)$
GPU-Warpsort [11]	$O\left(\frac{n}{b}\log\frac{n}{b}\right)$	$O\left(\frac{n}{p}\log\frac{n}{b}\log b\right)$
Our algorithm	$O\left(\frac{n}{b}\log_{\frac{M}{b}}\frac{n}{b}\right)$	$O\left(\frac{n}{p}\log\frac{n}{b}\log b\right)$

et al. [56] proposed sample sort algorithms, which is a multi-separator variation of the quick sort.

GPU-Warpsort [11] utilizes both bitonic sort and efficient merge sort. This is one of the fastest algorithms for comparison sorting, but its I/O complexity is not optimal. We therefore propose a new algorithm with the optimal I/O complexity. Table 6.1 shows the I/O and time complexities of these algorithms.

We first discuss some known parallel sorting algorithms that can be executed on the AGPU model in Section 6.2. Then, we discuss the lower bound on the time and I/O complexities for comparison-based sorting algorithms on the AGPU model in Section 6.3. After that, we explain the detail of our sorting algorithms in Section 6.4. Finally, we evaluate our algorithms on real GPUs and show that our algorithm is faster than the existing algorithm in practice in Section 6.5.

6.2 Analyses of Known Parallel Sorting Algorithms

6.2.1 Bitonic Sort

Bitonic sort [47] is a sorting algorithm based on *bitonic sorting networks*, and it can sort n elements in parallel using multiple comparators.

Figure 6.1 shows an example of bitonic sorting networks. The bitonic sorting network for $n = 2^d$ input elements consists of $d = \log n$ phases, and phase i $(0 \le i < d)$ consists of $i + 1$ stages. Given 2^{d-i} sorted sequences of length 2^i each at Phase i, 2^{d-i-1}

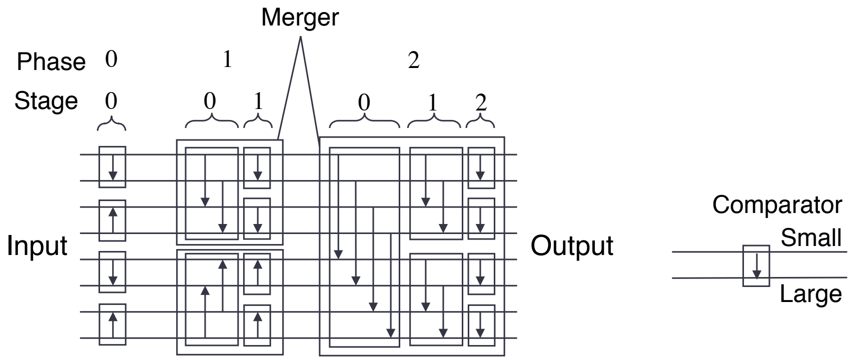


Figure 6.1: The bitonic sorting network for 8 elements

mergers output 2^{d-i-1} sorted sequences of length 2^{i+1} each. This algorithm is suitable for GPU-based implementations because it requires no conditional branches.

We can execute the bitonic sort algorithm on the p processor PRAM model. The PRAM model can simulate p comparators in parallel, while the sorting network executes $n/2$ comparators in parallel at each stage. Since the number of stages of the bitonic sort is $O(d^2) = O(\log^2 n)$, the running time of a bitonic sort algorithm on the PRAM model is $O((n \log^2 n)/p)$.

We next execute the bitonic sort algorithm on the AGPU(p, b, M) model. As is the case with the PRAM model, the AGPU model can simulate p comparators in parallel. Additionally, we take multiprocessors into account. The input and output elements are stored in the global memory. In order to simulate a comparator, a multiprocessor reads the elements from the global memory to the shared memory, compares them, and then outputs the results to the global memory. We modify the method by simulating one merger using one multiprocessor. If the input size of the merger is equal to or smaller than M, a multiprocessor can simulate a merger without the global memory, which decreases the I/O complexity. Otherwise, the multiprocessor has to use the global memory as a temporary buffer. Furthermore, if the input size of the merger is equal to or smaller than M, a multiprocessor can simulate multiple phases without the global memory, which also decreases the I/O complexity.

We analyze the I/O complexity of the algorithm. The output of phase i $(0 \leq i < d)$ is 2^{d-i-1} sorted sequences of length 2^{i+1} each. Because the input size of a merger is M at the phase $\log M - 1$, computations in phases 0 to $\log M - 1$ are done without communication between multiprocessors. Therefore the I/O complexity in the phases is $O(n/b)$ in total. With respect to phases $\log M$ to $\log n$, the algorithm does not require any I/Os in stages $\log n - \log M + 1$ to $\log n$. Though it is necessary to read and write all elements in stages 0 to $\log n - \log M$, those I/Os are done to consecutive addresses and all global memory accesses are done in units of b elements. From these analyses, the I/O complexity of the bitonic sort is $O(n(\log n - \log M)^2/b)$.

We next analyze the time complexity of the algorithm. Since this algorithm requires no conditional branch, and multiprocessors can always access b contiguous elements in the shared memory, each stage can be simulated in $O(n/p)$. Thus, the time complexity is $O((n \log^2 n)/p)$. We therefore obtain the following:

Theorem 6.1 *The bitonic sort algorithm for n elements on the AGPU(p, b, M) model has I/O complexity $O((n\log^2(n/M))/b)$ and time complexity $O((n\log^2 n)/p)$.*

6.2.2 GPU-Warpsort

Since the bitonic sort for n elements consists of $O(\log^2 n)$ stages, it is inefficient if n is large. In order to get rid of the inefficiency of the bitonic sort, GPU-Warpsort [11] combines the bitonic sort and the merge sort. The algorithm first divides the input into sequences with length b, and sorts all sequences. Then it repeatedly merges two sorted sequences similar to merge sort algorithm.

We explain how to merge two sorted sequences A, B using a multiprocessor with b cores. Suppose the length of each sequence is $n/2$ and it is equal to or larger than b. It is illustrated in Figure 6.2. The procedure first copies the first b elements of both sequences from the global memory to the shared memory. Let $a_{\max}$ and $b_{\max}$ denote the maximum values read from the sequences A and B, respectively. The multiprocessor sorts those $2b$ elements using the bitonic sort. It is enough to execute only the last phase of the bitonic sort because those elements are composed of two sorted sequences. The time complexity is $O(\log b)$. After sorting the $2b$ elements, the procedure outputs the smallest b elements, and reads b new elements from either sequence A or B. If $a_{\max} \leq b_{\max}$, the b new elements are copied from sequence A, and otherwise from sequence B. The procedure repeats this procedure until all elements of the sequences A and B are processed.

In order to prove the correctness of the procedure, we first define the *rank* of the

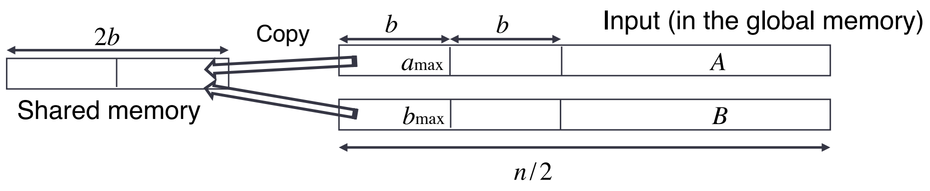


Figure 6.2: Explanatory diagram showing how to merge two sorted sequences A, B

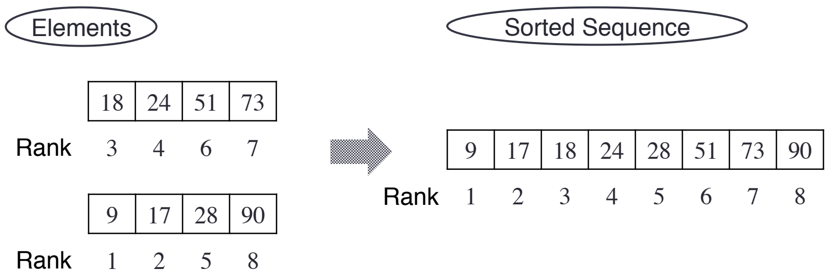


Figure 6.3: An example of the ranks for 8 distinct elements

elements. Given n elements, the rank of element e is the number of elements smaller than or equal to the element e in the n elements. If the elements are distinct, the rank of element e is equal to the position of the element e in the sorted sequence of n elements. Figure 6.3 shows an example for 8 distinct elements.

Now we prove the correctness of the procedure, which is not mentioned in the original paper [11]. The procedure outputs b elements per iteration. Since the procedure obviously outputs the smallest b elements at the first iteration, we consider the subsequent iterations. If the procedure outputs all elements whose rank is equal to or smaller than ib by the i-th iteration ($2 \leq i \leq n/b$), the output is correct. Note that the number of such elements is at most ib. Suppose the output of the procedure is correct until the $(i-1)$-th iteration, and $a_{\max}$ is equal to or smaller than $b_{\max}$ at the beginning of the i-th iteration. The proof at the case where $a_{\max}$ is larger than $b_{\max}$ is analogous. Figure 6.4 shows the situation at the beginning of the i-th iteration. All elements in the gray-colored region are already copied to the shared memory. If $a_{\max} \leq b_{\max}$, the b new elements are copied from sequence A. In Figure 6.4, the elements are in the region (1). If all elements whose rank is equal to or smaller than ib are copied to the shared memory, the procedure outputs the correct elements using the

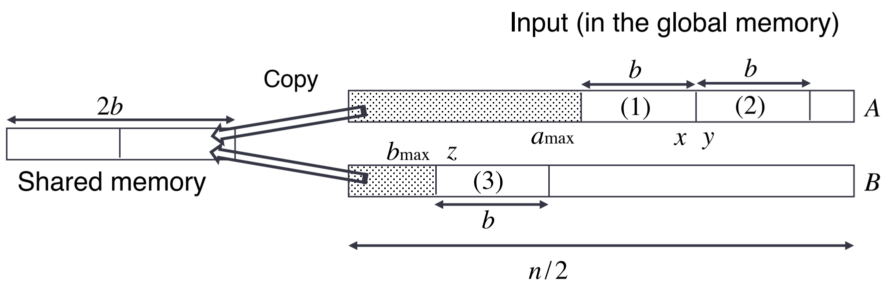


Figure 6.4: The proof of the correctness of the process that merges two sorted sequences. The gray-colored regions represent elements in the regions are already copied to the shared memory.

bitonic sort. We therefore prove all elements that are not copied to the shared memory have the ranks larger than ib. It is enough to prove the first element y in the region (2) and the first element z in the region (3) in Figure 6.4 have the ranks larger than ib. Let x be the last element in the region (1).

By the $(i-1)$-th iteration, the procedure outputs $(i-1)b$ elements that include all elements whose rank is equal to or smaller than $(i-1)b$. If x and y are distinct, the rank of y is at least one plus the rank of x. Since the rank of x is at least $(i-1)b + b = ib$, the rank of y is at least $ib + 1$. If x and y have the same value, the region (1) and (2) have at least $b + 1$ elements such that the value of the element is equal to or smaller than y. Therefore, the rank of x is at least $(i-1)b + (b+1) = ib + 1$. It indicates that the rank of y is at least $ib + 1$. Similarly, if $b_{\max}$ and z are distinct, the rank of z is at least one plus the rank of $b_{\max}$. The rank of $b_{\max}$ is at least ib because $b_{\max}$ is the largest value in the elements copied to the shared memory by the beginning of the i-th iteration and the number of such copied elements is ib. The rank of z is therefore at least $ib + 1$. If $b_{\max}$ and z have the same value, the gray-colored region and the region (3) have at least $b + 1$ elements such that the value of the element is equal to or smaller than $b_{\max}$. Therefore, the rank of $b_{\max}$ is at least $(i-1)b + (b+1) = ib + 1$. It indicates that the rank of z is at least $ib + 1$. Thus, the ranks of y and z are strictly larger than ib. Accordingly, the procedure do not need these elements at the i-th iteration.

The procedure always reads b elements from the global memory using coalesced memory access. Since the procedure basically reads b element and outputs b elements at each iteration, we obtain the following.

Lemma 6.2 *On the AGPU(p, b, M) model, merging of two sorted sequences of length $n/2$ each is done with I/O complexity $2n/b + O(1)$ and time complexity $O((n \log b)/b)$. This algorithm uses one multiprocessor and $O(b)$ words of shared memory.*

When the number of sorted sequences becomes smaller than the number of multiprocessors, we cannot utilize all multiprocessors. The GPU-Warpsort therefore adds some phases. It consists of four phases.

1. Given an input sequence of length n, compute n/b sorted sequences of length b each.
2. Merge two sorted sequences into one, and repeat merging until the number of sorted sequences is less than the number of multiprocessors.
3. Pick up some elements from the sorted sequences, and divide the sequences into some groups using the picked elements as separators.
4. Merge the sequences in the same group.

Phases 3 and 4 prevent some multiprocessors from being idle.

The I/O and time complexities of the GPU-Warpsort are analyzed as follows. Phase 1 has I/O complexity $2n/b + O(1)$ and time complexity $O((n \log^2 b)/p)$. Phase 2 has I/O complexity $O((n \log(n/b))/b)$ and time complexity $O((n \log b \log(n/b))/p)$. Although the original paper [11] does not mention it, we can implement the algorithm such that the time and I/O complexities of Phases 3 and 4 are dominated by those of Phases 1 and 2. We will explain the detail in Section 6.4. Supposing $n = \Omega(b^2)$, the time complexity of Phase 1 is dominated by that of Phase 2. Thus, we obtain the following theorem.

Theorem 6.3 *Supposing $n = \Omega(b^2)$, comparison sorting for n elements using GPU-Warpsort runs on the AGPU(p, b, M) model with I/O complexity $O((n \log(n/b))/b)$ and time complexity $O((n \log(n/b) \log b)/p)$.*

Because a trivial lower bound of the time complexity for comparison-based sorting is $\Omega((n \log n)/p)$, the time complexity of the GPU-Warpsort is less than $O(\log b)$ times that of the optimal algorithm. On the other hand, the I/O complexity is $O(\log(M/b))$ times larger than the optimal, as shown in Section 6.3.

6.3 Sorting Lower Bound on the AGPU Model

We discuss the lower bound on the time and I/O complexities for comparison-based sorting algorithms. The lower bound on the time complexity for sequential algorithms is $\Omega\,(n \log n)$. Since a device in AGPU(p, b, M) has p cores, the trivial lower bound on time complexity for AGPU(p, b, M) is $\Omega((n \log n)/p)$.

With respect to the I/O complexity, the lower bound for the I/O model [12] is known as follows:

Theorem 6.4 (Aggarwal, Vitter [12]) *A lower bound of I/O complexities of comparison-based sorting algorithms for n elements on the I/O(b, M) model is $\Omega(\frac{n}{b} \log_{\frac{M}{b}} \frac{n}{b})$.*

Since Theorem 3.5 holds, the lower bound on the I/O complexity for the volatile model AGPU(p, b, M) is as follows:

Theorem 6.5 *Any comparison-based algorithm for sorting n elements on the volatile AGPU(p, b, M) model requires $\Omega(\frac{n}{b} \log_{\frac{M}{b}} \frac{n}{b})$ I/Os.*

Even though Theorem 3.5 does not hold on non-volatile model AGPU′, we can obtain the same lower bound as follows.

Theorem 6.6 *Any comparison-based algorithm for sorting n elements on the non-volatile AGPU′(p, b, M) model requires $\Omega(\frac{n}{b} \log_{\frac{M}{b}} \frac{n}{b})$ I/Os.*

Proof. A trivial lower bound of I/O complexities for sorting n elements is n/b, which is necessary to read all the elements from the global memory. Therefore the I/O complexity of any sorting algorithm does not change asymptotically if a preprocess using $O(n/b)$ I/Os is added. Therefore we preprocess the input to a sorting algorithm so that the n elements are divided into blocks of consecutive b elements, and elements in each block are sorted using the shared memory of a multiprocessor. This preprocess is done with $O(n/b)$ I/Os. From now on, we assume that the input to a sorting algorithm is n/b sorted sequences of length b each. The number of possible inputs is $n!/(b!)^{n/b}$. A global memory access will transfer b elements into the shared memory of a multiprocessor. Because a multiprocessor can store M elements, a multiprocessor can compare the b elements that are newly copied into it with at most $M - b$ elements that already exist in it. After some computation in a multiprocessor, it will output data in the shared memory to the global memory. There are at most $\binom{M-b}{b} < M^b/b!$ different results of comparison after 2 accesses (read and write) to the global memory. Therefore the number of necessary global memory accesses to process $n!/(b!)^{n/b}$ different inputs is

$$\log_{M^b/b!} \frac{n!}{(b!)^{n/b}} = \Omega\left(\frac{n}{b} \log_{M/b} \frac{n}{b}\right).$$

□

6.4 I/O-optimal Sorting Algorithms

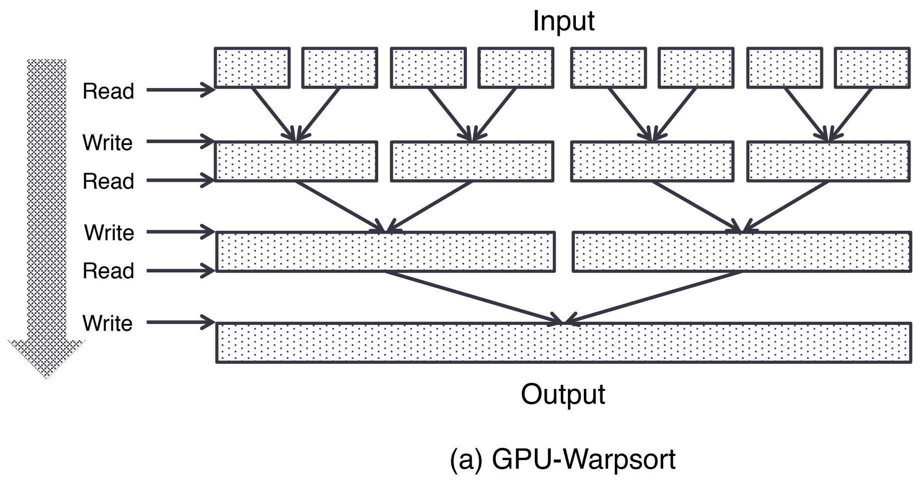
In this section we propose a comparison sorting algorithm on the AGPU(p, b, M) model whose I/O complexity is asymptotically optimal. Aggarwal and Vitter [12] propose an I/O-optimal algorithm on the I/O model, but do not take GPU architectures into account. We improve the I/O complexity of the GPU-Warpsort using the technique of Aggarwal's algorithm.

We extend the GPU-Warpsort so that each multiprocessor merges more than two sorted sequences at a time. In the GPU-Warpsort, every time multiprocessors merge two sorted sequences, the elements are read from and written to the global memory. By merging more than two sorted sequences at a time, we can reduce the number of global memory accesses. Figure 6.5 shows an example of merging eight sequences at a time. In this example, our algorithm merges eight sorted sequences at a time. Therefore, our algorithm accesses each element in the global memory twice, while the GPU-Warpsort accesses each element six times.

6.4.1 Overview of the Algorithm

Our algorithm consists of the following four parts:

1. Initialization,
2. Column-wise merge,
3. Subarray partition, and
4. Row-wise merge.



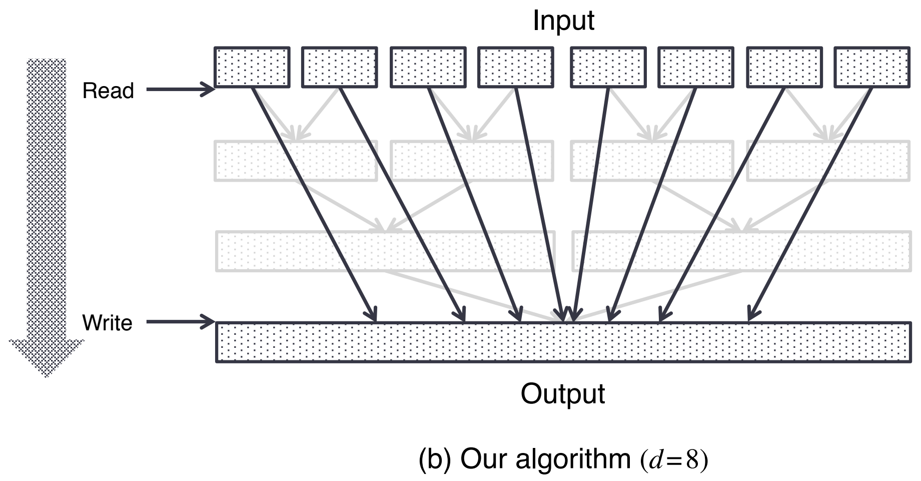


Figure 6.5: Global memory accesses to merge eight sorted sequences; (a) GPU-Warpsort, (b) our algorithm. In this example, our algorithm merges eight sequences at a time. In the GPU-Warpsort, the number of the global memory accesses for each element is six, whereas the number is two in our algorithm.

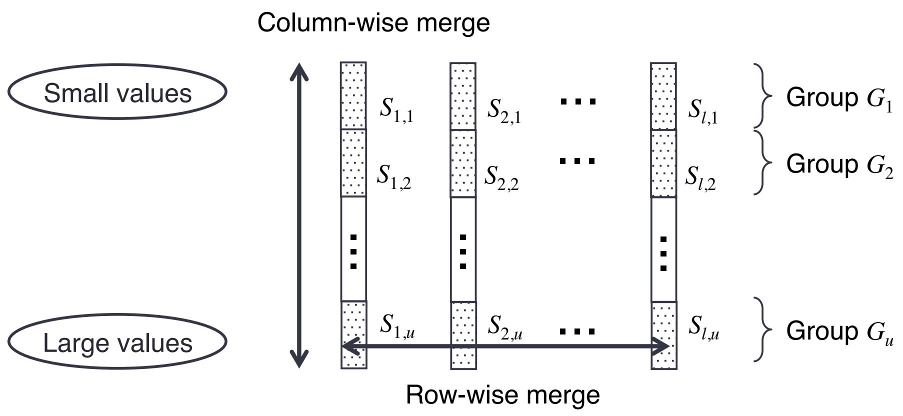


Figure 6.6: Column-wise and Row-wise merge

In Part 1, our algorithm divides the input sequence into pieces of b elements. We call each piece a *basic block*. Then we sort each basic block. We name sorted sequences *subarrays*. In Part 2, we repeatedly merge subarrays until the number of subarrays is equal to or smaller than a threshold ℓ, whose value will be determined later. In Part 3, we pick up some separators from each subarrays at regular intervals and merge all separators. We partition each subarray into groups based on the separators $p_1, p_2, \ldots, p_u$. All elements in group G_j are basically larger than or equal to separator p_{j-1} and smaller than separator p_j. Finally in Part 4, we merge subarrays that belong to the same group. This prevents some multiprocessors from being idle. Figure 6.6 illustrates the column-wise merge and the row-wise merge.

6.4.2 Initialization

In Part 1, our algorithm divides the input sequence into basic blocks of b elements and sorts them. Each basic block is sorted by a multiprocessor using the Bitonic sort. Since the number of the basic blocks is n/b and a multiprocessor can read or write all elements in a basic block by one global memory access, the I/O complexity is $2n/b + O(1)$. Since $k = p/b$ basic blocks are sorted in parallel, the time complexity is $O((n \log^2 b)/(bk)) = O((n \log^2 b)/p)$.

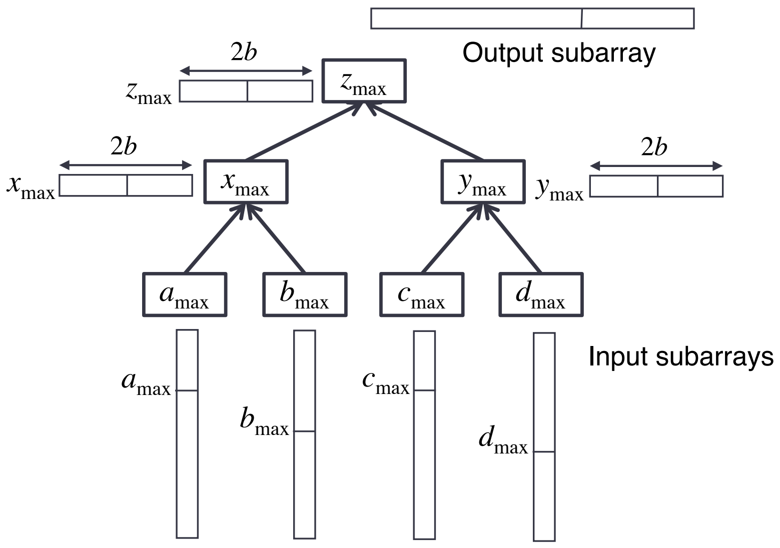


Figure 6.7: A heap used in the merge process

6.4.3 Column-wise Merge

In this part, we repeatedly merge d subarrays into one subarray using small amount of shared memory in a multiprocessor. The d input subarrays and the output subarray are stored in the global memory. First we explain the data structure used to merge the subarrays. It is a kind of heap structures; it is a rooted binary tree and each node has at most two child nodes. For simplicity, we assume every internal node has always two child nodes. In other cases, we can easily modify our algorithm. This structure has d leaf nodes. Due to the above assumption, d is power-of-two. Figure 6.7 shows an example for the case $d = 4$.

Each leaf node stores a pointer to an input subarray, while each internal node has a buffer in a shared memory that can store $2b$ elements. Because the number of internal nodes is $d - 1$, the amount of shared memory used is $2b(d - 1)$ words. Each buffer is sorted whenever new elements are inserted. Input elements are read into a leaf, and then transferred to its parent node. Each internal node moves elements inside its buffer to the parent node according to a rule that will be described later. The elements in the root node will be output to the global memory. Each internal node and leaf has a key, which is the last value moved to its parent node (in the case of the root node, last output value). Because elements in buffers and subarrays are sorted, the last moved value is the maximum value moved so far.

Next we explain the "Heapify" operation. Each node of the heap has an index. The root node has index 1. The left and the right children of node v have index $2v$ and $2v + 1$, respectively. The function Heapify(v) is the process to move b elements to the buffer in node v from the buffers in the descendent nodes. The function is only invoked when the number of elements in the buffer of node v is at most b. For simplicity, we assume the number of elements in the buffer is exactly b. We can easily modify our algorithm for the case where the number of elements is smaller than b. First, b smallest elements are moved to the buffer from the child node that has smaller key, and the key of the child is updated to the value of the last element of the b elements. Then the buffer of node v consists of two sorted sequences; one is already stored in node v before the move, and the other is newly moved to node v. We merge these two sequences into one using the Bitonic sort. Then we repeat carrying out the same process to the child node with the smaller key until we reach a leaf.

We can merge d sorted sequences into one by repeating the Heapify operation. First, we allocate the shared memory to this structure and construct the heap by repeating the Heapify operation on nodes in decreasing order of node indices. At the time we set the key of each node as $-\infty$. After that, we repeatedly output the smallest b elements in the buffer of the root node by the Heapify operation to the root node until all elements have been output.

In order to prove that the heap outputs a correct sorted sequence, we prove that the buffer of each node always stores the smallest b elements in the subtree that consists of its own node and the descendent nodes. Given a subtree, the rank of element e is defined as the number of elements smaller than or equal to the element e in the subtree. For instance, the rank of the smallest element in the subtree is one if the elements are distinct. For any node v, we now prove the ranks of any elements in the descendent nodes are larger than ib after the i-th output process. First we consider an internal node whose children are leaves. We assume this is the i-th output process for node v and the output is correct until the end of $(i - 1)$-th output process. Let $\alpha_{\max}$ and $\beta_{\max}$ be the keys of the left and the right children of node v for which the Heapify operation is done. We assume that $\alpha_{\max} \leq \beta_{\max}$; the other case is done analogously. When we conduct the Heapify operation to node v, the buffer of node v has b elements, and the smallest b elements in the left child are newly moved to the buffer of node v. Due to the discussion in Section 6.2.2, any elements in the left child buffer have ranks larger than ib after the operation. Similarly, any elements in the right child buffer have ranks larger than ib after the operation. We can recursively prove this property for any internal nodes.

Note that we can reduce the size of each buffer from $2b$ to b as follows. The above algorithm repeatedly carries out the Heapify operation from the root to a leaf. However, we can improve this by recursively carrying out the Heapify of its child node before the Heapify of its own node, which makes it unnecessary to keep more than b elements in each buffer.

To sum it up, we obtain the following.

Lemma 6.7 *Merging d sorted sequences of length n/d each is done with I/O complexity $2n/b + O(d)$ and time complexity $O(((n/b)\log d + d)\log b)$ on the AGPU$(b, b, O(db))$ model.*

Accordingly, if a multiprocessor has M word shared memory, our algorithm can concurrently merge up to $d = O(M/b)$ sorted sequences. We carry out this process to all subarrays using k multiprocessors in parallel. In the column-wise merge part, we repeat this step s_0 times, where s_0 is a parameter determined later.

6.4.4 Subarray Partition

In Part 3, we divide each subarray into u subarrays. Let ℓ be the number of subarrays remaining after Part 2. First, we pick up ρ elements from each subarray at regular intervals, that is, we pick up one element per $n/\rho\ell$ elements. We obtain ℓ lists of ρ sorted elements. We call them *separators*. Then, we merge the lists into one using the algorithm of Lemma 6.7. Let $p_1 \leq p_2 \leq \ldots \leq p_{\rho\ell}$ denote the resulting separators, and let $p_0 = -\infty$ and $p_{\rho\ell+1} = \infty$. After that, we divide each subarray into $u = \rho\ell + 1$ subarrays using the separators. Supposing subarray S is divided into subarrays $S_1, S_2, \ldots, S_u$ using separators $p_0, p_1, p_2, \ldots, p_u$, any elements in the resulting subarray S_j are equal to or larger than the value of separator p_{j-1} and smaller than the value of separator p_j for any j $(1 \leq j \leq u)$.

Let group G_j $(1 \leq j \leq u)$ be a set of subarrays between p_{j-1} and p_j. Each group has ℓ subarrays. The size of a group represents the number of the elements in the group. Let $\left|G_j\right|$ denote the size.

We can use the algorithm of Lemma 6.2 to divide each subarray. We first merge a subarray and the separators to one sorted sequence, then, calculate the position of the separators in the resulting sequence.

6.4.5 Row-wise Merge

We assign the groups to k multiprocessors using the following algorithm. Each multiprocessor is serially assigned its groups. The first multiprocessor is repeatedly assigned a group while the total size of assigned groups is smaller than $2n/k$. When the total size of the assigned groups is equal to or larger than $2n/k$ or there are no groups to assign, we finish assigning groups to the multiprocessor. Then, the next multiprocessor is repeatedly assigned a group in the same manner. We repeat this to all multiprocessors.

Lemma 6.8 *If $u > \ell k + 1$, we can assign all groups to the k multiprocessors such that no multiprocessors are assigned more than $2n/k$ elements.*

Proof. Using the above algorithm, we can ensure that the total size of assigned groups is smaller than $2n/k$ for any multiprocessors. We prove we can assign all groups to the multiprocessors using the above algorithm. Assume for contradiction that there exist groups that are not assigned to any multiprocessors at the end of the algorithm.

Since we pick up $(u-1)/\ell$ separators from each subarray of n/ℓ elements, the size of a divided subarray is at most $n/(\ell((u-1)/\ell+1)) < n/(u-1)$. Since each group consists of ℓ subarrays, the size of a group is at most $n\ell/(u-1) < n/k$. For any multiprocessors, the total size of assigned groups is larger than n/k because we can assign one more group to a multiprocessor whenever the total size of assigned groups is equal to or smaller than n/k and there are any groups not assigned. Therefore, the total size of assigned groups to the multiprocessors is at least $(n/k)\cdot k = n$. This is a contradiction. □

Let $S_{i,j}$ $(1 \le i \le \ell, 1 \le j \le u)$ denote the subarray that is a part of S_i, and now in G_j.

A multiprocessor repeatedly merges d subarrays in a group using the algorithm of Lemma 6.7 and get $\lceil \ell/d \rceil$ subarrays. In the row-wise merge part, a multiprocessor repeatedly executes this step until all subarrays in a group are merged.

Suppose a multiprocessor merges d subarrays in G_j that consists of subarrays $S^t_{1,j}, S^t_{2,j}, \ldots, S^t_{v,j}$ at step t $(1 \leq t)$ and gets a set of subarrays $S^{t+1}_{1,j}, S^{t+1}_{2,j}, \ldots, S^{t+1}_{\lceil v/d \rceil,j}$, where $v = \ell/d^{t-1}$. Let w^{t+1}_{ij} be the size of $S^{t+1}_{i,j}$, that is, $w^{t+1}_{ij} = \left|S^{t+1}_{ij}\right|$. In order to get $S^{t+1}_{i,j}$, a multiprocessor executes Heapify $\left\lceil \left|S^{t+1}_{i,j}\right| / b \right\rceil$ times. Therefore, the time complexity is

$$O\left(\left\lceil \frac{\left|S^{t+1}_{i,j}\right|}{b} \right\rceil \log b \log d\right).$$

Supposing C_x is a set of indices of groups that are assigned to multiprocessor x, the total time complexity at step t is

$$\max_x \left(\sum_{j \in C_x} \sum_{i=1}^{\lceil v/d \rceil} \left\lceil \frac{\left|S^{t+1}_{i,j}\right|}{b} \right\rceil \log b \log d \right).$$

Due to Lemma 6.8, for any multiprocessor x,

$$\sum_{j \in C_x} \sum_{i=1}^{\lceil v/d \rceil} \left|S^{t+1}_{i,j}\right| < \frac{2n}{k}.$$

Therefore, the time complexity at step t is

$$O\left(\frac{n}{p} \log b \log d \left(1 + \frac{u}{d^{s_0+1} d^{t-1}}\right)\right).$$

Let t_0 be the number of the steps in this part. Since $t_0 = O\left(\log_d \ell\right)$, the total time complexity in this part is

$$O\left(\sum_{t=1}^{t_0} \frac{n}{p} \log b \log d \left(1 + \frac{u}{d^{s_0+1} d^{t-1}}\right)\right) = O\left(\frac{n}{p} \log b \log d \left(t_0 + \frac{u}{d^{s_0+1}}\right)\right).$$

The I/O complexity to get $S^{t+1}_{i,j}$ at step t is $O\left(\left\lceil \left|S^{t+1}_{i,j}\right| / b \right\rceil\right)$ because a multiprocessor access global memory two times at each Heapify. Therefore, the I/O complexity at step t is

$$O\left(\sum_x \sum_{S''_{ij} \in C_x} \left\lceil \frac{\left|S''_{i,j}\right|}{b} \right\rceil \right) = O\left(\frac{n}{b}\left(1 + \frac{u}{d^{s_0+1} d^{t-1}}\right)\right).$$

Therefore, the total I/O complexity in this part is

$$O\left(\sum_{t=1}^{t_0} \frac{n}{b}\left(1+\frac{u}{d^{s_0+1}d^{t-1}}\right)\right) = O\left(\frac{n}{b}\left(t_0+\frac{n}{d^{s_0+1}}\right)\right).$$

6.4.6 The Complexities and the Amount of Memory Used

We calculate the time and I/O complexities by summing up those of all parts. It holds $\ell = O(n/(bd^{s_0+1}))$, and $t_0 = \log_d(n/b) - s_0$ where s_0 is the number of steps in the column-wise merging part. Supposing $n = \Omega(b^2)$, the time complexity for the entire process is

$$O\left(\frac{n}{p}\log b \log\frac{n}{b} + \log b \log d\left(\frac{n}{b\ell}+\frac{u\ell}{k}+\frac{u}{b}\right)\right),$$

and the I/O complexity for the entire process is

$$O\left(\frac{n}{b}\log_d\frac{n}{b} + u\ell\right).$$

The amount of the shared memory used is $O(bd)$ words, and the amount of the global memory used is $2n + O(u\ell)$ words.

We determine the values of ℓ and u as $\ell = k$, and $u = n/p$ so that we can eliminate the second term of the time and I/O complexities.

Furthermore, the value of d is limited by the amount of shared memory M. We select the maximum value of d. Since the algorithm uses $O(db)$ words of shared memory, we determine the value of d as $d = O(M/b)$.

Taken together, we obtain the following theorem.

Theorem 6.9 *Supposing $n = \Omega(b^2)$, comparison sorting for n elements using our algorithm runs on the AGPU(p, b, M) model with I/O complexity $O(\frac{n}{b}\log_{\frac{M}{b}}\frac{n}{b})$ and time complexity $O(\frac{n}{p}\log b\log\frac{n}{b})$.*

This algorithm has the optimal I/O complexity. The time complexity is at most $O(\log b)$ times larger than the lower bound.

6.4.7 Effect of Multiplicity

Supposing d is variable, the I/O complexity of the algorithm is $O(\frac{n}{b}\log_d \frac{n}{b})$, and the multiplicity is $O(M/db)$. When d has the largest value $O(M/b)$, the I/O complexity is equal to the lower bound $O(\frac{n}{b}\log_{\frac{M}{b}} \frac{n}{b})$, while the multiplicity has the smallest value 1. It indicates the efficiency of the global memory accesses becomes worst. On the other hand, when d is equal to two, the I/O complexity is $O(\frac{n}{b}\log \frac{n}{b})$, while the multiplicity is $O(M/b)$, which is considered to be optimal. Thus, there is a tradeoff between the I/O complexity and the multiplicity.

6.5 Experimental Evaluation

6.5.1 Parameter Tuning

We checked that real GPUs have the same tradeoff as the AGPU model and determined the value of d. We used NVIDIA Tesla k20c for all experiments. The input was 2^{28} 32-bit integers. Figure 6.8 shows the number of global memory accesses for each value of d. These values were measured with nvprof, which is provided by NVIDIA. These values do not include the number of cache accesses. The minimum and maximum values of d are 2 and 256 respectively in this environment. We can see the number of the global memory accesses decreases with increasing d.

Figure 6.9 shows the sorting rate (the number of elements processed per second) for each value of d. The sorting rate is maximum at $d = 4$. When $d > 4$, although the number of global memory accesses decreases with increasing d, multiplicity also decreases with increasing d, which causes inefficiency of global memory accesses. On the other hand, when $d \leq 4$, the value of multiplicity does not depend on the value of d because the value of multiplicity is limited by device specifications and it has maximum value 64 when $d \leq 4$. Therefore, the sorting rate only depends on the I/O complexity and increases with increasing d. Note that the time complexity is independent of the value of d. We determined the value of d as $d = 4$.

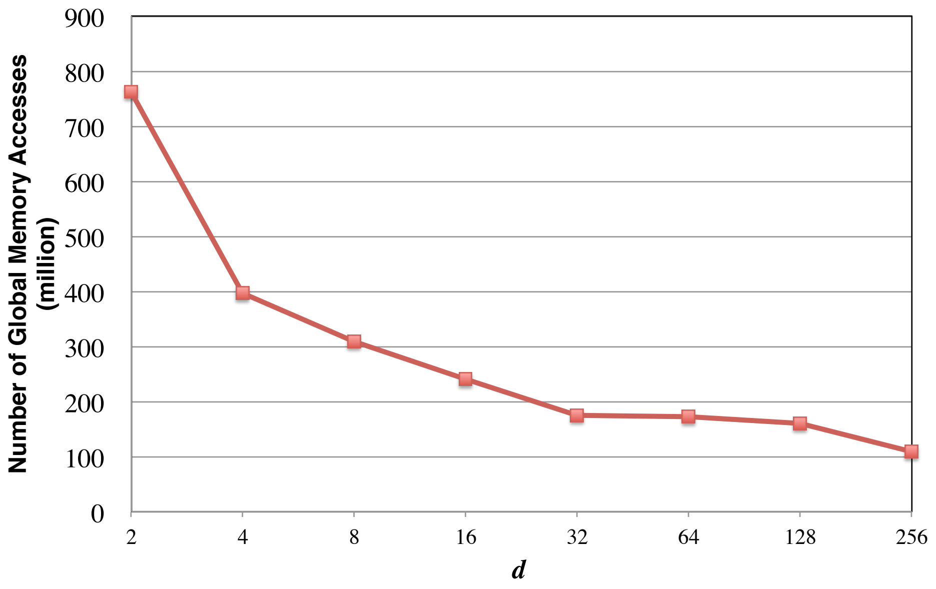


Figure 6.8: The number of the global memory accesses for each value of *d*

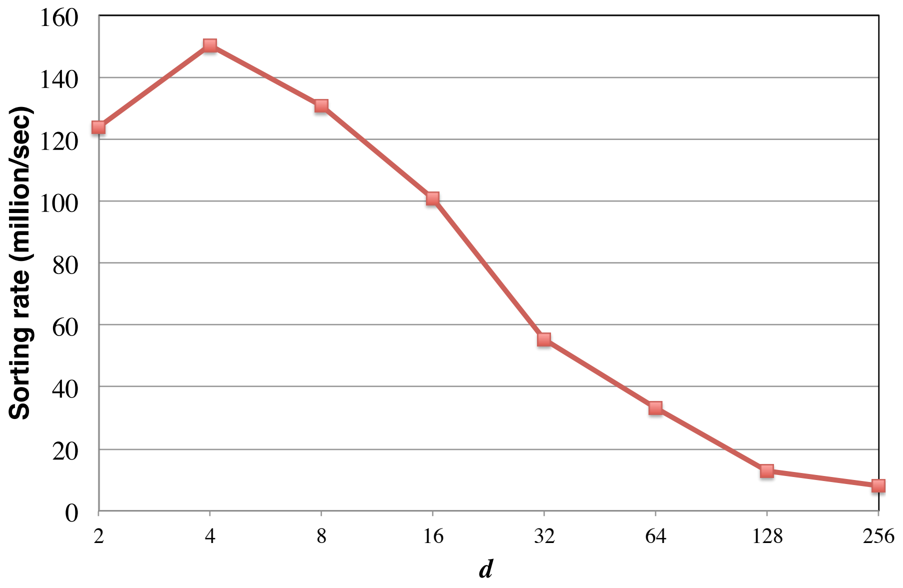


Figure 6.9: Sorting rate for each value of *d*

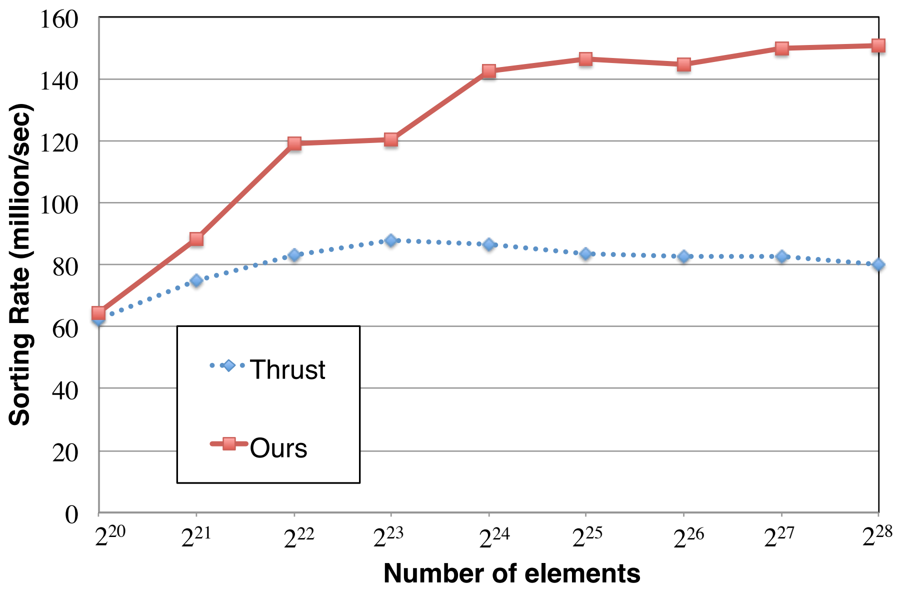


Figure 6.10: Sorting rate for our algorithm and Thrust comparison-based sorting

6.5.2 Comparison with Thrust

We compared our algorithm with Thrust comparison-based sorting. Figure 6.10 shows the sorting rate. Our algorithm was 1.9 times faster than Thrust when $n = 2^{28}$.

Figure 6.11 shows the number of global memory accesses. Thrust comparison-based sorting algorithm is similar to GPU-Warpsort and has the same I/O complexity. When $n = 2^{28}$, the number for our algorithm was equal to 27% of that for Thrust.

6.6 Short Summary

In this chapter, we discussed comparison sorting algorithms on GPUs. We first analyzed the complexities of Bitonic sort and GPU-Warpsort using the AGPU model. Moreover, we revealed the lower bound of the complexities of comparison sorting algorithm. Though GPU-Warpsort is one of the fastest comparison sort algorithms, the I/O complexity is larger than the lower bound. We therefore proposed an I/O-optimal sorting algorithm. Some parameter values of the algorithm were determined by the

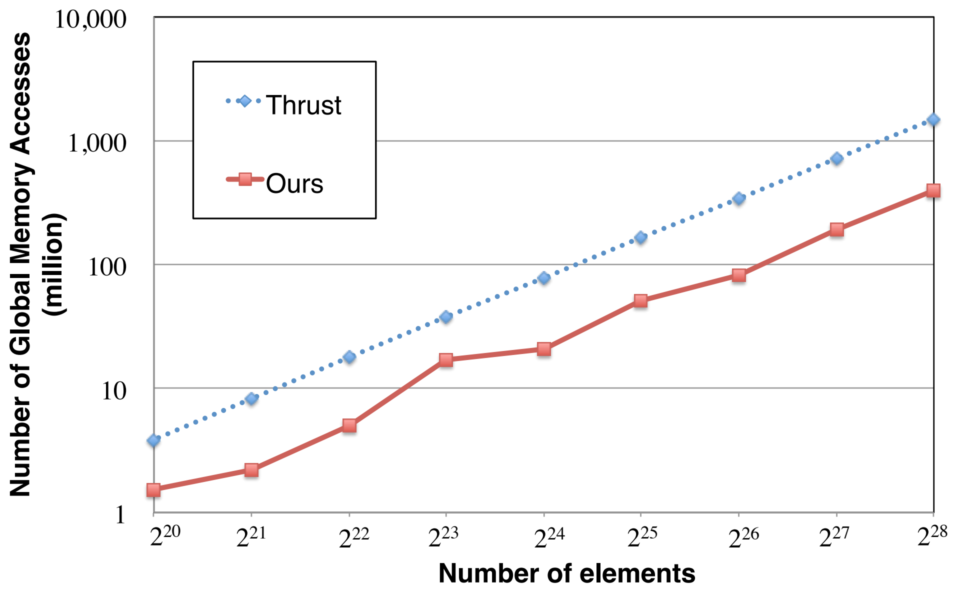


Figure 6.11: The number of global memory accesses for our algorithm and Thrust

complexity analysis using the AGPU model. Our algorithm is fast not only in theory but also in practice.

Thus, the complexity analysis using the AGPU model tells us what is the bottleneck of existing algorithms and helps us determine some parameter values.

7 Conclusion

7.1 Summary of the Dissertation

In order to make it possible to easily develop GPU-based discrete algorithms, we proposed a computational model AGPU and developed some GPU-based algorithms.

Asymptotic complexity analysis is very important to develop efficient discrete algorithms. The AGPU model is suitable for the analysis. The AGPU model abstracts the GPU's SIMT architecture using only three parameters, and it is able to take account of a lot of factors affecting the performance such as coalescing, bank conflicts, multithreading. The time and I/O complexities and the multiplicity are used to evaluate the performance of algorithms, and the amounts of the global and shared memory used are used to evaluate the memory usage of algorithms. We also revealed the relations between the AGPU model and other models including the PRAM model, the BSP model, and the I/O model. It enables us to utilize some knowledge on the other model for the AGPU model.

Next, we analyzed and developed reduction algorithms. Reduction is important because many problems can be calculated as reduction. We analyzed the tree-based algorithm and the cascading algorithm and gave the evidence that the cascading algorithm is faster than the tree-based algorithm. Namely, the time complexity of tree-based algorithm is larger than that of the cascading algorithm. Though the cascading algorithm has the optimal time and I/O complexities, it cannot be used for non-commutative operators. We therefore proposed the pipeline algorithm, which also has the optimal time and I/O complexities. Our algorithm solves the maximum segment sum problem, which is an instance of non-commutative reduction, up to 3.9 times faster than the tree-based algorithm and up to 29 times faster than the sequential algorithm on CPU. In addition, we evaluated the matrix-based algorithm to show the power of the AGPU model. Although the algorithm has the optimal time and I/O complexities on SIMD architectures, the algorithm is slow on GPUs. We can easily find the reason using the AGPU model; the reason is the small multiplicity. Then, we measured the actual running time of the algorithms and obtained the expected performance.

Next, we analyzed the time and I/O complexities of the fastest prefix scan algorithm and measured the performance on real GPUs. The algorithm has a tuning parameter α. We showed that the parameter makes tradeoff between the time complexity and the multiplicity. Then, we measured the actual running time of the algorithm with various parameter values on the real GPUs and obtained the expected performance.

Lastly, we discussed comparison sorting algorithms. Though GPU-Warpsort is one of the fastest comparison sort algorithms, we found that the I/O complexity is larger than the lower bound. We therefore proposed an I/O optimal comparison sorting algorithm. We showed our algorithm is fast not only in theory but also in practice. Our comparison sorting algorithm runs up to 1.9 times faster than GPU-Warpsort.

Thus, the complexity analysis using the AGPU model tells us what is the bottleneck of existing algorithms and helps us develop new algorithms.

7.2 Future Work

In the future, GPUs will put more and more applications into practical use by reducing computation time drastically. For example, utilizing GPUs for machine learning is actively being studied. To this end, we would like to improve our AGPU model and to develop more basic algorithms using the AGPU model.

The first challenge for the improvement of the AGPU model is to take more hardware modules into account. For example, recent GPUs from NVIDIA are equipped with texture memory, and read-only data can be efficiently accessed using the memory. Although the size of input data is substantially increasing in many research fields, input data have the read-only property in many cases. Therefore, analyzing algorithms that utilize texture memory is important. Moreover, NVIDIA has a plan to provide GPUs with 3D-stacked memory. This memory boosts memory band width drastically. In the AGPU model, I/O complexity is defined as the total number of global memory access. However, more detailed analyses are potentially required in order to evaluate efficient use of wide band width. It seems to be useful to consider work and depth [17] for the I/O complexity.

The second challenge for the improvement of the AGPU model is to expand the model to deal with multiple GPUs. We can obtain wider memory band width from multiple GPUs. On the other hand, communication between GPUs is very slow compared to memory access inside a GPU. We have to develop a model that can evaluate the communication.

The challenge for development of basic algorithms is to deal with graphs. Recently, many problems are defined on graphs. Therefore, graph algorithms are getting more and more important. GPU-based graph algorithms have difficulty avoiding bank conflicts and making global accesses coalesce. In order to tackle this problem, we would first like to develop algorithms on trees. Parallel tree contraction is an important operation on trees. Many problems on trees can be solved using tree contraction and it is also a powerful tool to design a wide class of graph algorithms. We would like to design efficient GPU-based tree contraction algorithms.

Bibliography

[1] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), March 2005.

[2] Sushant Sharma, Chung-Hsing Hsu, and Wu chun Feng. Making a case for a green500 list. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006)/ Workshop on High Performance - Power Aware Computing*, 2006.

[3] NVIDIA Corporation. NVIDIA CUDA C programming guide version 4.2, 2012.

[4] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.

[5] K. Nakano. The hierarchical memory machine model for gpus. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 591–600, 2013.

[6] Nodari Sitchinava and Volker Weichert. Provably efficient gpu algorithms. *CoRR*, abs/1306.5076, 2013.

[7] K. Kothapalli, R. Mukherjee, M.S. Rehman, S. Patidar, P.J. Narayanan, and K. Srinathan. A performance prediction model for the cuda gpgpu platform. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 463 –472, dec. 2009.

[8] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the*

36th annual international symposium on Computer architecture, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[9] Mark Harris. Optimizing parallel reduction in cuda, 2008.

[10] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. Fast scan algorithms on graphics processors. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 205–213, New York, NY, USA, 2008. ACM.

[11] Xiaochun Ye, Dongrui Fan, Wei Lin, Nan Yuan, and P. Ienne. High performance comparison-based sorting algorithm on many-core gpus. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10, 2010.

[12] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[13] John von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, June 1945.

[14] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[15] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept 1972.

[16] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[17] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

[18] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39 –55, march-april 2008.

[19] NVIDIA Corporation. NVIDIA's next generation CUDA compute architecture: Fermi, 2009.

[20] NVIDIA Corporation. NVIDIA's next generation CUDA compute architecture: Kepler gk110, 2012.

[21] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In Jan van Leeuwen, editor, *Handbook of theoretical computer science (vol. A)*, pages 869–941. MIT Press, Cambridge, MA, USA, 1990.

[22] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, Mass., 1992.

[23] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[24] Junjie Lai and André Seznec. Break down gpu execution time with an analytical method. In *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '12, pages 33–39, New York, NY, USA, 2012. ACM.

[25] Koji Nakano. Simple memory machine models for gpus. In *IPDPS Workshops*, pages 794–803, 2012.

[26] Lin Ma, Kunal Agrawal, and Roger D. Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Generation Computer Systems*, 30(0):202 – 215, 2014. Special Issue on Extreme Scale Parallel Architectures and Systems, Cryptography in Cloud Computing and Recent Advances in Parallel and Distributed Systems, ICPADS 2012 Selected Papers.

[27] Qiming Hou, Kun Zhou, and Baining Guo. Bsgp: bulk-synchronous gpu programming. *ACM Trans. Graph.*, 27(3):19:1–19:12, August 2008.

[28] R.S. Bird. Lectures on constructive functional programming. Technical Report PRG69, OUCL, September 1988.

[29] Murray Cole. Parallel programming, list homomorphisms and the maximum segment sum problem. Technical report, Proceedings of Parco 93. Elsevier Series in Advances in Parallel Computing, 1993.

[30] Akimasa Morihata. *Calculational Approach to Automatic Algorithm Construction.* PhD thesis, The University of Tokyo, Japan, March 2009.

[31] R. S. Bird. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design*, pages 5–42, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[32] Kenta Hongo, Ryo Maezono, and Kenichi Miura. Random number generators tested on quantum monte carlo simulations. *Journal of Computational Chemistry*, 31(11):2186–2194, 2010.

[33] Takeshi Fukuda, Yasukiko Morimoto, Shinichi Morishita, and Takeshi Tokuyama. Data mining using two-dimensional optimized association rules: Scheme, algorithms, and visualization. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 13–23, New York, NY, USA, 1996. ACM.

[34] Jon Bentley. *Programming Pearls.* ACM, New York, NY, USA, 1986.

[35] Guy E. Blelloch. Prefix sums and their applications. In John H. Reif, editor, *Synthesis of Parallel Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[36] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with cuda. In Hubert Nguyen, editor, *GPU Gems 3.* Addison Wesley, August 2007.

[37] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 666–675, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[38] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[39] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.3.0.

[40] Scott Le Grand. Broad-phase collision detection with cuda. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, August 2007.

[41] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 46:1–46:12, New York, NY, USA, 2007. ACM.

[42] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[43] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 351–362, New York, NY, USA, 2010. ACM.

[44] DUANE MERRILL and ANDREW GRIMSHAW. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu. *Parallel Processing Letters*, 21(02):245–272, 2011.

[45] Duane G. Merrill and Andrew S. Grimshaw. Revisiting sorting for gpgpu stream architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 545–546, New York, NY, USA, 2010. ACM.

[46] Vasileios Kolonias, Artemios G. Voyiatzis, George Goulas, and Efthymios Housos. Design and implementation of an efficient integer count sort in cuda gpus. *Concurr. Comput. : Pract. Exper.*, 23(18):2365–2381, December 2011.

[47] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[48] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '03, pages 41–50, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[49] Ian Buck and Tim Purcell. A toolkit for computation on GPUs. In Randima Fernando, editor, *GPU Gems*, chapter 37. Addison Wesley, 2004.

[50] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM.

[51] Alexander Greßand Gabriel Zachmann. Gpu-abisort: optimal parallel sorting on stream architectures. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 45–45, Washington, DC, USA, 2006. IEEE Computer Society.

[52] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. A novel sorting algorithm for many-core architectures based on adaptive bitonic sort. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 227–237, Washington, DC, USA, 2012. IEEE Computer Society.

[53] Peter Kipfer and Rüdiger Westermann. Improved GPU sorting. In Matt Pharr, editor, *GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 733–746. Addison-Wesley, 2005.

[54] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[55] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:4:1.4–4:1.24, January 2010.

[56] N. Leischner, V. Osipov, and P. Sanders. Gpu sample sort. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10, April 2010.

Publication List

International Journal

1. (Refereed) Atsushi Koike, and Kunihiko Sadakane. *A Novel Computational Model for GPUs with Applications to Efficient Algorithms.* International Journal of Networking and Computing (IJNC), Volume 5, Number 1, pages 26–60, January 2015.

International Conferences

2. (Oral，refereed) Atsushi Koike, and Kunihiko Sadakane. *A Novel Computational Model for GPUs with Application to I/O Optimal Sorting Algorithms.* 2014 IPDPS Workshops (APDCM), Arizona USA, May. 2014.

3. (Oral，refereed) Atsushi Koike, and Kunihiko Sadakane. *Abstract Computation Model for Analyzing Complexity of GPU-based Algorithms.* The 6th Annual Meeting of Asian Association for Algorithms and Computation (AAAC2013), April. 2013.

Domestic Conferences

4. (Oral) 小池　敦，定兼　邦彦．*GPU*を用いた並列ソートアルゴリズム．第10回情報科学ワークショップ，E-3，2014年9月．

5. (Oral) 小池　敦，定兼　邦彦．*GPU*向け比較ソートアルゴリズムの実装と評価．2014年電子情報通信学会大会講演論文集，DS-1-9，2014年3月．

6. (Poster，refereed) 小池　敦，定兼　邦彦．*GPU*向けの*I/O*最適な比較ソートアルゴリズム．ハイパフォーマンスコンピューティングと計算科学シンポジウム（HPCS2014），2014年1月．
7. (Oral) 小池　敦，定兼　邦彦．*GPU*を用いた並列ソートアルゴリズムの実装と評価．情報処理学会研究報告. アルゴリズム研究会報告，AL 145-8，2013年11月．
8. (Poster，refereed) 小池　敦，定兼　邦彦．*GPU*アルゴリズム解析のための並列計算モデル．GPU Technology Conference Japan 2013年7月．
9. (Oral) 小池　敦，定兼　邦彦，Hoa Vu．*AGPU*モデルでの並列ソートアルゴリズムの計算量について．電子情報通信学会技術研究報告，COMP2013 pp.75-80，2013年5月．
10. (Oral) 小池　敦，定兼　邦彦．*AGPU*モデルにおけるマルチスレッディングの効果．2013年電子情報通信学会大会講演論文集，DS-1-13，2013年3月．
11. (Poster，refereed) 小池　敦，定兼　邦彦．*GPU*のための並列計算モデル．2013年ハイパフォーマンスコンピューティングと計算科学シンポジウム（HPCS2013），2013年1月．
12. (Oral) 小池　敦，定兼　邦彦．*GPU*のための並列計算モデル．電子情報通信学会技術研究報告，COMP，112(272) 53-60, 2012年10月．