International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 5, Number 1, pages 26–60, January 2015

A Novel Computational Model for GPUs with Applications to Efficient Algorithms

Atsushi Koike
Information Systems Architecture Research Division, National Institute of Informatics, 2-1-2 Hitotsubashi
Chiyoda-ku, Tokyo, 101-8430, Japan
and
The Department of Informatics, The Graduate University for Advanced Studies, 2-1-2 Hitotsubashi
Chiyoda-ku, Tokyo, 101-8430, Japan

and

Kunihiko Sadakane
Department of Mathematical Informatics, Graduate School of Information Science and Technology, The University of Tokyo, 7-3-1 Hongo
Bunkyo-ku, Tokyo, 113-8656, Japan

Received: July 31, 2014
Revised: September 29, 2014
Accepted: December 3, 2014
Communicated by Susumu Matsumae

Abstract

We propose a novel computational model for GPUs. Known parallel computational models such as the PRAM model are not appropriate for evaluating GPU-based algorithms. Our model, called *AGPU*, abstracts the essence of current GPU architectures such as global and shared memory, memory coalescing and bank conflicts. Using our model, we can evaluate asymptotic behavior of GPU algorithms more efficiently than the known models and we can develop algorithms that run fast on real GPU devices.

As a showcase, we analyze the asymptotic behavior of basic existing algorithms including reduction, prefix scan, and comparison sorting. We further develop new algorithms by detecting and resolving performance bottlenecks of the existing algorithms. Our reduction algorithm has the optimal time and I/O complexities and works with non-commutative operators. Our comparison sorting algorithm has the optimal I/O complexity. Additionally, we show our algorithms run faster than the existing algorithms not only in theory but also in practice.

Keywords: GPU, GPGPU, parallel computational models, reduction algorithms, prefix scan algorithms, sorting algorithms

1 Introduction

Parallel architectures are becoming more important as processor clock speeds are beginning to reach a limit. Graphics Processing Units (GPUs) were originally designed for efficient processing of graphics, but nowadays they are widely used for a variety of parallel computation applications because they are equipped with high memory bandwidth and high parallelism. This approach is known as General-purpose GPU (GPGPU).

GPUs have unique architectures for efficient processing with many cores. We therefore have to consider the architectures carefully to develop fast algorithms. NVIDIA [1] 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 the case of sequential algorithms, Random-Access Machine (RAM) model is commonly used to estimate computational complexities of algorithms. Because the RAM is a unifying abstracted machine for all sequential machines, the complexities are useful for all sequential machines. 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) [2, 3, 4] models, which consist of multiple cores and a single shared memory unit, are standard computational models for parallel algorithms. However, algorithms developed on the models do not always show good performance on GPUs because the PRAM models are substantially different from actual GPU architectures. For estimating the performance of GPU-based algorithms, several models have been proposed [5, 6, 7, 8]. Hong et al. [9] and Kothapalli et al. [5] have proposed analytical models to estimate actual running time of GPU-based algorithms without executing applications on GPUs. Ma et al. [7] and Nakano [8] have proposed memory access models that take memory access latency into account.

In this paper, we propose a novel parallel computational model called AGPU. The AGPU model focuses on analyzing asymptotic computational complexities of GPU-based algorithms, while the previous models relatively focus on predicting actual running time of the algorithms. The purpose of the analyses using the AGPU model is to grasp where the bottleneck for performance is. Therefore the AGPU model aims to be simple and able to takes 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. Sitchinava and Weichert [10] have proposed an algorithmic model for GPUs independently of the AGPU model. The AGPU model is simpler than their model and therefore it can analyze the asymptotic complexities more easily. Moreover, we can show the AGPU model has a lot of relations to other models, which is useful for designing effective algorithms.

We next analyze some basic algorithms. First, we analyze GPU-based reduction algorithms. There exist two main algorithms for reduction: tree-based algorithm and cascading algorithm. The latter is faster than the former in practice, and we give evidence using the AGPU model; the latter has a lower time complexity than the former. We also give a novel and efficient algorithm for reduction with non-commutative operators. Next, we analyze a prefix scan algorithm. We prove the algorithm has a tradeoff between the time complexity and the effect of multithreading in the AGPU model, and check that the running time of the algorithm in practice shows the same tendency. Next, we analyze comparison sorting algorithms. We show the I/O complexities of existing algorithms are not optimal and develop a new algorithm that has the optimal I/O complexity.

The rest of the paper is organized as follows. Section 2 gives a brief overview of GPU architectures and GPU programs. In Section 3, we describe our computational model AGPU. In Section 4, we explain the relations between the AGPU model and other computational models. Section 5 deals with reduction algorithms. Section 6 deals with prefix scan algorithms. Section 7 deals with comparison sorting algorithms. Section 8 concludes the paper.

2 GPU Architectures and GPU Programs

We briefly explain GPU architectures. A lot of GPU architectures are proposed recently, but most architectures have similar characteristics. In this paper, we explain Tesla architecture [11] proposed by NVIDIA.

The Tesla architecture is a hybrid system of CPU and GPU devices. The GPU device comprises multiple cores called *streaming processors (SPs)* organized as multiprocessors called *streaming multiprocessors (SMs)*. For instance, C1060 model comprises 128 SPs organized as 16 SMs. Each SM

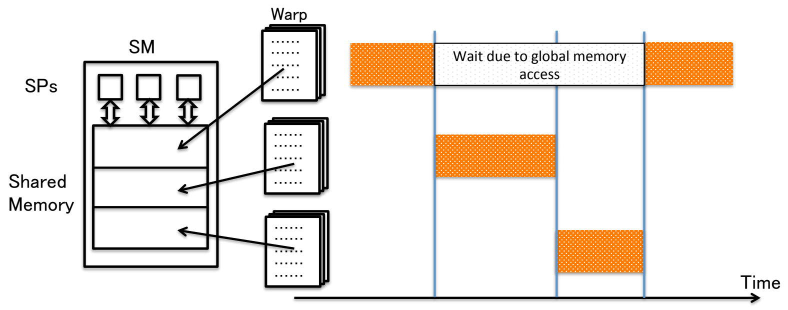

Figure 1: An example of Multithreading

individually executes programs and it does not have communication means with other SMs. The CPU invokes GPU programs and only the CPU can synchronize SMs. On the other hand, all SPs in an SM execute the same instruction at the same time. The Tesla architecture has mainly three types of memory. The first one is a *global memory*, which can be accessed from all SMs. The global memory is stored in DRAM. Therefore accesses to the global memory take longer time than arithmetic operations. The second one is a *shared memory*, which can be accessed only from SPs in an SM. Each SM has a shared memory that is stored in SRAM inside the SM. The third one is *registers* in SPs. The Tesla architecture has a large set of registers that can be used for storage space for internal process.

NVIDIA provides a programming model called *CUDA*. Programs implemented using CUDA can be executed on all NVIDIA GPUs. GPU programs launched by CPU are called *kernels*. A kernel has 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. A block is a set of threads that are executed by the same SM. An SM may be assigned more than one blocks. Programmers are unconscious of the assignment of blocks to SMs. One thread is executed by a single SP, and each SP concurrently executes multiple threads.

Each SM executes threads in a block in groups of 32 threads called *warps*. In the C1060 model, the 8 SPs in an SM execute an instruction for 32 threads in four cycles. Though we cannot directly specify how to process warps by SMs using CUDA, it is important to design programs with consideration for warps to obtain high performance, for example, high execution efficiency by designing programs so that all threads in a warp go through the same path at conditional branches. It is also important to consider memory access patterns including bank conflicts and coalescing, which will be described in Section 3.1.

Each multiprocessor executes multiple warps concurrently, namely, each multiprocessor can switch the processing warp dynamically at the hardware level. In the C1060 model, each SM can execute up to 24 warps concurrently. It is called *hardware multithreading* or *multithreading*. Multithreading is a key factor for GPUs to efficiently access the global memory. When a warp waits for data to load from the global memory, the multiprocessor executes the other warps in the meantime. Thus, global memory load latency can be hidden by multithreading. Figure 1 shows an example. Note that the maximum number of warps executed concurrently is constrained by the amount of shared memory and register used by warps. Because all warps in an SM share the same memory, the number is limited by the amount of memory used.

3 AGPU Model

3.1 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 related to

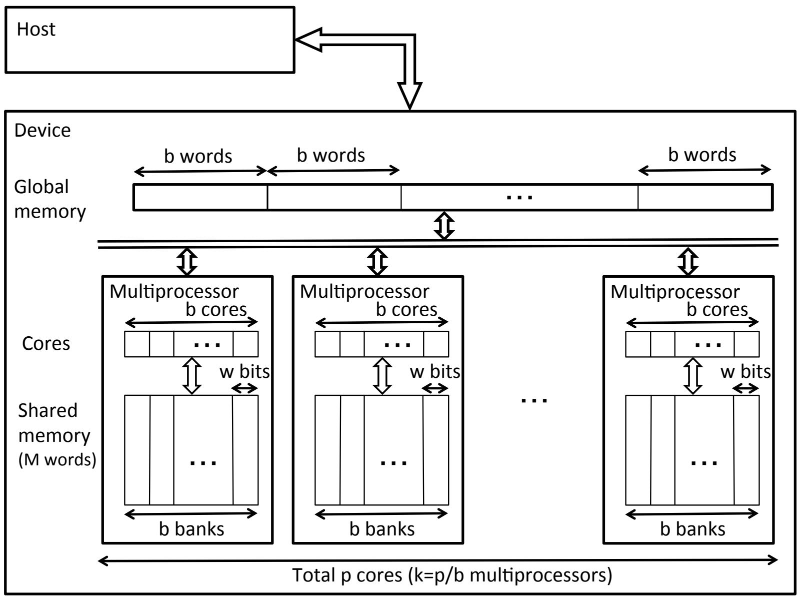

Figure 2: The architecture of AGPU model

performance and make the model as simple as possible. Figure 2 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*. 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 launched by the host.

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.

All cores in a multiprocessor execute the identical instruction at the same time, but data addresses of their operands can be arbitrary. In other words, all cores 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*. Figure 3 shows examples of global memory accesses. In Figure 3(a), all memory access instructions coalesce and are executed in a unit time. On the other hand, in Figure 3(b), the instructions are executed in 4 unit times because the words are spread out among 4 blocks. Furthermore, the number of multiprocessors that can access the global memory simultaneously is limited, that is, the bandwidth of the global memory is within a constant factor of $b \cdot w$ bits.

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 4 shows the examples of shared memory accesses. The columns of the shared memory unit represent the banks. In Figure 4(a), all cores access distinct banks, therefore bank conflicts do not occur. On the other hand, in Figure 4(b), the second column is accessed by two cores, therefore the instruction is divided into two instructions.

We denote this model by AGPU(p, b, M, w). We may omit the parameters w and M if they do

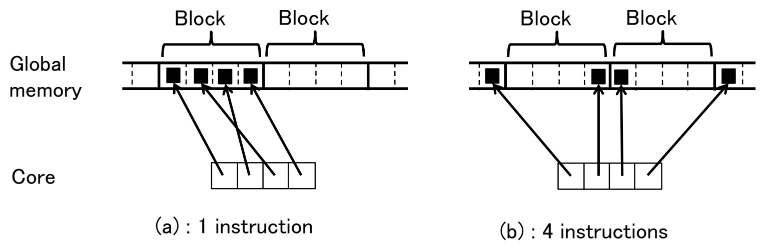

Figure 3: Examples of global memory accesses. Each block can store b word. (a) All instructions coalesce. (b) These instructions do not coalesce because the words are spread out among 4 blocks.

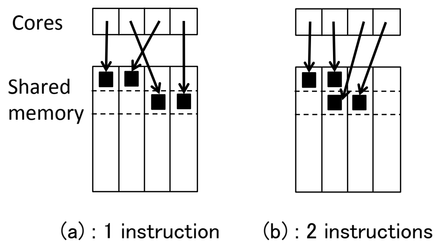

Figure 4: Examples of shared memory accesses. Each column of the shared memory units represents a memory bank, and each cell can store one word. (a) Bank conflicts do not occur since no banks are accessed by multiple cores. (b) The instruction is divided into two instructions since the second column is accessed by two cores (bank conflict).

not affect the performance of algorithms. In this case we denote the model by AGPU(p, b, M) or AGPU(p, b).

We consider two variations 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 execution. The CUDA environment uses the volatile model. Therefore we denote the volatile model as AGPU, and the non-volatile model as AGPU$'$.

3.2 Metrics

To evaluate the performance of algorithms, we use four metrics: the *time complexity*, the *I/O complexity*, the *amount of the global memory used* and the *amount of the shared memory used*. 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 are used to evaluate the memory usage of algorithms. If the amount of the shared memory varies according to the multiprocessors, the largest amount is adopted. It is important to reduce the amount of the global memory used, especially if the input size is large. Furthermore, if the amount of the shared memory used is larger than M words, the algorithm cannot be implemented on GPU. Additionally, as we discuss later, a large amount of the shared memory used makes multithreading less effective.

3.3 The Effect of Multithreading

As mentioned in Section 2, real GPU devices have a mechanism called multithreading. Altough 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, 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 larger than M/b^2, but at most M/b. Namely, when $m = \mathcal{O}(b)$, we can assign the maximum number of warps to the multiprocessors. Therefore, when $m = \mathcal{O}(b)$, we say the multiplicity is optimal. On the other hand, when $m = \Omega(b^2)$, we consider the multiplicity is not optimal.

3.4 Divergence from Real Architectures

Since our model is designed for analyzing time and I/O complexities, it is rather simplified. We discuss the divergence 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.

Secondly, 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 at most a constant factor of those in real GPU devices.

Thirdly, 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 4.9, 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 3.3. Therefore it is not necessary to use synchronization in a multiprocessor.

3.5 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 miltiprocessor. When multiple multiprocessors execute programs 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 execute multiple multiprocessors on AGPU(p, b, M), programmer do not need to care the assignment.

When all cores in the multiprocessor execute programs 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 global memory access instructions. We can access at most b contiguous 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 two times to execute the instruction. However, it do not change the asymptotic I/O complexity.

The symbols “$\rightarrow$” and “$\leftarrow$” represent 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.

3.5.1 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 fast.

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, the amount of shared memory used is reduced, which leads to make the multiplicity large. Since the number of blocks assigned to an SM is limited, it is also effective for large multiplicity to make the number of the warps in a block increased to more than one. 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.

4 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 4.1 *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 $X \leq \mathcal{O}(1)Y$ and $Y \leq \mathcal{O}(1)X$, we denote this $X = Y$. We define $X_{IO} = Y_{IO}$ analogously.*

4.1 PRAM Model

Parallel random access machine (PRAM) model [2, 3, 4] has p processors that can execute arbitrary instructions with a constant number of operands simultaneously. The p processors have a shared memory unit of M words. Each of the processors can read/write from/to an arbitrary address in parallel. We consider only EREW (exclusive-read, exclusive-write) PRAM model; all processors access distinct addresses on the memory at the same time. We denote the model by PRAM(p, M).

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. In the AGPU model, the number of multiprocessors that access the global memory simultaneously is limited. Moreover, unless all cores inside a multiprocessor access contiguous 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$(p, g + \frac{pM}{b})$ algorithm running in the same time complexity, that is, PRAM$(p, g + \frac{pM}{b}) \leq \mathcal{O}(1)$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 4.2 *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$(p, p, M) \leq \mathcal{O}(\frac{M}{p})$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 $\left\lceil \frac{M}{p} \right\rceil$. $\square$

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

4.2 Bulk Synchronous Parallel Model

Bulk Synchronous Parallel (BSP) model [12] 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. 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 of the p processors, the maximum size of sent and received messages among the processors, and the time for synchronization.

We can implement a BSP algorithm using AGPU$'(p, 1)$, that is, each multiprocessor has only one core. Each of p the multiprocessors corresponds to a processor of the BSP model. Communication between processors in the BSP model is done by using the global memory of the non-volatile AGPU model. However the cost of communication in the simulation by the AGPU model is higher than that in the BSP model because the global memory cannot be accessed by multiple multiprocessors in the AGPU model.

4.3 I/O Model

The standard *I/O model* [13] 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).

Lemma 4.3 *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). 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 4.4 *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 3.1, 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 4.3 and 4.4, it is obvious that:

Theorem 4.5 *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 4.4 does not hold in this case.

Lemma 4.6 *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 4.4. 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 multiprocessor in AGPU(p, b, M) store in the shared memory. As with Lemma 4.4, 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 same I/O complexity as the algorithms on AGPU$(b, b, (p/b)M)$.

From Lemmas 4.3 and 4.6, we have:

Theorem 4.7 *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 4.8 *For any algorithm on the non-volatile AGPU using s synchronizations,*

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

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 $\mathcal{O}\left(\frac{spM}{b}\right)$ I/Os are enough for the simulation. □

4.4 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 4.9 *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 4.4, 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 4.9. For example, if the time complexity of an algorithm on AGPU(v, b, M) is $\mathcal{O}(n/v + \log v)$, the time complexity in case that the algorithm is executed on AGPU(p, b, M) is $\mathcal{O}(n/p + v \log v/p)$.

5 Reduction Algorithms

5.1 Reduction with a Commutative Operator

In this section, we analyze reduction algorithms as an example of algorithm analyses using the AGPU model. The reduction operation is defined as follows. 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 paper. 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 and an element of the array stores a w bit number. In this section we assume the operator $\oplus$ is commutative. We will present a novel algorithm for non-commutative operators in the next section.

We describe two algorithms suggested by Harris [14] using the AGPU model and analyze the time and I/O complexities and the amount of memory used. Harris suggested seven algorithms for reduction [14]. 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 in the AGPU model.

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 can omit some processes from the algorithms since we always use a single multiprocessor. However, we do not go into the detail of this case because we do not take full advantage of GPUs in this case. We mainly consider the cases where the input size n is much larger than p. In this section, if $p = o(n)$ holds, we say the input size is sufficiently larger than the number of cores.

5.1.1 Tree-based Algorithm

We describe the tree-based algorithm that is the fastest among the six algorithms. We first explain the outline of the tree-based algorithm. The input $T[0..n-1]$ is divided into blocks with $2b$ words and the reduction of each block is calculated by a single multiprocessor. When the reduction of all blocks is done, 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.

The reduction inside a multiprocessor is done as follows. Each multiprocessor reads the first half and the second half of a block in turn and carries out the operation to those elements two by two using b cores in the multiprocessor. Since each block has $2b$ elements, the size of the result is b. We

Algorithm 1 CalculateReductionUsingTreeBased(T, n)

```
Input: An array $T[0..n-1]$
Output: The reduction value of $T[0..n-1]$

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

repeat the same operation until the size becomes one and we get the reduction value of the block. In this procedure, cores in a multiprocessor access b contiguous elements in each step. This means that cores access distinct banks, therefore there are no bank conflicts. A pseudo code for tree-based algorithm is shown in Algorithm 1.

We analyze the time complexity of this tree-based algorithm. The loop on lines 13-16 runs $\log b$ times. So it takes $O(\log b)$ times to calculate the reduction for one block. The loop on lines 4-26 runs $\lceil \log_{2b} n \rceil$ times. Let $s(i)$ be the number of times that the loop in lines 6-22 are executed at the i-th iteration of the for loop in lines 4-26. We have

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

Therefore, the time complexity of the algorithm is

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

Since $b \ge 1$, we have

$$\sum_{i=1}^{\lceil \log_{2b} n \rceil} \left(\frac{1}{2b} \right)^{i-1} \le \sum_{i=1}^{\infty} \left(\frac{1}{2b} \right)^{i-1} = \frac{1}{1-\frac{1}{2b}} \le 2.$$

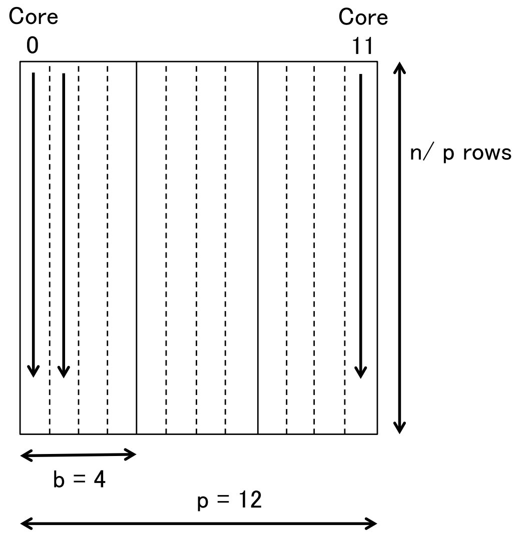

Figure 5: Input sequence arranged as a matrix with p columns.

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

We analyze the I/O complexity. Since each block is accessed three times in the loop of lines 8-20 and the number of blocks is $\lceil n/(2b)^i \rceil$, the multiprocessors access the global memory $3 \lceil n/(2b)^i \rceil$ times at the i-th iteration. Therefore, the I/O complexity is

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

If the data size is sufficiently larger than the number of cores, the I/O complexity is $\mathcal{O}(\frac{n}{b})$. Additionally, the amount of the shared memory used is $\mathcal{O}(b)$ words and the amount of the global memory used is $n + \mathcal{O}(n/b)$ words. The multiplicity is M/b.

5.1.2 Cascading Algorithm

We describe the cascading algorithm. The input is represented as a matrix with p columns. We use the row-major order; the first p elements in the input array are considered to be stored in the first row. Each of p cores is assigned to one of the columns in the matrix. Figure 5 shows an example for $b = 4$ and $p = 12$.

Each core calculates the reduction of one column sequentially. Then, 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. Algorithm 2 shows a pseudo code for the cascading algorithm. The details of the local reduction is shown in Algorithm 3.

We analyze the time complexity. Lines 1-17 in Algorithm 3 can be computed in $\mathcal{O}(n/p + \log b)$ times. The global reduction can be computed in $\mathcal{O}(\log k)$ times using the tree-based algorithm for the k resulting values. Thus, the time complexity is $\mathcal{O}(\frac{n}{p} + \log p)$. If the data size is sufficiently larger than the number of cores, the time complexity is $\mathcal{O}(\frac{n}{p})$.

Next, we analyze the I/O complexity. Since each multiprocessor can always get b values at a time, lines 1-17 in Algorithm 3 accesses to the global memory $\mathcal{O}(n/b)$ times. The global reduction algorithm accesses to the global memory $\mathcal{O}(k/b)$ times using the tree-based algorithm. Since we

Algorithm 2 CalculateReductionUsingCascading(T, n)

Input: An array $T[0..n-1]$

Output: The reduction value of $T[0..n-1]$

```
$W[0..p/b-1]$ = CalculateLocalReductionUsingCascading($T, n$) {Local reduction}
return CalculateReductionUsingTreeBased($W, p/b$) {Global reduction}
```

Algorithm 3 CalculateLocalReductionUsingCascading(T, n)

Input: An array $T[0..n-1]$

Output: The array $W[0..p/b-1]$ of the local reduction values each of which is calculated by a multiprocessor

```
for all $\rho \in MP[0..k-1]$ in parallel do
  for all $\epsilon \in Core[0..b-1]$ in parallel do
    $x[\epsilon] \leftarrow 0$ {Reduction value so far}
    for ($i = 0; i < n/p; i++$) do
      $d[\epsilon] \Leftarrow T[ip + b\rho + \epsilon]$ {Each multiprocessor reads the $i$-th row}
      $x[\epsilon] \leftarrow x[\epsilon] \oplus d[\epsilon]$
    end for
    $\delta \leftarrow b/2$
    while ($\delta > 0$) do
      $x[\epsilon] \leftarrow x[\epsilon] \oplus x[\epsilon + \delta]$
      $\delta \leftarrow \delta/2$
    end while
    if ($\epsilon == 0$) then
      $W[\rho] \Leftarrow x[0]$ {Each multiprocessor writes the reduction value so far}
    end if
  end for
end for
return $W[0..p/b-1]$
```

assume $n > p$, the I/O complexity is $\mathcal{O}\left(\frac{n}{b}\right)$. The amount of the shared memory used is $\mathcal{O}(b)$ words and the amount of the global memory used is $n + \mathcal{O}(p/b)$ words. The multiplicity is $\mathcal{O}(M/b)$.

To sum it up, we confirmed that the cascading algorithm is faster than the tree-based algorithm not only in practice but also in the AGPU model. If the data size is sufficiently larger than the number of cores, the cascading algorithm is $\mathcal{O}(\log b)$ times faster than the tree-based algorithm.

5.2 Reduction with a Non-commutative Operator

The two algorithms described in the previous section use commutativity of the reduction operator for efficient memory access. Therefore, these algorithms do not work if the operator is non-commutative. In this section, we propose a novel reduction algorithm for a non-commutative operator. The basic idea is that reduction procedure is divided into several pipeline stages and the cores in each multiprocessor are assigned to pipeline stages. We call this *pipeline algorithm*.

As shown in Figure 6(a), the input array is represented as a matrix with b columns and n/b rows in the row-major order. The n/b rows are divided into k groups, each of which has n/p rows. Each of the k multiprocessors calculates the reduction of a group.

Next, we explain how each multiprocessor calculates the reduction of a group. Figure 6(b) shows an example of shared memory allocation for $b = 4$ and Figure 7 shows the process of calculating the reduction in this example. We suppose that we can carry out one operation per unit time $t_1, t_2, \cdots$.

Each multiprocessor uses $2b$ words of the shared memory. The elements in the first row are

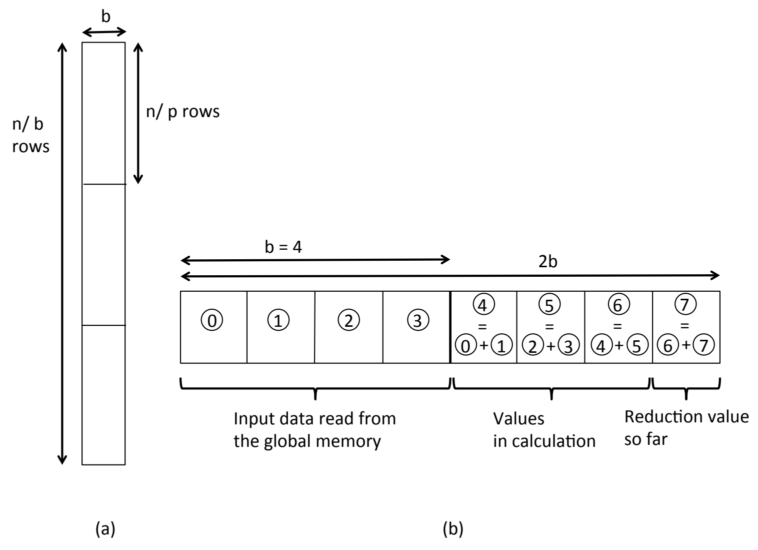

Figure 6: Memory allocation of the pipeline algorithm. (a) Input sequence arranged as a matrix with b columns. (b) An example of the shared memory allocation for $b = 4$.

processed at time $t = t_1$ to t_6. First, the row is copied to ⓪①②③ in the shared memory at $t = t_1$. Then $b/2$ cores carry out the operation to $b/2$ pairs of values and store the resulting $b/2$ reduced values to the shared memory. In this example, the resulting values are stored to ④⑤ in the shared memory at time $t = t_2$. Next, $b/4$ cores carry out the operation to $b/4$ pairs of values and store the resulting $b/4$ reduced values to the shared memory. In this example, the resulting value is stored to ⑥ in the shared memory at $t = t_4$. This procedure is repeated until the number of the resulting values becomes one. Finally, one core carries out the operation to the resulting value and the reduced value so far, and it stores the resulting value to the shared memory. In this example, the resulting value is stored to ⑦ in the shared memory at time $t = t_6$.

The elements in the second row are processed similarly at time t_3, t_4, t_6, t_8, and those in the third row are processed at time t_5, t_6, t_8, t_{10}. It is clear that anytime only b cores are used simultaneously. Therefore this pipelining works.

When we calculate the reduction of one block using tree-based algorithm, the number of active cores decreases as the calculation proceeds. On the other hand, using the pipeline algorithm, all cores are always active. Therefore the pipeline algorithm is fast.

Next, we explain the shared memory layout to avoid bank conflicts. Each multiprocessor stores $2b$ values in the shared memory according to the following rules.

1. Arrange values with even indices in Figure 6(b) to the first row of the shared memory in ascending order.

2. Arrange values with odd indices to the second row so that values with indices from $b/2$ to $b-1$ are to the first half and those with indices from 0 to $b/2-1$ are to the second half.

Figure 8 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 between cores and banks of the shared memory at any time. The cores access the shared memory three times.

1. When we store the input elements to the shared memory, the core with index i handles the element with index i. In this case, if the bank index β is smaller than $b/2$, the corresponding core is 2β. Otherwise, the corresponding core is $2(\beta - b/2) + 1$.

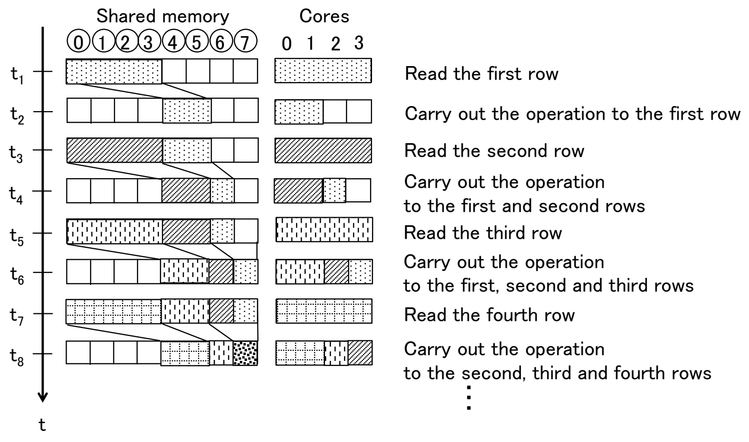
Shared memory
Cores
0 1 2 3 4 5 6 7
0 1 2 3
t1
t2
t3
t4
t5
t6
t7
t8
t
Read the first row
Carry out the operation to the first row
Read the second row
Carry out the operation to the first and second rows
Read the third row
Carry out the operation to the first, second and third rows
Read the fourth row
Carry out the operation to the second, third and fourth rows

Figure 7: An example of the procedure of the pipeline algorithm.

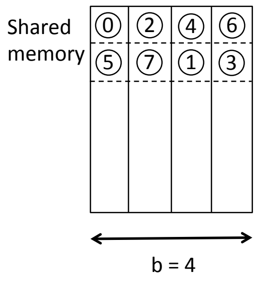

Figure 8: An example of the shared memory layout to avoid bank conflicts.

2. When we load the first elements to carry out the operation, the core with index i handles the element with index $2i$. In this case, the core corresponding to the bank β is β.

3. When we load the second elements to carry out the operation, the core with index i handles the element with index $2i + 1$. In this case, if the bank index β is smaller than $b/2$, the corresponding core is $\beta + b/2$. Otherwise, the corresponding core is $\beta - b/2$.

Thus, the mapping from the cores to the banks is bijective in all cases. Consequently, we conclude bank conflicts do not occur. Algorithm 4 shows a pseudo code for the pipeline algorithm. The details of the local reduction is shown in Algorithm 5. Algorithm 4 uses the tree-based algorithm for the global reduction. We need to change Algorithm 1 slightly for non-commutative operation. A pseudo code for the tree-based reduction algorithm with non-commutative operators is shown in Algorithm 6. Algorithm 6 always carries out the operation to two consecutive elements. Asymptotic behavior of Algorithm 6 is the same as that of Algorithm 1.

Next, we analyze the complexities of the algorithm. Lines 1-29 in Algorithm 5 can be computed

Algorithm 4 CalculateReductionUsingPipeline(T, n)

```
Input: An array $T[0..n-1]$
Output: The reduction value of $T[0..n-1]$

1: $W[0..p/b-1]$ = CalculateLocalReductionUsingPipeline($T, n$) {Local reduction}
2: return CalculateReductionUsingTreeBased2($W, p/b$) {Global reduction}
```

Algorithm 5 CalculateLocalReductionUsingPipeline(T, n)

Input: An array $T[0..n-1]$

Output: The respective reduction values of n/b blocks generated by dividing $T[0..n-1]$

```
1: for all $\rho \in MP[0..k-1]$ in parallel do
2:     for all $\epsilon \in Core[0..b-1]$ in parallel do
3:         if $(\epsilon\%2 \; != \; 0)$ then
4:             $i_s \leftarrow \epsilon/2 + 3b/2$
5:             $i_r \leftarrow \epsilon/2 + b$
6:         else
7:             $i_s \leftarrow \epsilon/2$
8:             $i_r \leftarrow \epsilon/2 + b/2$
9:         end if
10:        $i_1 \leftarrow \epsilon$
11:        if $(\epsilon < b/2)$ then
12:            $i_2 \leftarrow \epsilon + 3b/2$
13:        else
14:            $i_2 \leftarrow \epsilon + b/2$
15:        end if
16:        for $(j = 0; j < n/p; j++)$ do
17:            $x[\epsilon] \Leftarrow T[\rho nb/p + bj + epsilon]$ {Each multiprocessor reads the $j$-th row}
18:            $y[i_s] \leftarrow x[\epsilon]$ {Each multiprocessor copies input to $y[i_s]$}
19:            $y[i_r] \leftarrow y[i_1] + y[i_2]$ {Each multiprocessor carries out the operation}
20:        end for
21:        for $(j = 0; j < \log b; j++)$ do
22:            $y[i_s] \leftarrow 0$
23:            $y[i_r] \leftarrow y[i_1] + y[i_2]$
24:        end for
25:        if $(\epsilon == 0)$ then
26:            $W[\rho] \Leftarrow y[3b/2 - 1]$ {Each multiprocessor writes the reduction value so far}
27:        end if
28:    end for
29: end for
30: return $W[0..p/b-1]$
```

in $\mathcal{O}(n/p + \log b)$ time. The global reduction can be computed in $\mathcal{O}(\log k)$ time using the tree-based algorithm for the k resulting values. Thus, the time complexity is $\mathcal{O}(\frac{n}{p} + \log p)$. If the data size is sufficiently larger than the number of cores, the time complexity becomes $\mathcal{O}(\frac{n}{p})$. As is the case with the cascading algorithm, the I/O complexity is $\mathcal{O}\left(\frac{n}{b}\right)$. The amount of the shared memory used is $\mathcal{O}(b)$ words and the amount of the global memory used is $n + \mathcal{O}(p/b)$ words. Finally, the multiplicity is M/b. Table 1 shows the I/O and time complexities and multiplicity of these algorithms. Algorithm "Matrix-based" will be explained in the next section.

5.2.1 Importance of multithreading

If the data size is sufficiently larger than the number of cores, the pipeline algorithm has the optimal time and I/O complexities and the multiplicity is considered to be optimal in NVIDIA GPUs. If we do not care the value of multiplicity, we can easily develop an algorithm that has the optimal time and I/O complexities by modifying the cascading algorithm. We first read b^2 elements to a matrix with b columns and b rows, then transpose the matrix. We can avoid bank conflicts by using the same method as the algorithm that we will describe in Section 6.1. We call this algorithm "matrix-based". While the time and I/O complexities of this algorithm are optimal, the multiplicity

Algorithm 6 CalculateReductionUsingTreeBased2(T, n)

Input: An array $T[0..n-1]$
Output: The reduction value of $T[0..n-1]$

```
1:  $Q := \&T[1]$
2:  $\Omega := \&W[1]$ {Reduction values so far}
3:  $d \leftarrow n$
4:  while $(d > 1)$ do
5:    $s \leftarrow \lceil d/2kb \rceil$ {The number of serialization}
6:    for $(j = 0; j < s; j++)$ do
7:      for all $\rho \in MP[0..k-1]$ in parallel do
8:        for all $\epsilon \in Core[0..b-1]$ in parallel do
9:          $x[\epsilon] \Leftarrow Q[b(jk+\rho)+\epsilon]$ {Each multiprocessor reads $b$ elements}
10:         for $(j = 0; j < \log b; j++)$ do
11:           if $(\epsilon < b/2^{j+1})$ then
12:             $x[\epsilon] \leftarrow x[2\epsilon] \oplus x[2\epsilon+1]$
13:           end if
14:         end for
15:         if $(\epsilon == 0)$ then
16:           $\Omega[jk+\rho] \Leftarrow x[0]$ {Each multiprocessor writes the reduction value of the $b$ elements}
17:         end if
18:       end for
19:     end for
20:   end for
21:   $Q := \Omega$ {The same calculation is repeatedly done to the resulting values}
22:   $d \leftarrow \lceil d/2b \rceil$ {The number of elements is reduced to $1/2b$}
23:   $\Omega := \&\Omega[d]$
24: end while
25: return $Q[0]$
```

is smaller than the other algorithms. In the next section, we will experimentally show the running time of reduction algorithms.

5.3 Experimental Evaluation

5.3.1 Experimental Comparisons of Running Time

We have implemented the reduction algorithms using CUDA and have measured their running time using NVIDIA k20c GPU. The reduction operator is the summation of integers.

Figure 9 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 value is $208GB/s$.

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.

Table 1: Complexities and multiplicity of reduction algorithms on AGPU(p, b, M). Here n is the number of elements to be reduced. We assume $p = o(n)$. Pipeline and Matrix-based can be used with non-commutative operator, whereas the others cannot.

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

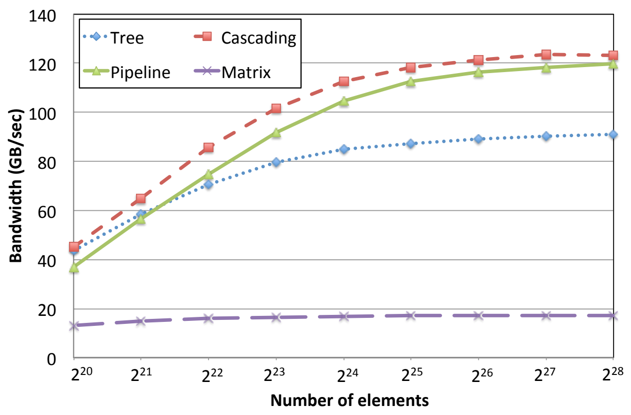

Figure 9: The comparison of processing time of several reduction algorithms.

6 Prefix Scan Algorithms

In this section, we deal with a prefix scan (prefix sum) algorithm. It aims to show that we can predict the performance of GPU-based algorithms using the AGPU model. After we analyze the asymptotic behavior of the algorithm, we measure the actual running time of the algorithm with various parameter values.

Prefix scan 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:

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

where the operator $\oplus$ is associative and commutative and $I_{\oplus}$ is the identity element for the operator. This definition is also known as exclusive scan or prescan.

Dotsenko et al. [15] proposed a matrix-based algorithm for prefix scan on GPUs. Input data 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 [16], which is one of the standard CUDA libraries, uses a similar algorithm for prefix scan. We analyze the prefix scan algorithm in Thrust.

6.1 Description and Analysis of Prefix Scan Algorithms

A pseudo-code for the prefix scan algorithm used in Thrust is shown in Algorithm 7. The code is slightly modified to make the algorithm suitable for the AGPU model. In addition, for simplicity,

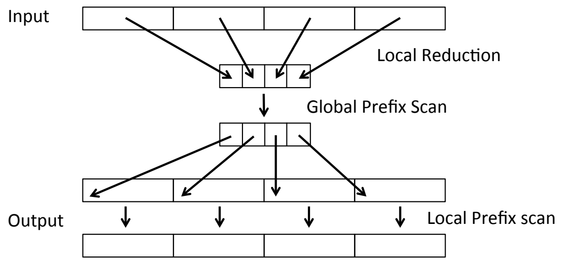

Figure 10: The outline of prefix scan algorithm

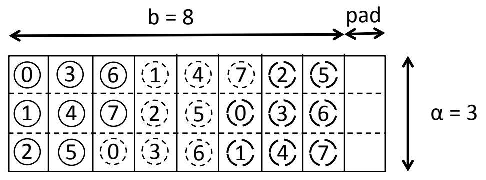

Figure 11: An example of the data alignment

we omit the process of rounding of fractions. In the pseudo-code, parameter k represents the total number of multiprocessors, namely, $k = p/b$. Algorithms 8 is a subroutine invoked by Algorithm 7.

Figure 10 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 3. 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 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 scan.

Algorithm 8 shows 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 Algorithm 8, 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 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 11 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 $\alpha = b$ elements are stored in distinct banks. Next, each core in a multiprocessor calculates the reduction of the α elements in one column in parallel. Then, 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

Table 2: 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	$\mathcal{O}((\frac{n}{p} + \frac{p}{b^2})(\min\{\lceil\frac{b}{\alpha}\rceil, \alpha\} + \frac{\log b}{\alpha}))$	$\frac{3n}{b} + \mathcal{O}(\frac{p}{b})$	$\mathcal{O}(\frac{M}{\alpha b})$

Algorithm 7 CalculatePrefixScan(T, U, n)

Input: An input array $T[0..n-1]$
Output: The prefix scan $U[0..n-1]$ of $T[0..n-1]$

```
// Local reduction
$C[0..p/b-1]$ = CalculateLocalReductionUsingCascading($T, n$)

// Global prefix scan
for all $\rho \in MP[0..0]$ in parallel do
    for all $\epsilon \in Core[0..b-1]$ in parallel do
        CalculateBlockPrefixScan($C, C, k, NULL$)
    end for
end for

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

$\mathcal{O}(n/p+\log b)$ time to calculate the local reduction because each core calculates n/p rows and it takes $\mathcal{O}(\log b)$ 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 $\min\{\lceil\frac{b}{\alpha}\rceil, \alpha\}$ 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 $\mathcal{O}(\min\{\lceil\frac{b}{\alpha}\rceil, \alpha\} \cdot \alpha)$. Writing the resulting values of a matrix to global memory has the same time complexity. In addition, it takes $\mathcal{O}(\alpha + \log b)$ 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 is $\mathcal{O}(\min\{\lceil\frac{b}{\alpha}\rceil, \alpha\} \cdot \alpha + \log b)$ 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 $\frac{n}{k(\alpha b)}$ times. Therefore, the total time complexity is $\mathcal{O}((\frac{n}{p} + \frac{p}{b^2})(\min\{\lceil\frac{b}{\alpha}\rceil, \alpha\} + \frac{\log b}{\alpha}))$.

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

Table 2 shows the I/O and time complexities and multiplicity of the matrix-based algorithm.

Algorithm 8 CalculateBlockPrefixScan(T, U, n, C)

Input: An input array $T[0..n-1]$, a carry C
Output: The prefix scan $U[0..n-1]$ of $T[0..n-1]$

```
Allocate a matrix $m[\alpha][b+1]$ in the shared memory.

// Each multiprocessor sequentially calculates the prefix scan of the matrix with $\alpha b$ elements.
for $(i = 0; i < n; i++)$ do
    // Each multiprocessor reads all elements in a matrix and arrange them in column-major order.
    for $(j = 0; j < \alpha; j++)$ do
        $data\_tmp[\epsilon] \Leftarrow T[\alpha bi + jb + \epsilon]$
        $m[(jb+\epsilon)\%\alpha][(jb+\epsilon)/\alpha] \leftarrow data\_tmp[\epsilon]$
    end for

    // Each core calculates the reduction of one column.
    $carry \Leftarrow C$
    if $((\epsilon == 0)\&\&(C\ != NULL))$ then
        $val\_column[\epsilon] \leftarrow C$
    else
        $val\_column[\epsilon] \leftarrow I_\oplus$
    end if
    for $(j = 0; j < \alpha; j++)$ do
        $val\_column[\epsilon] \leftarrow val\_column[\epsilon] \oplus m[j][\epsilon]$
    end for

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

    // Each core calculates the prefix scan of one column.
    if $((\epsilon == 0)\&\&(C\ != NULL))$ then
        $next \leftarrow C \oplus m[0][\epsilon]$
        $m[0][\epsilon] \leftarrow C$
    else
        $next \leftarrow m[0][\epsilon]$
        $m[0][\epsilon] \leftarrow val\_column[p-1]$
    end if
    for $(j = 1; j < \alpha; j++)$ do
        $tmp \leftarrow m[j][\epsilon]$
        $m[j][\epsilon] \leftarrow next$
        $next \leftarrow next \oplus tmp$
    end for

    // Each multiprocessor writes all words in a sub-block to global memory.
    for $(j = 0; j < a; j++)$ do
        $data\_tmp[\epsilon] \leftarrow m[(jb+\epsilon)/a]$
        $U[abi + jb + \epsilon] \Leftarrow data\_tmp[\epsilon]$
    end for
end for
```

6.2 Discussion

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 $\mathcal{O}((\frac{n}{p} + \frac{p}{b^2})(\log b))$. When $\alpha = \Omega(b)$, the time complexity is $\mathcal{O}(\frac{n}{p} + \frac{p}{b^2})$. 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 $\mathcal{O}(b)$. On the other hand, When $\alpha = \Omega(b)$, the multiplicity is not considered to be optimal since the amount of shared memory used is $\mathcal{O}(b^2)$. 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 $\mathcal{O}(\frac{n}{p})$ and $\mathcal{O}(\frac{n}{b})$, respectively, which are asymptotically optimal. However, if the input size n is not sufficiently larger than p, the term $\frac{p}{b^2}$ 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 $\mathcal{O}(\frac{n}{p}(\min(\lceil b/\alpha \rceil, \alpha) + \frac{\log b}{\alpha}) + \log p)$ by using the algorithm proposed by Harris et al. [17].

6.3 Experimental Results

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 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 12 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 $n \times 2 \times \mathit{sizeof}(\mathit{int})/t$ 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, 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. The line of the C1060 shows the same tendency. The multiplicity has the maximum value at $\alpha < 8$. We can conclude that the AGPU model can explain the behavior of the algorithm, which is affected by the effect of multithreading and the time complexity.

7 Comparison Sorting Algorithms

We design an effective comparison-based sorting algorithm on the AGPU model. Since sorting is one of the most fundamental operations used for many applications, it is useful to speed it up. A lot of GPU-based sorting algorithms have been proposed [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28]. We start with analyzing the complexities of GPU-Warpsort [25] on the AGPU model. The I/O complexity of it, which will be discussed later, is not optimal. We therefore propose a new algorithm with the optimal I/O complexity. Table 3 shows the I/O and time complexities of these algorithms.

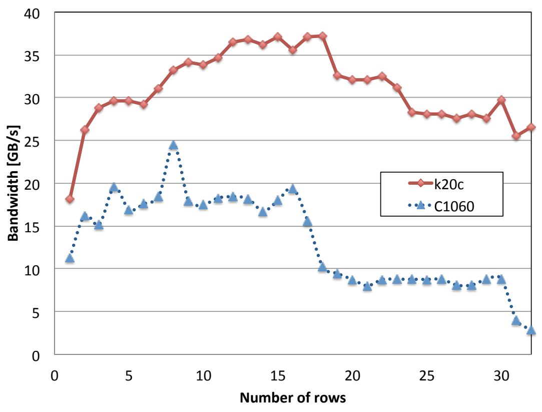

Figure 12: Bandwidth of the prefix scan algorithm with varying number of row in the matrix

Table 3: 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)	$\Omega(\frac{n}{b}\log_{\frac{M}{b}}\frac{n}{b})$	$\Omega(\frac{n}{p}\log n)$
Bitonic sort [29]	$\mathcal{O}(\frac{n}{b}\log^2\frac{n}{M})$	$\mathcal{O}(\frac{n}{p}\log^2 n)$
GPU-Warpsort [25]	$\mathcal{O}(\frac{n}{b}\log\frac{n}{b})$	$\mathcal{O}(\frac{n}{p}\log\frac{n}{b}\log b)$
Our algorithm	$\mathcal{O}(\frac{n}{b}\log_{\frac{M}{b}}\frac{n}{b})$	$\mathcal{O}(\frac{n}{p}\log\frac{n}{b}\log b)$

7.1 Analyses of Known Parallel Sorting Algorithms

7.1.1 Bitonic Sort

Bitonic sort [29] is a parallel sorting algorithm based on a sorting network, and it can sort n numbers using an $\mathcal{O}(\log^2 n)$ level network.

Figure 13 shows an example of bitonic sort networks. The bitonic sort network for $n = 2^d$ elements consists of $d = \log n$ phases, and phase i ($0 \le i < d$) consists of $i + 1$ stages. Phase i is given 2^{d-i} sorted sequences of length 2^i each, and outputs 2^{d-i-1} sorted sequences of length 2^{i+1} each. Parallel sorting algorithms based on sorting networks are suitable for implementations using GPUs because of absence of conditional branches.

On the p processor PRAM model, the running time of a bitonic sort algorithm for $n = 2^d$

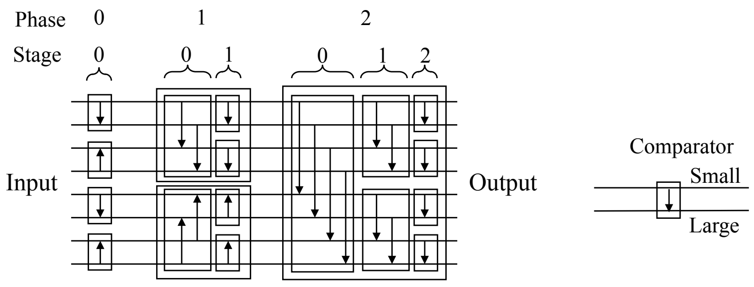

Figure 13: The bitonic sort network for 8 elements

elements is $\mathcal{O}(\frac{n}{p}\log^2 n)$. In this section, we analyze the I/O and time complexities of this algorithm when it is executed on the AGPU(p, b, M) model.

First we analyze the I/O complexity. The output of phase i $(0 \le i < d)$ is 2^{d-i-1} sorted sequences of length 2^{i+1} each. Because each multiprocessor can store M values, computations in phases 0 to $\log M - 1$ are done without communication between multiprocessors. Therefore the I/O complexity in the phases is $\mathcal{O}(\frac{n}{b})$ in total. With respect to phases $\log M$ to $\log n$, the algorithm does not require any I/O 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, we obtain that the I/O complexity of the bitonic sort is $\mathcal{O}(\frac{n}{b}(\log n - \log M)^2)$.

It is easy to show that the time complexity of the bitonic sort on the AGPU(p, b, M) model is equal to that in PRAM model, which is $\mathcal{O}(\frac{n}{p}\log^2 n)$. We therefore obtain the following.

Theorem 7.1 *The bitonic sort algorithm for n elements on the AGPU(p, b, M) model has I/O complexity $\mathcal{O}\left(\frac{n}{b}\log^2\frac{n}{M}\right)$ and time complexity $\mathcal{O}(\frac{n}{p}\log^2 n)$.*

7.1.2 GPU-Warpsort

The bitonic sort for n elements consists of $\mathcal{O}(\log^2 n)$ stages. It is therefore inefficient if n is large. GPU-Warpsort [25] is a sort algorithm that combines the bitonic sort and the merge sort to improve the inefficiency of the bitonic sort.

Consider merging two sorted sequences A, B, each of which is of length $n/2$, using b cores. The GPU-Warpsort first reads the first b elements of both sequences in the global memory into a shared memory. Let $a_{\max}$ and $b_{\max}$ denote the maximum values read from the sequences A and B, respectively. The GPU-Warpsort sorts those $2b$ elements using the bitonic sort. However, because those elements are from two sorted sequences, it is enough to perform the last phase of the bitonic sort. After sorting the $2b$ elements, the algorithm outputs the smallest b elements, and reads b new elements from either sequence A or B. If $a_{\max} \le b_{\max}$, the b new elements are taken from sequence A, and otherwise from sequence B. The algorithm repeats this until the sequences A and B become empty. All global memory accesses in this algorithm are done efficiently because these are done in units of b elements. Therefore we obtain the following.

Lemma 7.2 *On the AGPU(p, b) model, merging of two sorted sequences of length $n/2$ each is done with I/O complexity $2n/b + \mathcal{O}(1)$ and time complexity $\mathcal{O}(\frac{n}{b}\log b)$. This algorithm uses one multiprocessor and $\mathcal{O}(b)$ words of shared memory.*

The GPU-Warpsort consists of four steps.

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 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 using the elements as separators.
4. Merge the sequences divided in the step 3.

The step 2 is inefficient if the number of sorted sequences becomes smaller than the number of multiprocessors. Therefore the GPU-Warpsort changes the algorithm to utilize them.

The I/O and time complexities of the GPU-Warpsort are analyzed as follows. Step 1 has I/O complexity $2n/b + \mathcal{O}(1)$ and time complexity $\mathcal{O}(\frac{n}{p}\log^2 b)$. Step 2 has I/O complexity $\mathcal{O}(\frac{n}{b}\log\frac{n}{b})$ and time complexity $\mathcal{O}(\frac{n}{p}\log b \log\frac{n}{b})$. We can implement the algorithm such that the time and I/O complexities of Steps 3 and 4 are dominated by those of Steps 1 and 2. Supposing $n = \Omega(b^2)$, the time complexity of Steps 1 is dominated by that of Steps 2. Thus, we obtain the following theorem.

Theorem 7.3 *Supposing $n = \Omega(b^2)$, GPU-Warpsort for n elements runs on the AGPU(p, b, M) model with I/O complexity $\mathcal{O}(\frac{n}{b}\log\frac{n}{b})$ and time complexity $\mathcal{O}(\frac{n}{p}\log\frac{n}{b}\log b)$.*

Because a trivial lower bound of the time complexity for comparison-based sorting is $\Omega(\frac{n}{p}\log n)$, the time complexity of the GPU-Warpsort is at most $\mathcal{O}(\log b)$ times of the optimal algorithm. However, the I/O complexity is $\mathcal{O}(\log\frac{M}{b})$ times larger than the optimal, as shown in Section 7.2.

7.2 A Sorting Lower Bound in 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 lower bound on time complexity for AGPU(p,b,M) is $\Omega\left(\frac{n}{p}\log n\right)$.

With respect to the I/O complexity, the lower bound for I/O(b,M) [13] is known as follows.

Theorem 7.4 (Aggarwal, Vitter [13]) *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 4.5 holds, the lower bound on the I/O complexity for the volatile model AGPU(p,b,M) is as follows.

Theorem 7.5 *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 4.5 does not hold on non-volatile model AGPU$'$, we can obtain the same lower bound as follows.

Theorem 7.6 *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 $\mathcal{O}(n/b)$ 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 $\mathcal{O}(n/b)$ 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 $\frac{n!}{(b!)^{n/b}}$. 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 $\frac{n!}{(b!)^{n/b}}$ different inputs is

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

□

7.3 I/O-optimal Sorting Algorithms on AGPU Model

In this section we propose a comparison-based sorting algorithm on the AGPU(p,b,M) model whose I/O complexity is asymptotically optimal. Aggarwal's algorithm on the I/O model [13] is I/O-optimal but does 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 (see Lemma 7.2) so that a multiprocessor with b cores merges more than two sorted sequences at a time. The GPU-Warpsort merges two sorted sequences at a time, and all elements are read from and written to global memory every time. By merging $d > 2$ sorted sequences at a time, we can reduce the number of global memory accesses. Figure 14 shows an example of merging eight sequences at a time. The GPU-Warpsort accesses each element in the global memory six times, while our algorithm accesses each element twice.

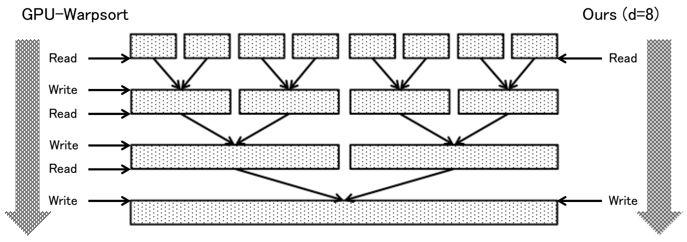

Figure 14: Global memory accesses for merging eight sorted sequences. Let the number of sequences merged by our algorithms at a time be eight. In the GPU-Warpsort, the number of the global memory accesses for each element is six, whereas the number is two in our algorithm.

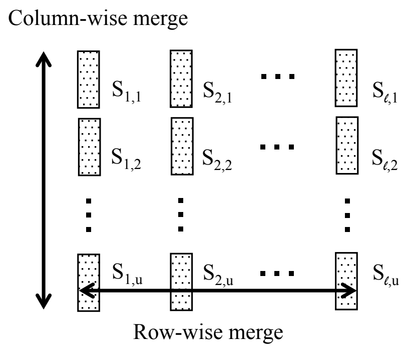

Figure 15: Column-wise and Row-wise merge

7.3.1 Overview of the Algorithm

Our algorithm consists of the following four parts; 1. Initialize; 2. Column-wise merge; 3. Subarray partition; 4. Row-wise merge.

In Part 1, our algorithm partitions the input sequence into pieces of b elements. We call each piece a *basic block*. Then we sort each basic block by using the Bitonic sort. We use all the $k = p/b$ multiprocessors, each of which performs the bitonic sort for b elements. The I/O and the time complexities are $\frac{2n}{b} + \mathcal{O}(1)$ and $\mathcal{O}(\frac{n}{bk}\log^2 b) = \mathcal{O}(\frac{n}{p}\log^2 b)$, respectively. We call the output sequences *subarrays*.

In Part 2, we repeatedly merge subarrays until the number of those is equal to or smaller than a threshold ℓ, which will be determined later.

In Part 3, we pick up some separators from each subarrays at regular intervals and merge all separators. We partition subarrays based on the separators.

Finally in Part 4, we merge partitioned subarrays, which store values between two separators, for each pair of consecutive separators. This prevents any multiprocessor being idle.

7.3.2 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 16 shows an example for the case $d = 4$.

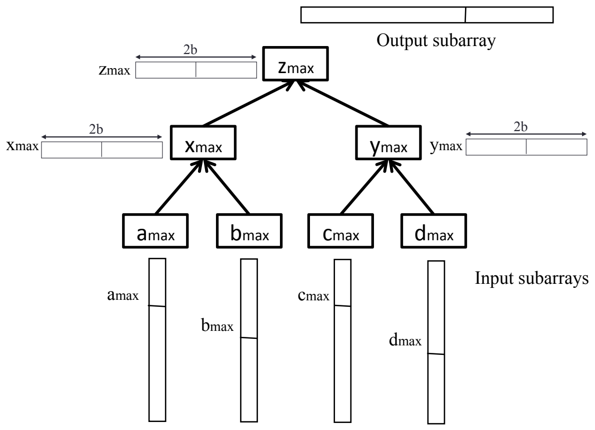

Figure 16: A heap used in merge process

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 i has index $2i$ and $2i+1$, respectively. The function Heapify(i) is the process to move b elements to the buffer in node i from the buffers in the descendent nodes. The function is only invoked when the number of elements in the buffer of node i is at most b. For simplicity, we assume the number of elements in the buffer is exactly b. If the number of elements in a input subarray is not a multiple of b, the number of elements in the buffer can be smaller than b. In this case, we can easily modify our algorithm. 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 b. Then the buffer of node i consists of two sorted sequences; one is already stored in node i before the move, and the other is newly moved to node i. 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 using the Heapify operation. First, we allocate the shared memory to this structure and repeat conducting the Heapify operations 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, and conduct 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. We define the rank of an element as the number of elements smaller than or equal to the element in the subtree. For instance, the rank of the smallest element in the subtree is one. Now we prove the ranks of any elements in the descendent nodes are larger than b after the Heapify operation. First we consider an internal node whose children are leaves. Let $\alpha_{\max}$ and $\beta_{\max}$ be the keys of the left and the right child of a node i for which the Heapify operation is done. We assume that $\alpha_{\max} < \beta_{\max}$; the other case is done analogously. When we conduct the Heapify operation to node i, the buffer of node i has b elements, and the smallest b elements in the left child are newly moved to the buffer of node i. Therefore, any elements in the left child buffer have ranks larger than

b after the operation. Before conducting the Heapify operation, the value of the last element (that is, the largest value) in the buffer is $\beta_{\max}$. Therefore, the rank of this element is equal to or larger than b. Therefore, any elements in the right child buffer have ranks larger than b. We can prove it at any nodes recursively.

Note that we can reduce the size of buffers in the nodes 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 7.7 *Merging d sorted sequences of length $\frac{n}{d}$ each is done with I/O complexity $\frac{2n}{b} + \mathcal{O}(d)$ and time complexity $\mathcal{O}\left(\left(\frac{n}{b}\log d + d\right)\log b\right)$ on the $AGPU(b, b, \mathcal{O}(db))$ model.*

Accordingly, if a multiprocessor has M word shared memory, our algorithm can merge up to $d = \mathcal{O}(\frac{M}{b})$ sorted sequences. We concurrently carry out this process to all subarrays using k multiprocessor. We call this process a *column-wise merge step.* In part 2, we repeat it s_0 times, where s_0 is a parameter determined later.

7.3.3 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 7.7. Let $p_1 \le p_2 \le \ldots \le p_{\rho\ell}$ 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 $(1 \le j \le u)$.

Let group G_j $(1 \le j \le u)$ 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 $|G_j|$ denote the size.

We can use the algorithm of Lemma 7.2 to divide subarrays. After merging a subarray and separators, we calculate the position of the separators in the resulting sequence.

7.3.4 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 7.8 *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 $\frac{u-1}{\ell}$ separators from each subarray of n/ℓ elements, the size of a divided subarray is at most $\frac{n}{\ell} \cdot \frac{1}{\left(\frac{u-1}{\ell}+1\right)} < \frac{n}{u-1}$. Since each group consists of ℓ subarrays, the size of a group is at most $\frac{n\ell}{u-1} < \frac{n}{k}$. 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 $\frac{n}{k} \cdot k = n$. 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 7.7 and get $\lceil \ell/d \rceil$ subarrays. We call this process a row-wise merge step. A multiprocessor repeatedly executes the 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 $(1 \le t)$ 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 $\mathcal{O}\left(\left\lceil \left|S^{t+1}_{i,j}\right| / b \right\rceil \log b \log d\right)$. Supposing C_x is a set of indices of groups that are assigned to multiprocessor x, the total time complexity at step t is

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

Due to Lemma 7.8, for any multiprocessor x, $\sum_{j \in C_x} \sum_{i=1}^{\lceil v/d \rceil} \left|S^{t+1}_{i,j}\right| < 2n/k$. Therefore, the time complexity at step t is

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

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

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

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

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

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

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

7.3.5 The Complexities and the Amount of Memory Used for the Entire Process

We calculate the time and I/O complexities by summing up those of all parts. It holds $\ell = \mathcal{O}\left(\frac{n}{bd^{s_0+1}}\right)$, and $t_0 = \log_d \frac{n}{b} - s_0$ where s_0 is the number of steps in the column-wise merging part. Supposing $n = \Omega\left(b^2\right)$, the time complexity for the entire process is $\mathcal{O}\left(\frac{n}{p} \log b \log \frac{n}{b} + \log b \log d \left(\frac{n}{b\ell} + \frac{u\ell}{k} + \frac{u}{b}\right)\right)$, and the I/O complexity for the entire process is $\mathcal{O}\left(\frac{n}{b} \log_d \frac{n}{b} + u\ell\right)$. The amount of the shared memory used is $\mathcal{O}(bd)$ words, and the amount of the global memory used is $2n + \mathcal{O}(u\ell)$ words.

We determine the values of ℓ and u as $\ell = k$, and $u = \frac{n}{p}$ 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 $\mathcal{O}(db)$ words of shared memory, we determine the value of d as $d = \mathcal{O}(M/b)$.

Taken together, we obtain the following theorem.

Theorem 7.9 *Supposing $n = \Omega(b^2)$, the time complexity of our algorithm is*

$$\mathcal{O}\left(\frac{n}{p} \log b \log \frac{n}{b}\right),$$

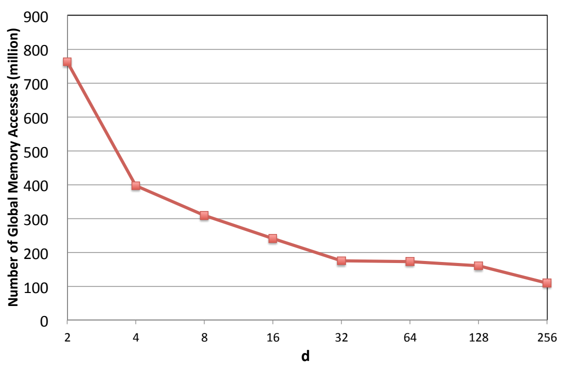

Figure 17: The number of the global memory accesses for each value of d

and the I/O complexity of our algorithm is

$$\mathcal{O}\left(\frac{n}{b}\log_{\frac{M}{b}}\frac{n}{b}\right).$$

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

7.3.6 Effect of Multiplicity

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

7.4 Evaluation

7.4.1 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 bits integers. Figure 17 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 18 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 $d = 4$.

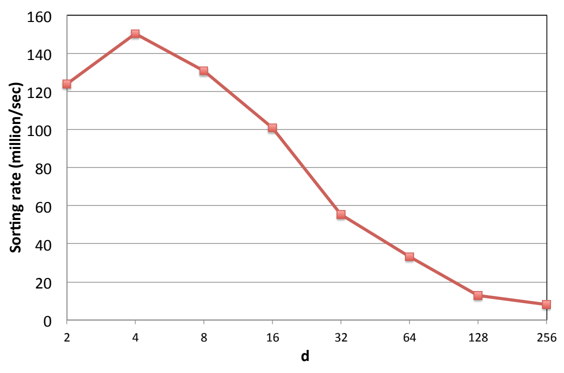

Figure 18: Sorting rate for each value of d

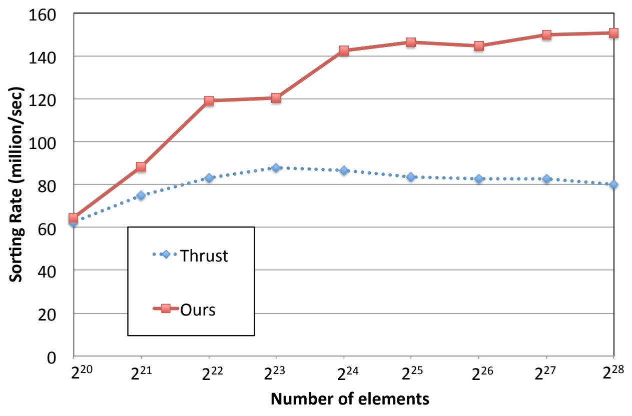

Figure 19: Sorting rate for our algorithm and Thrust comparison-based sorting

7.4.2 Comparison with Thrust

We compared our algorithm with Thrust comparison-based sorting. Figure 19 shows the sorting rate. Our algorithm was 1.9 times faster than Thrust when $n = 2^{28}$.

Figure 20 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.

8 Concluding Remarks

We have proposed AGPU model, a computational model for analyzing complexities of GPU-based algorithms. We can design algorithms to take advantage of GPU architectures and analyze asymptotic computational complexities of the algorithms using the AGPU model. As an example of analyses using the AGPU model, we analyzed several basic algorithms including reduction, prefix scan, and comparison sorting. The AGPU model can explain the behavior of the algorithms. In particular, it brings out the bottlenecks of the algorithms. It is useful to improve the algorithms. In fact, we have developed some novel algorithms by removing the bottlenecks. Our algorithms are faster than existing algorithms not only in theory, but also in practice.

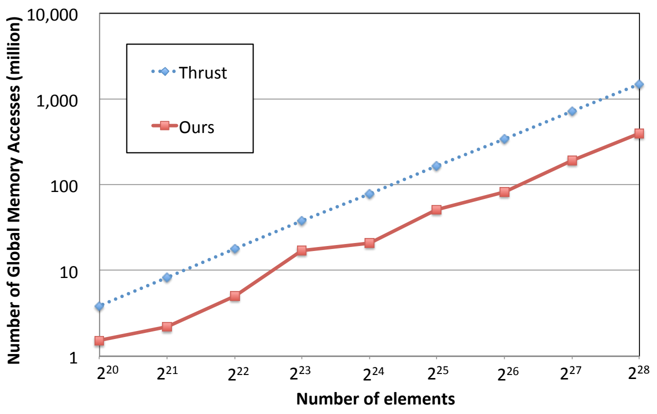

Figure 20: The number of global memory accesses for our algorithm and Thrust

8.1 Guideline for Writing Efficient Algorithms Using 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 4.3, 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 4.1).

8.2 Future Work

In the future, we would like to analyze and develop more basic algorithms using the AGPU model. Firstly, we would like to improve our sorting algorithm. Our current algorithm has a tradeoff between the I/O complexity and the multiplicity. We would like to remove the tradeoff. Moreover, we wold like to tackle GPU-based non-comparison sorting. Secondly, we would like to develop new algorithms for graphs. In graph algorithms, it is difficult to avoid bank conflicts and to make global accesses coalesce. We would like to analyze graph algorithms and find the bottlenecks. In particular, we would like to develop algorithms for trees. Thirdly, we would like to study algorithms that utilize multiple GPU devices. We will improve the AGPU model in order to analyze this kind of algorithms, and develop the new algorithms.

Acknowledgment

Work supported in part by KAKENHI 23240002.

References

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

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

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

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Elahe Khorasani, Brent D. Paulovicks, Vadim Sheinin, and Hangu Yeo. Parallel implementation of external sort and join operations on a multi-core network-optimized system on a chip. In *Proceedings of the 11th international conference on Algorithms and architectures for parallel processing - Volume Part I*, ICA3PP'11, pages 318–325, Berlin, Heidelberg, 2011. Springer-Verlag.

[22] 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.

[23] 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.

[24] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast in-place sorting with cuda based on bitonic sort. In *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, PPAM'09, pages 403–410, Berlin, Heidelberg, 2010. Springer-Verlag.

[25] 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.

[26] G. Capannini, F. Silvestri, R. Baraglia, and F.M. Nardini. Sorting using bitonic network with cuda. In *Proceedings of the 7th Workshop on LSDS-IR*, 2009.

[27] 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.

[28] 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.

[29] 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.