Research on Performance Optimization Methods based on Performance Analytical Modeling and Communication Latency Hiding in GPU

(GPUの性能分析モデリングと通信レイテンシの隠蔽に基づく性能の最適化方法に関する研究)

Doctoral Dissertation

博士論文

Cheng Luo

羅成

Submitted to Department of Computer Science,
Graduate School of Information Science and Technology,
The University of Tokyo
in partial fulfillment of the requirements for the degree of
Doctor of Information Science and Technology

Thesis Supervisor: Reiji Suda (須田礼仁)
Professor of Computer Science

June 14, 2013

Abstract: Stream processing has been widely used since the emergence of stream applications such as time varying visualization and audio/video processing. Stream processing can exploit the inherent parallelism of the pipeline while the different stream elements also can be processed simultaneously to achieve data parallelism. Graphic processing unit(GPU) is one of the most successful stream architectures in recent years which is originally designed for acceleration of graphics applications. Now, it is widely used as General-purpose computing on graphics processing units (GPGPU) to accelerate many scientific applications with more than 10 times speedup over CPU platform. There are many new programming languages that help programmers to write parallel applications with GPUs such as Brook+, CUDA and OpenCL. With these programming and architectural features, programmers can quickly port their programs to a GPU based platform. However, if programmers want to have a better performance, they need to have a further understanding at various features of the low-level architecture and associated bottlenecks in their applications which will increase their burden in writing parallel applications. Therefore, there are many researches working on performance optimization methods from many aspects for programmers without much knowledge of GPU.

The motivation behind this work was caused by the emerging of high computational potential GPU along with the difficulty of writing high performance parallel programs on GPU based system. Our interests focus on performance prediction problem and communication latency between the host and the device problem. For performance prediction problem, it is difficult to predict the performance of kernel codes on GPU without enough knowledge about the low level architecture. Therefore, programmers may use unsuitable configuration to run their applications on GPUs which may lead to poor performance. Therefore, performance analytical model is needed to help programmers better understand the performance of their applications on GPU and find out the performance bottlenecks.

On the other hand, the communication latency between the host and device also can greatly affect the performance. CUDA programs include two parts: host code running on CPU and device code running on GPU. The host code invokes the device code to execute the kernel operation while the input and output streams are stored in device memory. As the device memory is separated from the host memory, streams are required to be transferred between them which leads to the communication latency between the host and device. According to different application types, the communication latency overhead between the host and device will account from very little to very high proportion of the total execution time cost. It is difficult for programmers to achieve high performance without awareness of the communication latency. CUDA supports concurrent execution for kernel execution and data transfer. Notice that some latest NVIDIA Tesla series GPUs begin to support two copy engines for bi-directional data transfer which enables to launch data send, kernel execution and data receive simultaneously. With this new feature, it is possible to use three streams respectively for data send, kernel execution and data receive to hide the communication latency by overlapping the three streams.

In this thesis I am proposing performance optimization methods based on performance analytical modeling and communication latency hiding to solve the performance prediction problem and communication latency problem respectively. For performance prediction problem, I propose a performance analytical model which can help programmers have a better insight into their applications and give a better configuration to execute application based on the predicted results. For communication latency problem, I propose a task partitioning and scheduling method named TPSM to help programmers achieve to hide the communication latency between the host and the device in GPU based system.

The performance analytical model can estimate the execution time of massively parallel programs which take the instruction-level and thread-level parallelism into consideration. The model contains two components: memory submodel and computation submodel. The memory submodel can estimate the cost of memory instructions by considering the number of active threads and GPU memory bandwidth. Correspondingly, the computation submodel can estimate the cost of computation instructions by considering the number of active threads and application's arithmetic intensity. I use ocelot to analyze PTX codes to obtain several input parameters for the two submodels such as memory transaction number and data size. Based on the two submodels, the analytical model can estimate the cost of each instruction while considering instruction-level and thread-level parallelism, thereby estimating the overall execution time of an application. With the predicted results, programmers can choose a suitable configuration to execute their applications with better performance.

I also propose a Task Partition and Scheduling Method(TPSM) which can help programmers to partition individual GPU application into subtasks and improve the performance of individual application with three streams by overlapping data send, kernel execution and data receive. With two copy engines, the work support simultaneous data send, kernel execution and data receive while previous work can only support simultaneous unidirectional data transfer and kernel execution. To extract the features of application, I classify GPU applications into several basic types from computation-to-communication ratio aspect and send-to-receive ratio aspect.

With the classification, I design corresponding task partitioning and scheduling submethods. I also design a time optimal data transfer algorithm to achieve optimal data transfer between host and device in multiple GPU architecture. TPSM can be applicable in single GPU architecture, multiple GPU symmetric architecture and multiple GPU non-symmetric architecture.

I use four benchmarks to test the performance analytical model and tasking partitioning and scheduling method on various GPUs. The results show that the performance analytical model can achieve on average around 90% accurate ratio for the prediction of kernel execution. The results of TPSM show that the work proposed in this thesis can successfully hide the communication latency between for individual application to achieve high performance which is very close to the lower bound time cost.

Keywords: Performance, prediction, model, task partitioning, task scheduling, overlapping computation and communication, parallelism, GPU

Acknowledgments

I would like to express gratitude to all those people who helped me during the writing of this thesis.

I owe my deepest gratitude to my supervisor, professor Reiji Suda, for guiding me into this inspiring and challenging area in computer science. I have been extremely lucky to have a supervisor who cared so much about my work, my life in Japan and who discussed with me and responded to my questions every week. Thank him so much for all his support and encouragement over these past three years.

I also would like to thank all members in Suda lab for provide the support for everything.

I thank my parents for supporting and understanding me all the time.

I must express my gratitude to Ting, my wife, for her continued support, encouragement, quiet patience unwavering love which in the end makes this dissertation possible.

Many thanks to Sunpyo Hong for applying information about GPU emulator and benchmarks.

Thanks to Michael Linderman for applying Merge benchmarks.

Thanks to the patient technological replies from the gpuocelot groups.

This work was partially supported by Core Research of Evolutional Science and Technology (CREST) project "ULP-HPC: Ultra Low-Power, High-Performance Computing via Modeling and Optimization of Next Generation HPC Technologies" of Japan Science and Technology Agency (JST) and Grant-in-Aid for Scientific Research of MEXT Japan.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Contents

1.1 Parallel Computing from CPU to CPU+GPU

Gordon Moor once predicted that the number of transistors placed on an integrated circuit will double every 18 to 24 months[85]. Until several years ago, Moor's law translated to increase in clock speed and performance which enable the same sequential program automatically gain performance just by using a faster hardware. The performance improvement for single thread is achieved by scaling up the speed and automatic exploitation of Instruction Level Parallelism (ILP).

Automatic extraction of instruction level parallelism from sequential programs is very limited[115, 116]. The scaling up of clock frequency stopped working in around 2005 because the memory speed could not catch up with the increase of processing capability. The power consumption also became very high. Therefore, people start to add more processing cores into a single microprocessor chip in order to exploit thread level parallelism to gain performance improvement.

Many desktop and notebook computers began to use dual core processors since 2006 and some high-end desktops and workstation computers began to use four core processors since 2008. Moreover, Graphic Processing Units (GPUs) which contain hundreds of processing cores are emerged. CPUs are also going into many-core period[22]. Processors in portable devices such as mobile phones also have been using multi-core processors for better power efficiency and performance.

Besides the increase of core count, people also begin to use heterogeneous cores in a single system for the demand to increase power efficiency[70]. Ultra-low Energy per Instruction (EPI) cores are very essential to scale multi-core processor to many-core processors [50]. With this consideration, computing system designers combines heterogeneous processors to optimize the systems. Currently, most of super computer adopts a heterogeneous design which includes both multi-core CPUs and GPUs as co-processing units on each node of the super computer. With this design evolution, now mainly parallel systems have many processing cores with different processing characteristics. The CPUs with simultaneous multi-threading are used for running complex operating system software and task parallelism while the GPUs with hundreds or thousands of simpler cores are used for simple throughput computation task as shown in Figure 1.1 [45].

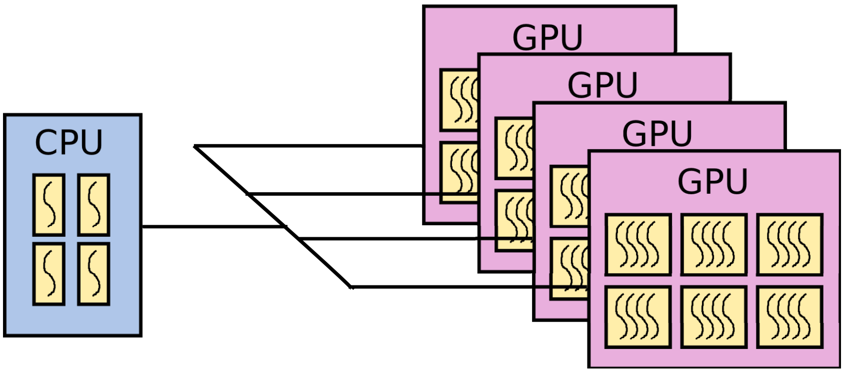


Figure 1.1: CPU + GPU heterogeneous system

The movement to massively parallel hardware leads to huge impact on software programmers. In the past, even a poorly written program can speed up with a faster hardware. With many-core hardware, however, the sequential programs can not be able to achieve better performance on parallel hardware unless the codes are properly parallelized.

Besides parallelism, programmers also need to pay attention to the characteristics of the heterogeneous processors. CPU cores are suitable for complex control flows and has limited throughput and poor power efficiency. GPU cores can work together to calculate simple computation with higher speed. Depending on different requirements on performance and energy efficiency, the cooperation between CPUs and GPUs to optimize power consumptions also is a hot research topic. Moreover, the memory access also greatly affects the performance as the hierarchical nature of memory system depends on the programs[105, 51].

The movement of parallel system from CPU based system to the system equipped both CPU and GPU and using GPU for acceleration (CPU+GPU system) brings high performance as well as new challenge.

1.2 Motivation

Graphic Processing Unit (GPU) which is originally designed for acceleration of graphic applications now is widely used as General-Purpose Computing on Graphic Processing Units (GPGPU) to accelerate many scientific applications with more than ten or hundred times speedup over CPU platform. There are many programming languages to help programmers with writing parallel applications in GPU architecture such as Brook+ [24] CUDA [37] and OpenCL [89]. With these programming and architectural characteristics, programmers can easily port their programs to a GPU based system. However, further understanding at various features of the low level architecture and associated bottlenecks in the programs are required to achieve better performance in a GPU architecture. This will require programmers pay more energy on the implementation rather than the design of the parallel applications.

The motivation behind this thesis was caused by the emergence of high computational potential from GPU along with the difficulty of implementing high performance parallel programs in GPU platform. The interests of this thesis focus on performance prediction problem and communication latency problem between CPU and GPU.

For performance predication problem, I find that sometimes it is difficult to predict the performance of kernel codes on GPU as programmers do not have enough knowledge about the low level architecture. Therefore, programmers may use a unsuitable configuration to run their applications on GPUs which may lead to poor performance.

For example, the thread configuration can greatly affect the kernel performance on GPUs. Ideally it is thought that the performance of applications can be improved with more threads. However, the performance does not always increase along with the increase of thread number. There are many factors that can affect the performance such as the process clock, the bandwidth and application features. For computing bound applications that have more computing instructions than memory access instructions, increasing the number of threads will lead to a linear increase of performance. This is because more computing resources are used to compute with more threads. However, when the number of threads continue to increase and the utilization of GPU reaches the peak performance, the performance will decrease because of running out of computing resources and extra overhead from thread launching and synchronization. In this case, the limited computing capability is the performance bottleneck. For memory access bound applications that have more memory access instructions than computing instructions, there is a similar problem.

The bandwidth of global memory is limited while the increase of thread number will lead to the increase of bandwidth requirement from applications. In this case, the performance will increase first along with the increase of thread number and then it will decrease as bandwidth is used up and frequent switch between threads leads to extra overhead.

Therefore, it is very important to have a good understanding of the application and the architecture details to achieve high performance in GPU architecture. To release programmers from this burden, performance analytical model is required to help programmers implement better parallel programs in GPU architecture.

For the communication latency problem, the communication latency can be very significant for some applications which already greatly affect the total performance. As CUDA programs include two parts, the host codes run on CPU and the device codes run on GPU. The host codes invoke the device codes to execute kernel operation while the input and output streams are stored in the device memory. As the device memory (GPU memory) is separated from the host memory (CPU main memory), streams are required to be transferred between CPU and GPU which leads to the communication latency between them. In CUDA, stream includes a sequence of sequential execution commands such as send, kernel execution or receive. For some data transfer bound applications such as black-scholes (a benchmark that we will introduce in later), the communication latency can account for half of the total time cost in single GPU platform. In multiple GPU platform, the proportion of the communication latency in the total time cost becomes larger as the limited bandwidth is shared by all GPUs. Therefore, it is difficult to achieve high performance without awareness of the communication latency between CPU and GPU.

Notice that CUDA enables concurrent execution for data transfer and kernel execution. Also some latest NVIDIA Tesla series GPUs support two copy engines for bidirectional data transfer which means it is possible to launch data send, kernel execution and data receive simultaneously. With these new features, it is possible to hide the communication latency by overlapping.

1.3 Contributions

The main research question that I aim to solve can be defined as the following questions:

- How to predict the performance of CUDA programs in GPUs?
- How to use efficiently the bandwidth between CPU and GPU in multiple GPU architecture?

- How to hide the communication latency between CPU and GPU?

In order to solve these questions, I am presenting performance optimization methods based on performance analytical modeling and communication latency hiding in GPU which includes a performance analytical model and a task partitioning and scheduling method.

To solve the first question, I propose a *performance analytical model* with instruction-level and thread-level parallelism awareness to predict the kernel execution time cost of CUDA program on GPU. I propose two definitions: CPD and MPD for the prediction of CUDA program on GPU. I introduce *Computing Parallel Degree* (CPD) to describe the parallel execution for computation instructions and to present the characteristic of applications. I also introduce *Memory Parallel Degree* (MPD) to describe the maximum number of memory accesses that can be executed concurrently. Based on the two definitions, the performance analytical model includes two submodels: memory submodel and computation submodel. The memory submodel uses MPD to estimate the time cost of memory instructions by considering the number of active threads and the GPU global memory bandwidth. The computation submodel uses CPD to estimate the time cost of computation instructions with awareness of active thread number and arithmetic intensity of applications. Based on the PTX codes generated from Ocelot[88], I develop a set of micro benchmark to test the time cost of PTX instructions. With the two submodels and time cost of PTX instructions as input, I use calculate model to estimate the time cost of overall kernel execution. I compare the predicted results with the actual execution results with four benchmarks in four kinds of GPUs, and the results show that the performance analytical model can achieve on average around 90% accurate rate.

To solve the second question, I propose a *time optimal data transfer algorithm* to achieve optimal data transfer between CPU and GPU in multiple GPU architecture (include symmetric and non-symmetric architecture). I propose a series of definitions to describe the status of GPUs and the whole system. I also introduce many notations to model the time cost of data transfer with mathematical methods. By finding the solution of the dual problem of time optimal data transfer problem, the time optimal data transfer can give out a optimal data transfer plan for a given data transfer input in multiple GPU architecture.

To solve the third question, I propose a *Task Partitioning and Scheduling Method* (TPSM) which can partition individual GPU application into subtasks and hide the communication latency between CPU and GPU by overlapping the data send part, kernel execution part and data receive part of different subtasks. Notice that it is very important to take the characteristic of application into consideration. Therefore, I propose a classification for GPU applications which classify GPU applications into six basic types from computation-to-communication ratio aspect and send-to-receive ratio aspect. For each basic type, I propose a corresponding task partitioning and scheduling sub-method based on the characteristic of the type. I use the performance analytical model in TPSM to make suitable load allocation in multiple GPU non-symmetric architecture. I also use the time optimal data transfer algorithm in TPSM to improve the utilization of bandwidth between CPU and GPU. I use four benchmarks and four type GPUs to test my work and the results show that TPSM can well hide the communication latency between CPU and GPU. The results are very close to the lower bound results (the maximum time cost of data send part, kernel execution part and data receive part).

Chapter 2

Background

Contents

In this chapter, we discuss the relevant details of the CUDA framework, the GPU architecture, GPU ocelot and the divisible load scheduling.

2.1 CUDA Framework

Nvidia developed Compute Unified Device Architecture (CUDA)[37] in order to provide a more developer friendly environment for GPU application development. CUDA looks like an extension to the C language which provides access to all of the threading, memory and functions required by developers when working with the GPU.

The GPU device provides a tremendous level of exploitable parallelism within one single chip. A standard GPU contains hundreds of processing cores and support thousands, hundreds of thousands or even millions of threads being scheduled for execution. CUDA provides a number of levels of thread organization to make the management of all threads simpler. The top level of the thread organization is the thread grid. The *thread grid* includes all threads that will execute the GPU kernel. The thread grid is split into multiple equal size blocks named *thread block*. Users specify the organization of threads within a thread block and thread blocks within a grid. Therefore, we can organize and address threads in one, two or three dimensional fashion. We can also organize and address the blocks in the thread grid in the same way. Figure 2.1 shows an example of two dimensional organization.

In the lowest level of the thread organization is the *thread warp*. The thread warps are formed by equal sized chunks of threads within the block. The size of warp is determined by the hardware specifications and the threads within one warp

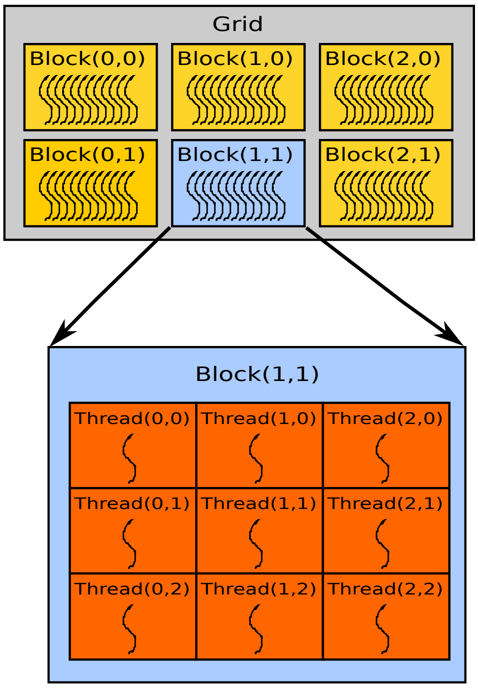


Figure 2.1: Grid of Thread Blocks

are ordered in a one dimensional fashion. For Tesla C2075, one warp includes 32 threads. GPU issues each thread within the warp the same instruction to execute. CUDA also provides some synchronization mechanisms based around the thread warp, block and grid. For the threads within one warp, they are always synchronized as they all receive the same instruction to execute. For block level synchronization, CUDA provides *__syncthreads()* instruction. All threads reaching the instruction will wait until all threads in the block arrive this point. However, CUDA does not provide any mechanisms within kernel to synchronize all threads in a grid. Therefore, we have to complete execution of the kernel and depend on the CPU to perform the synchronization. CUDA provides two kind of methods for the kernel synchronization:

- Launching another kernel: After launch one kernel, launching another kernel will lead to the application halting until the previous kernel execution finish.

- *cudaThreadSynchronize()* instruction in CPU code: This has the same results but need to be controlled by users.

2.2 GPU Architecture

Here we will introduce the GPU architecture details combined with the CUDA framework information discussed in the previous section. GPU includes two separate units: the processing cores and the off-chip memory which is connected by a proprietary and undisclosed interconnection network. One GPU device normally compose several Streaming Multiprocessors (SMs). As shown in Figure 2.2, each SM contains eight cores or more (For Nvidia Tesla K20c, each SM contains 192 cores). These cores are the computational cores of the GPU and handle the execution of instructions for the threads executing within the SM. SMs also contain a multi-threaded instruction dispatcher and some special function units which provide extra transcendental mathematic capabilities.

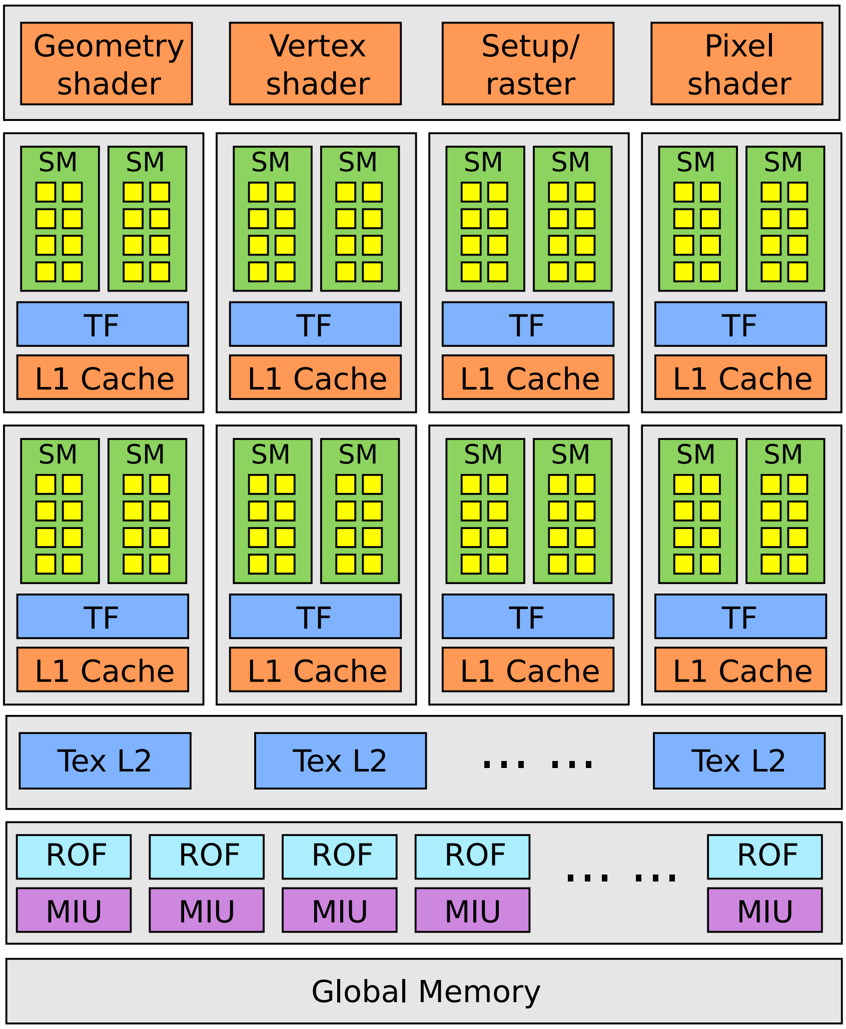


Figure 2.2: GPU Architecture

Each SM executes the instructions in a SIMD (Single Instruction Multiple Data) pattern. And all threads within a thread block must execute entirely within a single SM which means the threads in a thread block can not be split up between multiple SMs. However, multiple thread blocks can be executed on a single SM only if the SM has enough resources to support the requirements of more than one thread block.

Each SM also has small shared memory which essentially acts as a user-controlled cache for data required during kernel execution. Therefore, users are responsible to place data into this memory space. There is no any automatic hardware data caching. Nvidia changed this in their Fermi architecture and introduced a hardware-controlled cache at each SM. The accesses to shared memory are up to 100 times faster than global memory without bank conflicts. As shared memory is split into 16 32-bit wide banks, multiple data requests from the same bank at the same time will cause bank conflicts so they are serialized. Bank conflicts reduce the overall shared memory throughput. Because some threads have to wait for the data until previous threads are serviced. Shared memory is exclusive to each thread block on a SM. Therefore, one thread block can not access the shared memory data from another thread block even within the same SM.

GPU device includes global memory which is the largest memory space available on the GPU and is accessible to all threads. Global memory accesses have significant latency and are not cached at any level. Therefore, each access to global memory results in the same latency hit. Each SM has access to caches for the constant and texture memory which are still technically part of global memory. These caches help to reduce the latency by exploiting data locality.

2.3 GPU Ocelot

GPU Ocelot[88, 39] is a dynamical compilation and binary translation infrastructure for CUDA which implements CUDA runtime API and executes PTX kernels on various backend execution targets. Ocelot contains a functional simulator for offline workload characterization, correctness checking and profiling. With an additional runtime execution manager, a translator from PTX to LLVM provides efficient execution of PTX kernels on multi-core CPUs. To support Ocelot's Nvidia GPU device, PTX kernels are launched and invoked via CUDA driver API. Ocelot can inspect the state of the application as it is running, transforming PTX kernels before executed natively on GPU devices. Ocelot also manages additional resources and data structures that are needed to support instrumentation.

Ocelot replaces the CUDA runtime API library linked with CUDA applications. Ocelot enables API calls to CUDA provide a layer of compilation support, resource management and execution. CUDA kernels may be modified by Ocelot as they are registered and launched. Ocelot will insert additional state and functionality into the host applications. Therefore, it is mainly designed for transparently instrument applications and respond to data dependent applications behaviors which is not possible for static transformation techniques. For instance, Ocelot can insert instrumentation into a kernel, profile for a brief period and ten re-issue the original kernel without instrumentation to implement random sampling while the application is running.

2.4 Divisible Load Scheduling

Scheduling is an important research area for a long time which is also one of the main area of contemporary mathematics. The origin of scheduling on operation research is mainly about production and project management[5, 12, 33]. As computer systems become more complex, scheduling is now an important part for the design of compilers and libraries[7, 94], operating systems[32, 110] and real-time systems [99, 101, 108, 121].

Since the computer systems come to parallel distributed systems which have high computing capabilities to process larger computation tasks, how to exploit its parallelism in these system is one major challenge. Programmers often focus on improving functional parallelism which means to identify and adapt the features of serial programs to be properly executed in parallel. For data intensive applications, there is another kind of parallelism named *data parallelism* which means large computational load can be distributed among available processing units and executed in parallel.

Parallel load distribution is mainly about the partitioning of single large load for one processing unit. If the large load is processed as a whole, the processing time may be unbearable. To reduce the total time cost, the large load can be partitioned and distributed among the processing units in the system. However, it is very important to have knowledge of data features and system to assure an appropriate data partitioning. *Divisible Load Theory* (DLT)[17] is used to study the problem of partitioning and sharing load in parallel systems.

Divisible load theory is a mathematical model which can enable performance analysis of parallel and distributed systems by including both communication and computation problems[28]. The divisible load scheduling theory uses a system of linear equations to define load distribution and provides many models which have lots of advantages such as the ease of computation, the use of a schematic language and the facility to be applied to different fields[102, 20, 21]. There are many studies on the optimization of Divisible Load Scheduling (DLS) with DLT[8, 10, 11, 15, 16, 18, 112, 19, 35, 48, 47, 71, 119, 29, 52, 71, 81, 40, 111, 95, 106, 107, 120, 126].

However, the partitioning method depends on the load *divisibility property* which refers to the features that determine whether the load can be decomposed into a set of smaller ones[114]. The divisibility property classifies into indivisible and divisible. For divisible load, it can further classify into modularly divisible and arbitrarily divisible. On the one hand, the loads can be *indivisible* where the size of new pieces may be different and can not be further subdivided. Therefore, they do not have any precedence relations and need to be assigned and processed in single processor. On the other hand, the loads can be *divisible* including *modularly divisible* and *arbitrary divisible*.

A modularly divisible load can be divides into smaller modules based on the features of the load or the system. The load processing is completed when all the modules are processed and the processing of the modules should be subject to precedence relations. Normally such kind of loads are represented as interaction graphs tasks that the vertices correspond to the modules and the edges correspond to interaction between these modules and the precedence relationships.

For arbitrary divisible load, all the elements in the load can be processed in the same way. These kind of loads can be arbitrarily split into any number of load fractions. These fractions may or may not have precedence relationships. For example, the data can be arbitrarily partitioned but there may be a precedence relationship among the generated data chunks. Or if the data chunks do not precedence relationships, then each chunk can be processed independently.

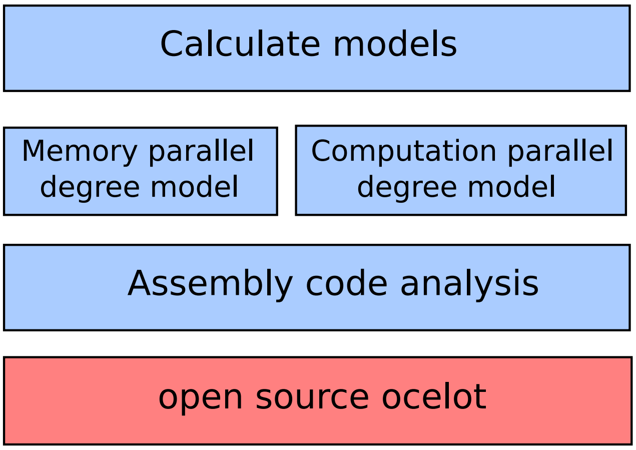
The applications that satisfy the divisibility property include image processing applications, processing of massive experimental data, signal processing applications, matrix computations, tree and database search and Monte Carlo simulations. As the divisible load scheduling considers that both the communication and computation loads can be arbitrarily partitioned among the parallel system[96], it is suitable for modeling a large class of data intensive problems. Under this scheme it is possible to model and schedule load distribution for systems with GPU devices.

Chapter 3

Performance Analytical Model

Contents

3.1 Architecture



Calculate models
Memory parallel degree model
Computation parallel degree model
Assembly code analysis
open source ocelot

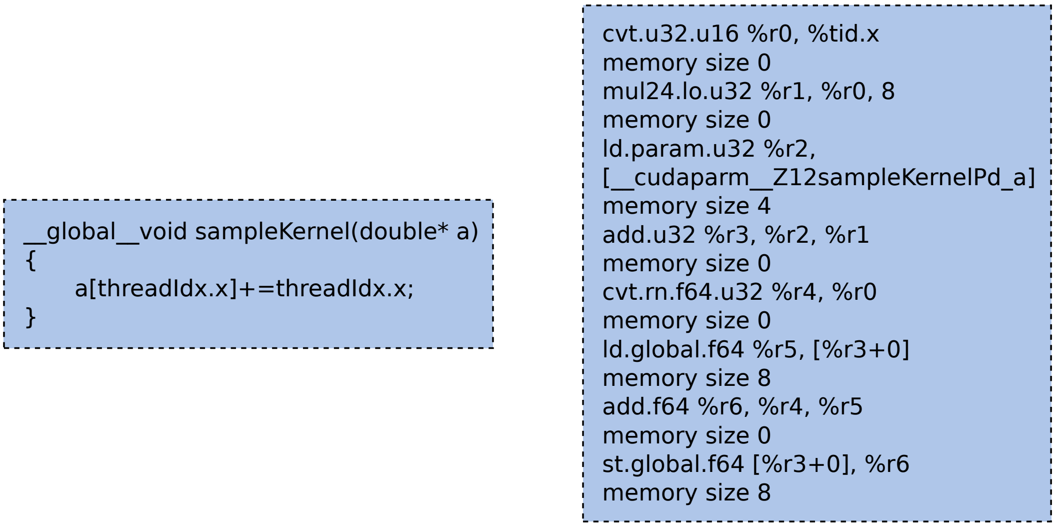
Figure 3.1: Performance Analytical Model Architecture

Our performance analytical model is based on the open source ocelot and includes four parts: assembly code analysis, memory parallel degree model, computation parallel degree model and calculate models as shown in Figure 3.1. The assembly code analysis part is responsible for PTX code generating, PTX instructions time cost testing and PTX code information gathering. The memory parallel degree model is used to describe the parallel execution for memory access instructions. The computation parallel degree model is used to describe the parallel execution for computation instructions. The calculate models will use the information from previous three parts to prediction the time cost of the total kernel execution.

3.2 Assembly Code Analysis

Parallel Thread Execution(PTX)[36] is a pseudo-assembly language used in Nvidia CUDA programming environment. The NVCC compiler translates the CUDA programs into PTX codes, and the GPU driver has a compiler which translates the PTX codes into machine codes to execute on GPUs. By analyzing the PTX codes, we can have a deep insight into the performance bottlenecks in GPU architecture.

With the help of ocelot, many details of PTX codes from CUDA program can be obtained. As shown in Figure 3.2, we can generate PTX codes from CUDA codes with ocelot. Therefore, we can design two CUDA kernels :*Kernel A* and *kernel B* that there only exists one PTX instruction difference between the two corresponding PTX codes. Then we can test the time cost of the PTX instruction by running the Kernel A and B and calculating the time difference. We design a set of micro



```
__global__void sampleKernel(double* a)
{
    a[threadIdx.x]+=threadIdx.x;
}
```

(a)CUDA code

```
cvt.u32.u16 %r0, %tid.x
memory size 0
mul24.lo.u32 %r1, %r0, 8
memory size 0
ld.param.u32 %r2,
[__cudaparm__Z12sampleKernelPd_a]
memory size 4
add.u32 %r3, %r2, %r1
memory size 0
cvt.rn.f64.u32 %r4, %r0
memory size 0
ld.global.f64 %r5, [%r3+0]
memory size 8
add.f64 %r6, %r4, %r5
memory size 0
st.global.f64 [%r3+0], %r6
memory size 8
```

(b)PTX code

Figure 3.2: PTX code generated from CUDA code with ocelot

Table 3.1: Time cost of PTX instructions in GTX 260 (unit: GPU clock)

	int_const	int_reg	float_const	float_reg
add	22	65	22	65
sub	22	65	22	65
mul	44	136	22	65
div	728	753	748	783
neg	22	22	17	17
min	62	62	62	62
max	62	62	62	62
and	64	64	64	64
or	62	62	62	62
xor	62	62	62	62
not	22	22	22	22
shl	22	63	22	63
shr	22	63	22	63
mv	40	40	40	40
cvt	22	22	22	22
ld/st	200	200	200	200

benchmarks to test the time cost of PTX instructions and Table 3.1 shows the time cost of PTX instructions in NVIDIA GTX 260.

3.3 Execution of Multiple Warps

To explain how the execution of multiple warps in each SM affects the total execution time, we use a typical scenario to illustrate as shown in Figure 3.3. For each warp, the PTX codes can be considered as an instruction queue of computation instructions and memory access instructions. We define a set of continuous computation instructions in one warp as a **computation task**. Similarly, we also define a set of continuous memory access instructions in one warp as a **memory access task**. With these definitions, the PTX codes can be considered as a crossed permutation of computation tasks and memory access tasks. The time period from the beginning of the ith computation task to the beginning of the $(i + 1)th$ computation task in one warp is called **the ith calculate period**. The time cost of the ith calculate period is T_i. Here our model firstly assume that one SM can only execute one warp in one time and the computation task can not be executed in parallel. (We will discuss the parallel execution of computation task in CPD submodel). Therefore,

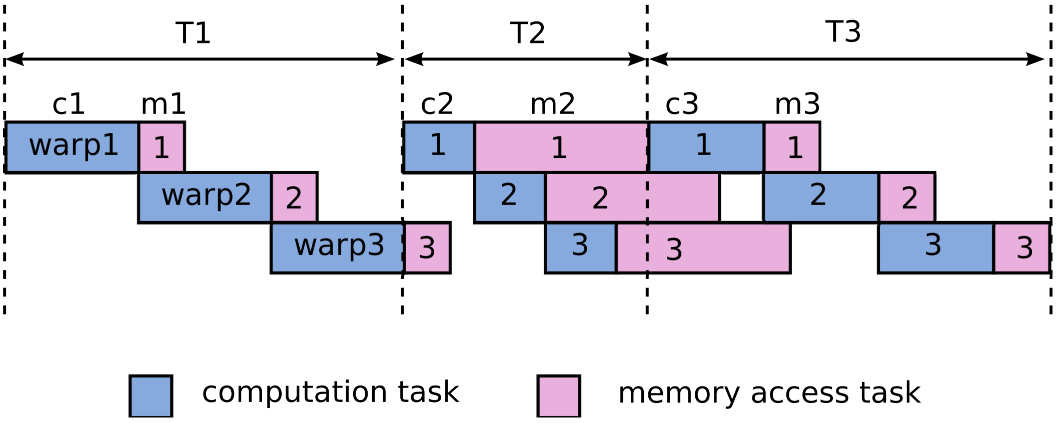


Figure 3.3: Multiple Warps Execution in GPU Architecture

the computation tasks between warps cannot be paralleled. However, the memory access tasks between warps can be executed in parallel. During the memory access waiting time, another active warp will be swapped to execute until the next memory access arrive.

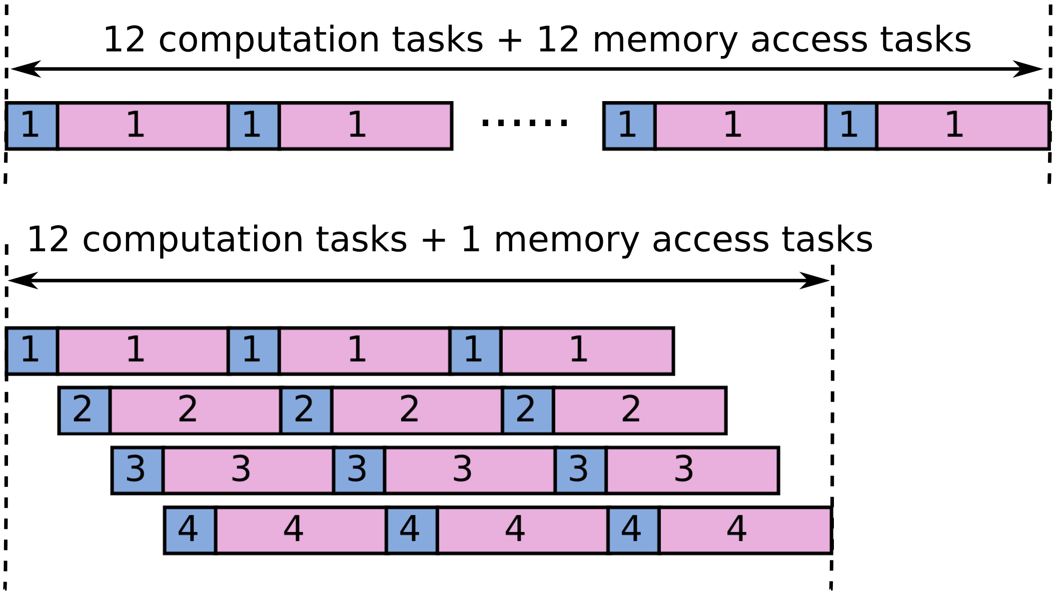

12 computation tasks + 12 memory access tasks
1
......
12 computation tasks + 1 memory access tasks
1
2
3
4

Figure 3.4: The Execution of Multiple Warps with MPD Awareness

3.3.1 MPD submodel

MPD is the Memory access Parallel Degree which is used to present the maximum warp number that can be executed in parallel. The MPD can greatly affect the total execution time. Here, we assume that there is no parallel execution for computation task. The value of MPD can be affected by the bandwidth of GPU device, the bandwidth used by each warp, the number of active warps in each SM and the number SMs in the GPU device. In one word, MPD means how many warps with memory access instructions can be executed in parallel under a limited bandwidth of device memory.

For example as shown in Figure 3.4, when there are not enough warps to execute or the value of MPD is very low (an extreme example is 1), the execution process would be serial execution like the case 1. When there are enough warps to execute and the value of MPD is very high (the value of MPD is higher than the number of active warps), the execution process would be like case 2. With high MPD, the latency of each memory access can be hidden by executing multiple memory access concurrently.

For each memory access task, we introduce the following equations to calculate MPD:

$$Warp_{bwt} = (N_{thread} * D_{mem})/(N_{trans} * t_{mem}), \tag{3.1}$$

$$MPD = \min\{N_{act}, \lfloor GPU_{bwt}/(N_{act} * N_{sm} * Warp_{bwt}) \rfloor\}. \tag{3.2}$$

N_{thread}: the number of threads in one warp, in this paper is 32;

D_{mem}: the data size required for each thread during each memory access;

N_{trans}: the number of memory transactions for each memory access instruction;

N_{act}: the number of active warps in one SM;

N_{sm}: the number of active SMs;

t_{mem}: the latency of memory access;

$Warp_{bwt}$: the bandwidth used by one warp during one memory access;

GPU_{bwt}: the bandwidth of GPU device.

We obtain the memory access addresses of half-warp threads with ocelot[88] and calculate the number of memory transactions by following the rule of the generation of memory transaction in PTX 1.4[36].

3.3.2 CPD submodel

CPD is the computation parallel degree which is used to present the parallel execution between warps and within warps in one SM. The parallel executions for the computation instructions in GPU are so complex that it is hard to give a perfect model to present. Many factors can affect the parallel execution degree such as the relationship of adjacent instructions, instruction types, computing resource requirements, the number of warps and the features of applications. To simplify the model, we only take the number of warps and the features of applications into consideration.

We use **computation instruction proportion** in the PTX codes to present the features of applications which is defined as follows:

$$P = T_{cmp}/(T_{cmp} + T_{mem}). \tag{3.3}$$

P: the computation instruction proportion;
T_{cmp}: the sum of the time cost of all computation instructions;
T_{mem}: the sum of the time cost of all memory access instructions. When the computation instruction proportion is very low, the increase of the number of warps will lead to the increase of parallel execution of computation instructions. With more warps, the number of computation instructions which can be executed in parallel will show a linear increase. Because of the low computation instruction proportion, the computing resources are always available to execute computation instructions in parallel. On this reasoning, we propose the following equation to calculate the CPD:

$$CPD1 = (c - P) * (N_{act} - b) + a. \tag{3.4}$$

a, b and c: the empirical parameters which get from each specific GPU device. (We write a micro benchmark to obtain these parameters. In GTX 260, a is set to 3, b is set to 11 and c is set to 0.5.)

The GPU will schedule warps to execute once there are spare computing resources. When the computation instruction proportion is high enough, the increase rate of the CPD will decrease along with a big enough warp number. Although the increase of warps leads to a linear increase of computation instructions that can be parallelized, the available computing resources become fewer and the increase rate of computation instructions that have enough computing resource to execute in parallel decreases. Therefore, when all computing resources are used up, the CPD will come to a limitation. In this situation, we use the following equations to calculate the CPD:

$$CPD2 = (n/(m-1)) * \sqrt{(m-1)^2 - (N_{act} - m)^2} + a, \tag{3.5}$$

$$n = d * (P - c)^2. \tag{3.6}$$

m: the maximum warp number that can be executed in GPU;

d: the empirical parameters which get from each specific GPU device.

We write a micro benchmark to obtain the parameter and d is set to 80 in GTX260. Therefore, the final value of CPD for a specific number of warps is equal to $\min\{CPD1, CPD2\}$. Notice that our CPD model is an empirical model as we do not know the details of the computing instruction execution. Based on black box testing, we have current CPD model.

3.4 Calculation Model

So far, we have explained the execution of multiple warps and two submodels. In this section, we put them all together into the prediction model to predict the total time cost of the execution. Notice that each block will be assigned to a SM for the execution. The execution of computation instructions between SM is relatively independent. The execution of memory access instructions can affect each other between SM as the local memory bandwidth is shared by all SMs. We have considered the factors in the previous two submodels. The total time cost of an application in GPU is equal to the time cost of SM which has the longest execution time cost. Therefore, to simplify the prediction, we can only consider the single block execution in one SM.

By analyzing the PTX codes, we can calculate the time cost of each calculate period and sum them up to get the total time cost. The calculate methods for each calculate period may be different due to long memory access waiting from current calculate period or previous calculate period. Ideally we hope the processors always have instructions to execute. However, long memory access tasks can let the processors wait because the following computation tasks need the results from the previous memory access tasks. Therefore, according to whether the calculate period has been affected, we classify the calculate periods into four types. We can analyze the relationships between time cost of computation tasks and memory access tasks in current period and previous period to select a corresponding type.

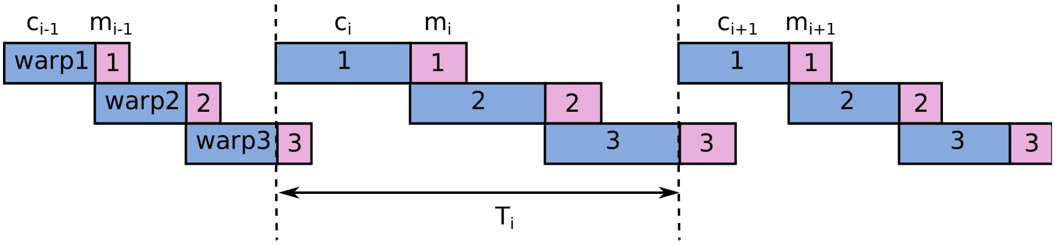


Figure 3.5: Calculate Period Type 1

As Figure 3.5 shows, the type 1 is that there is no long memory access waiting influence from current period and previous period which means $c_{i-1} \geq m_{i-1}$ and $c_i \geq m_i$. Here, the parameters are defined as follows:

c_i: the time cost of computation task in the ith calculate period;

m_i: the time cost of memory access task in the ith calculate period;

T_i: the time cost of the ith calculate period.

Therefore, only the computation tasks make contribution to the total time cost. We sum up the time cost of all computation tasks while the parallel execution of computation parts should also be taken into consideration. We can use the following equation to calculate the ith calculate period time cost:

$$T_i = \lceil (N_{act} * c_i)/CPD \rceil. \tag{3.7}$$

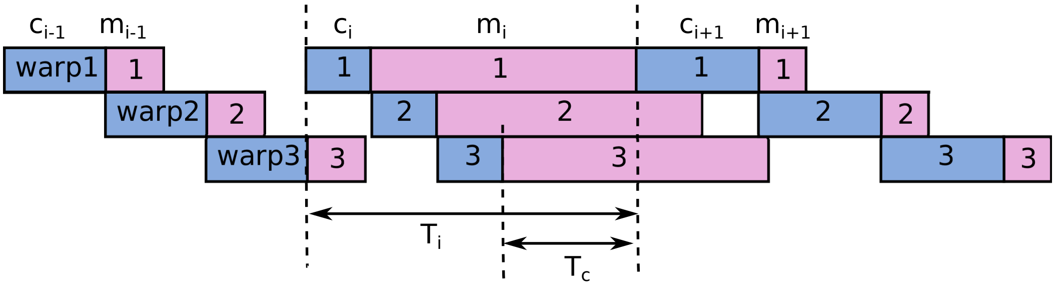


Figure 3.6: Calculate Period Type 2

In type 2, $c_{i-1} \geq m_{i-1}$ and $c_i < m_i$ as illustrated in Figure 3.6. The long memory access tasks in current period will cause a waiting period between the last computation task in current period and the first computation task in the following period. In this case, we can use the following equations to calculate the ith calculate period time cost:

$$T_i = \lceil (N_{act} * c_i)/CPD \rceil + T_c, \tag{3.8}$$

$$T_c = max\{m_i - (n-1) * c_i, 0\}. \tag{3.9}$$

T_c: the extra time cost caused by the long memory access tasks in current period.

In type 3, $c_{i-1} < m_{i-1}$ and $c_i \geq m_i$ as illustrated in Figure 3.7. The long memory access tasks in previous period will cause a waiting time between the execution of computation tasks in current period because the results of memory access task in previous period do not arrive. Therefore, we can use the following equations to calculate the ith calculate period time cost:

$$T_i = \lceil (N_{act} * c_i)/CPD \rceil + T_p, \tag{3.10}$$

$$T_p = \max\{m_{i-1} * \lfloor (N_{act}/MPD) - 1 \rfloor - (N_{act} - 1) * c_i, 0\}. \tag{3.11}$$

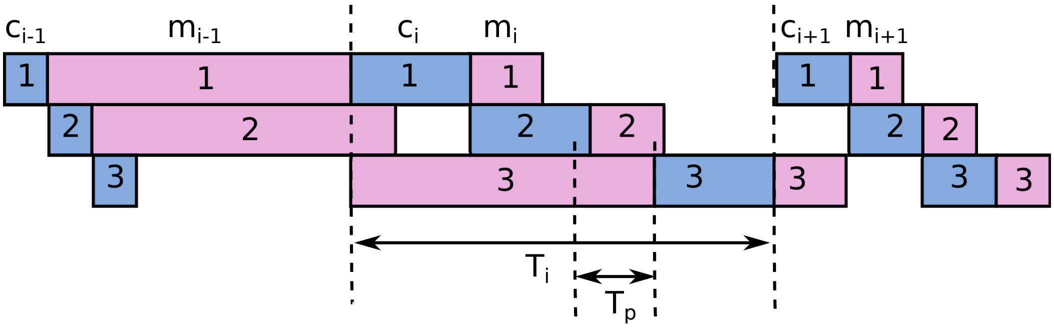


Figure 3.7: Calculate Period Type 3

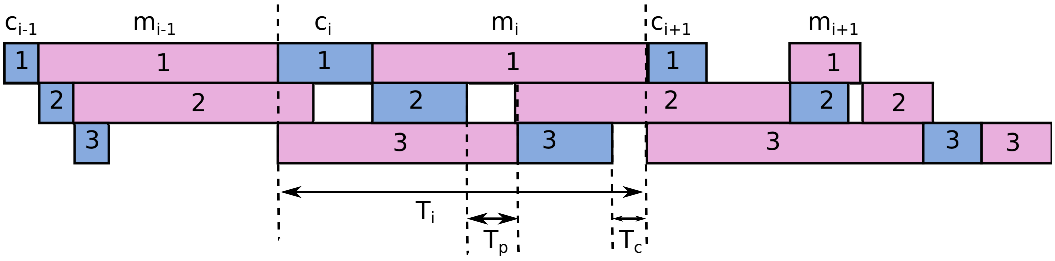


Figure 3.8: Calculate Period Type 4

T_p: the extra time cost caused by the long memory access task in previous period.

In type 4, $c_{i-1} < m_{i-1}$ and $c_i < m_i$ as illustrated in Figure 3.8. The long memory access tasks from current period and previous period both make extra time cost in the current calculate period. We can use the following equations to calculate the ith calculate period time cost:

$$T_i = \lceil (N_{act} * c_i)/CPD \rceil + T_p + T_c, \tag{3.12}$$

$$T_p = \max\{m_{i-1} * \lfloor (N_{act}/MPD) - 1 \rfloor - (N_{act} - 1) * c_i, 0\}, \tag{3.13}$$

$$T_c = \max\{m_i - ((N_{act} - 1) * c_i + T_p)\}. \tag{3.14}$$

Finally, we can calculate the time cost for each calculate period according to the different scenarios and sum up to obtain the total time cost.

3.5 Experiment Evaluation

3.5.1 Configuration

The GPUs used in our experiments are shown in Table 3.2. We have to big machines as shown in Table 3.3. We install GTX 260 and Tesla C2050 in machine AT38 and install Telsa C2075 and Tesla K20c in machine AT50. We use cudaEventRecord API in CUDA 4.2 to measure the execution time of GPU kernels. All the benchmarks are compiled with NVCC.

To test the performance of our prediction model, we use four different benchmarks that are mostly used in Linderman's work[78] and we port them from multicore platform to GPU platform. The benchmarks used in our work are explained in Table 3.4. The computation instruction proportions of the four benchmarks are different from very low 26.23% to very high 86.97%. We use these in the hope of proving our model can have good prediction results in various applications. We compare our predicted results with the actual test results. The abbreviations used in the Figures are as follow:
test: the results from the actual execution in GPU;
model: the results from our performance model prediction.

Table 3.2: The features of GPUs used in this work

features	GTX 260	Tesla C2050	Tesla C2075	Tesla K20c
the number of SPs	192	448	448	2496
CUDA core frequency(MHz)	1242	1150	1150	710
Memory size	896 MB	3 GB	6GB	4800MB
Memory bandwidth(GB/s)	111.9	144	144	208
Peak Tflop/s (double precision)	1.43	1.03	1.03	3.5

Table 3.3: Specification of The Machines

machine	Device	Cores	Clock speed	Cache	Main Memory
AT38	4 x Intel Xeon X5650	4 x 6	2.67GHz	12MB	6 x 2GB
AT50	2 x Intel Xeon E5-2680	2 x 8	2.7GHz	20MB	8 x 8GB

Table 3.4: The features of benchmarks used for Performance Prediction

benchmarks	description	input size	com proportion
Matrix	Matrix multiple	256 × 256	28.2%
Linear[78]	Image filter to compute 9-pixels avg	800 × 800	45.84%
Sepia[78]	Filter for artificially aging images	800 × 800	52.97%
Black-scholes[78]	European option pricing	900000	86.97%

3.5.2 Results

As GTX 260 and Tesla C2050 are old GPUs and Tesla C2075 and Tesla K20c are much newer than the previous two, the maximum thread numbers of Tesla C2075 and Tesla K20c are larger than GTX 260 and Tesla C2050. We change the number of warps to run the kernel with the same data size. For GTX 260 and Tesla C2050, we vary the number of warps from 1 to 16. For Tesla C2075 and Tesla K20c, we vary the number of warps form 1 to 32. Besides comparing the execution time cost with the prediction results, we also discuss about the accuracy. We define the accuracy as follow:

$$P_{acc} = \min\{T_{test}, T_{model}\} / \max\{T_{test}, T_{model}\}. \tag{3.15}$$

P_{acc}: the accuracy for a specific number of warps;
T_{test}: the time cost for a specific number of warps from actual measured results;
T_{model}: the time cost for a specific number of warps from predicted results.

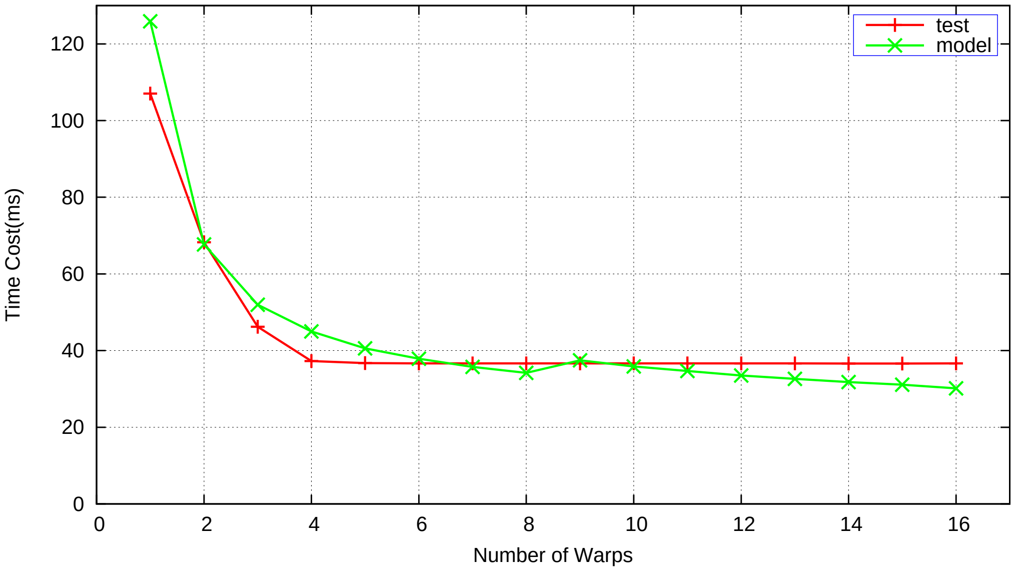


Figure 3.9: Linear Filter Results in GTX260

First of all, we use linear filter benchmark to test our work in the four type GPUs. The computation instruction proportion of linear filter benchmark is 45.86%. The execution results of GTX 260 and Tesla C2050 are shown in Figure 3.9 and Figure 3.10. The accuracy results of GTX 260 and Tesla C2050 are shown in Figure 3.11. We can find the performance of GTX 260 is much faster than Tesla C2050.

With our performance model, the average accuracy of GTX 260 is 91.18% while the average accuracy of Tesla C2050 is 89.52% as shown in Table 3.5. As we have some error factors in the performance model such as the empirical CPD model, we can hardly achieve perfect prediction. Due to lack of cache simulation, the prediction accuracy of Tesla C2050 is poorer than GTX 260 because GTX 260 has no cache while Tesla C2050 has.

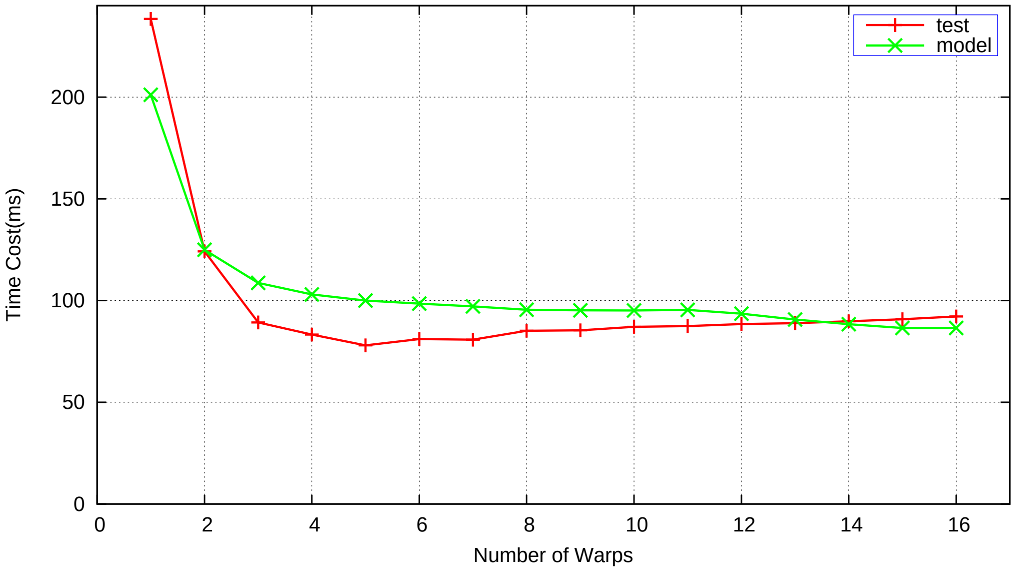


Figure 3.10: Linear Filter Results in Tesla C2050

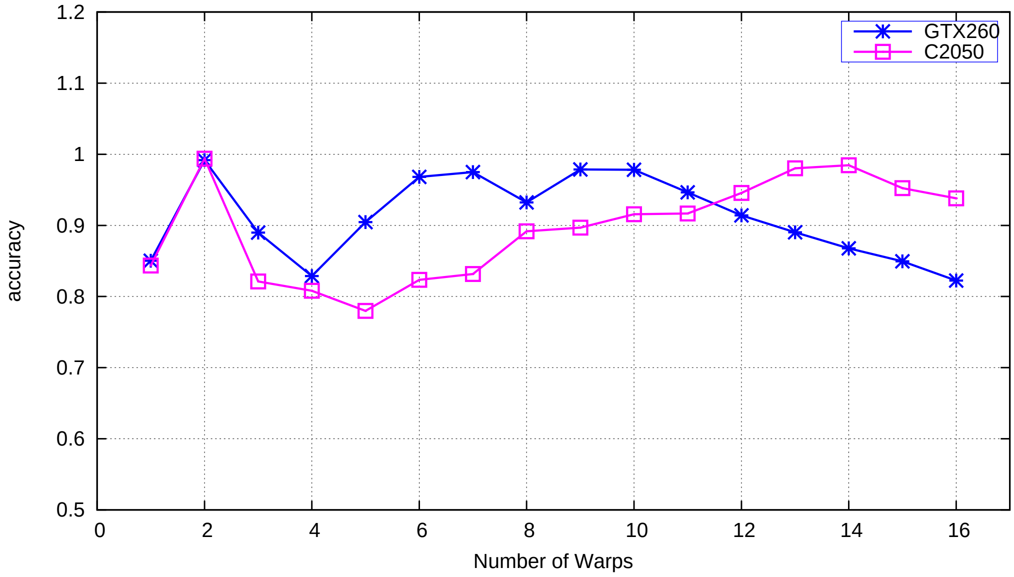


Figure 3.11: The Accuracy of Linear Filter in GTX260 and Tesla C2050

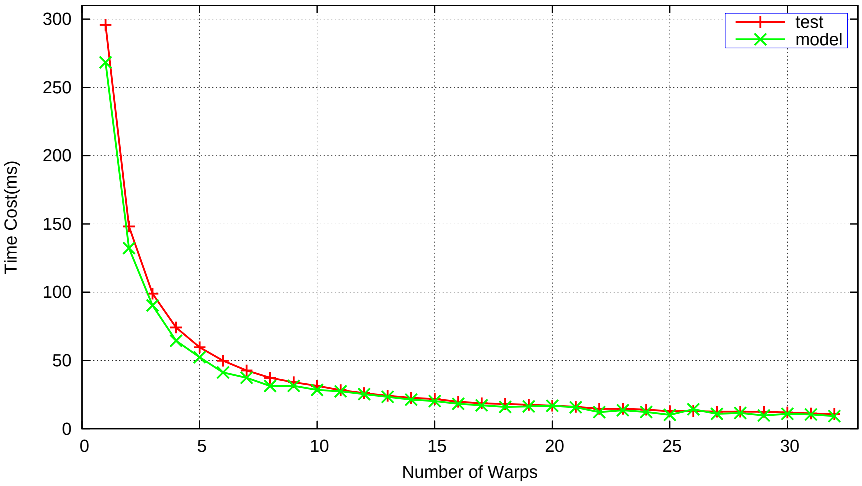


Figure 3.12: Linear Filter Results in Tesla C2075

We also repeated the experiments in Tesla C2075 and Tesla K20c which is much newer than the previous two GPUs. The execution results of Tesla C2075 and Tesla K20c are shown in Figure 3.12 and Figure 3.13. The accuracy results of Tesla C2075 and Tesla K20c are shown in Figure 3.14. Although the peak performance of K20c is much better than C2075, the GPU frequency of K20c is slower than C2075. Therefore, the execution of C2075 is faster than K20c with the same thread configuration. The average accuracy of C2075 is 90.17% while the average accuracy of K20c is 89.46%.

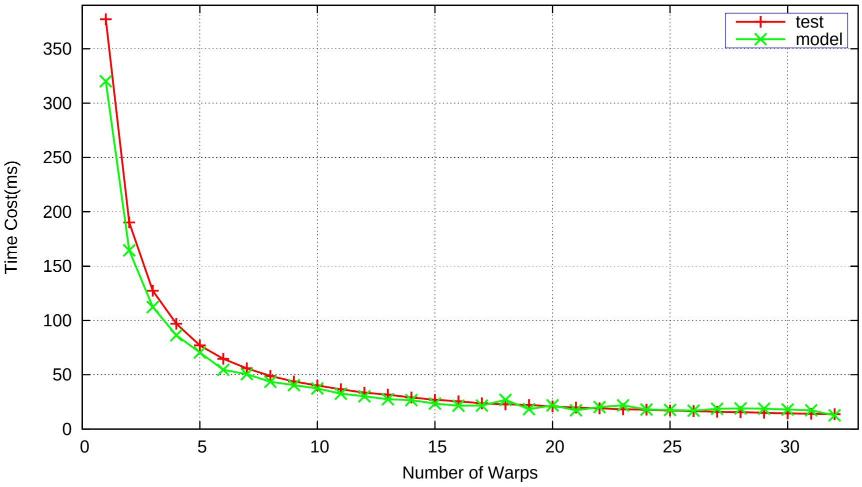


Figure 3.13: Linear Filter Results in Tesla K20c

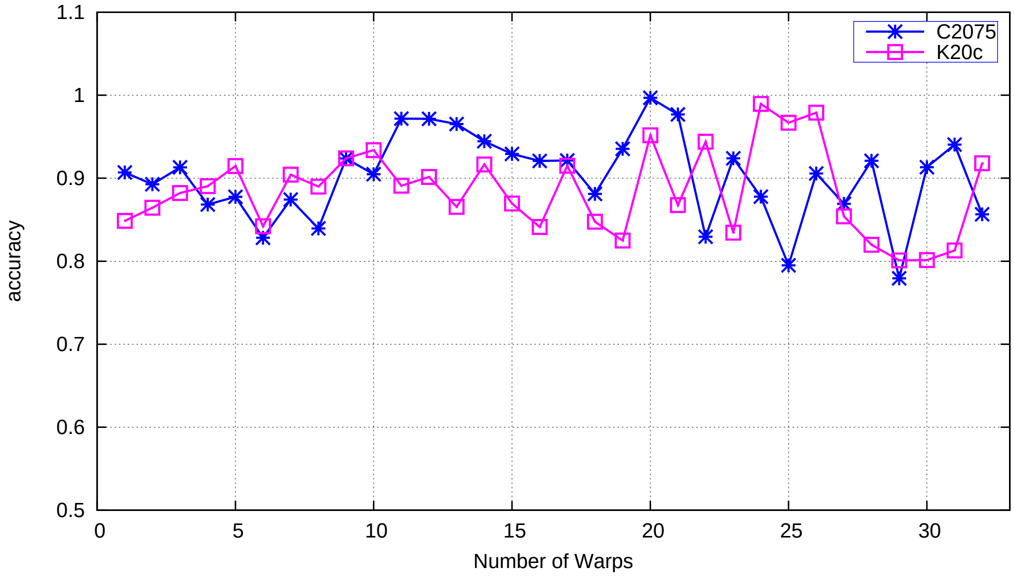


Figure 3.14: The Accuracy of Linear Filter in Tesla C2075 and Tesla K20c

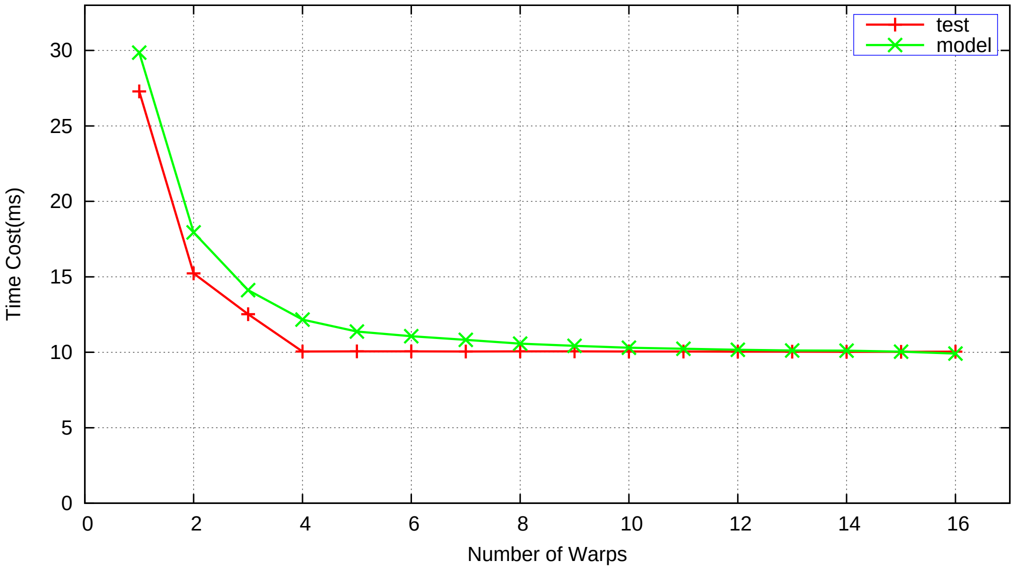


Figure 3.15: Sepia Filter Results in GTX260

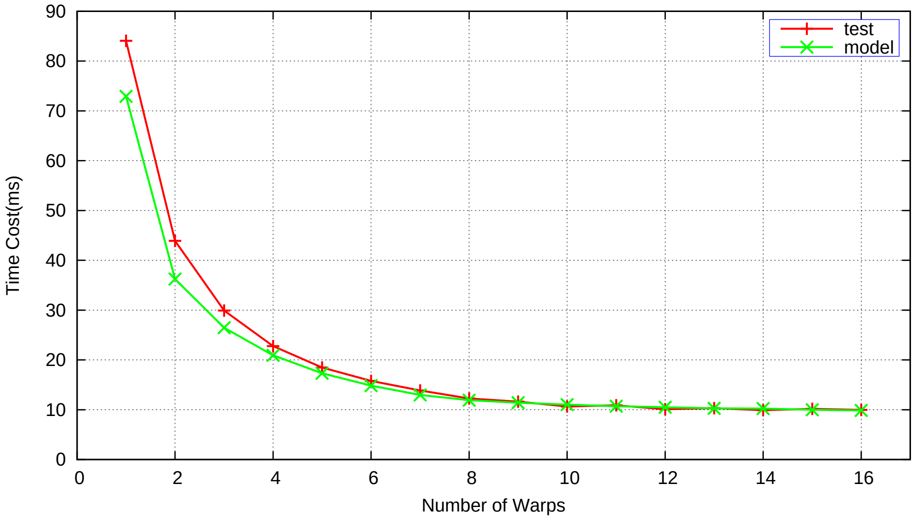


Figure 3.16: Sepia Filter Results in Tesla C2050

Then we use sepia filter benchmark to test our performance model. The computation instruction proportion is 52.97% which is a little higher than linear filter benchmark. First of all, we test our work in the two old GPUs. The execution time results are shown in Figure 3.15 and Figure 3.16 and the accuracy results are shown in Figure 3.17. The average accuracy for GTX 260 is 93.94% and 94.47% in Tesla C2050 which is much higher than linear filter. This is because the influence of cache decreases due to high computation instruction proportion.

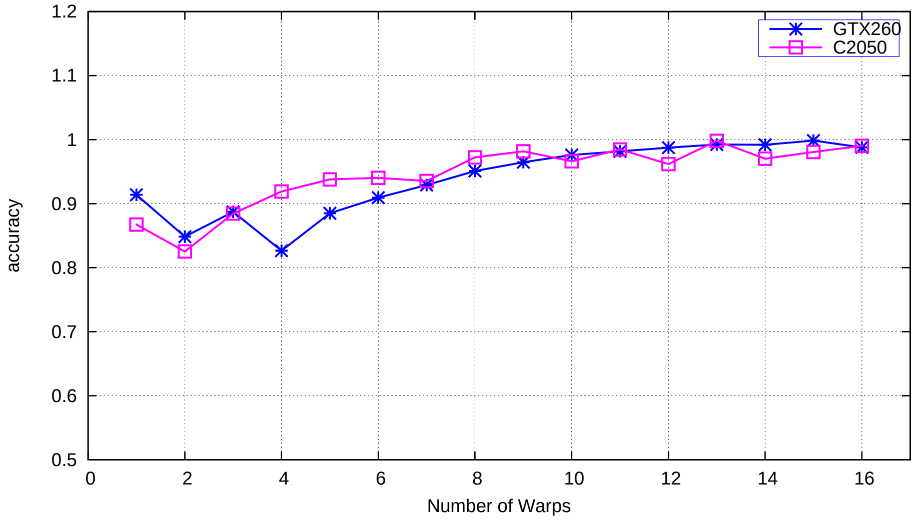


Figure 3.17: The Accuracy of Sepia Filter in Two GPUs

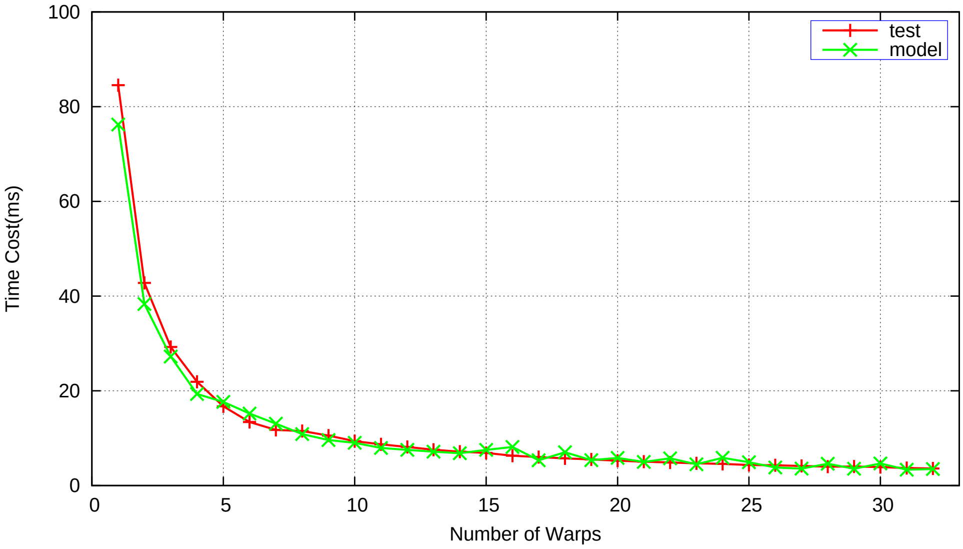


Figure 3.18: Sepia Filter Results in Tesla C2075

We repeated the sepia filter experiments in Tesla C2075 and Tesla K20c. The execution time results are shown in Figure 3.18 and Figure 3.19 and the accuracy results are shown in Figure 3.20. The average accuracy of Tesla C2075 is 91.1% and the average accuracy of Tesla K20c is 91.3%. This results show we can have a good prediction in the new GPU as well.

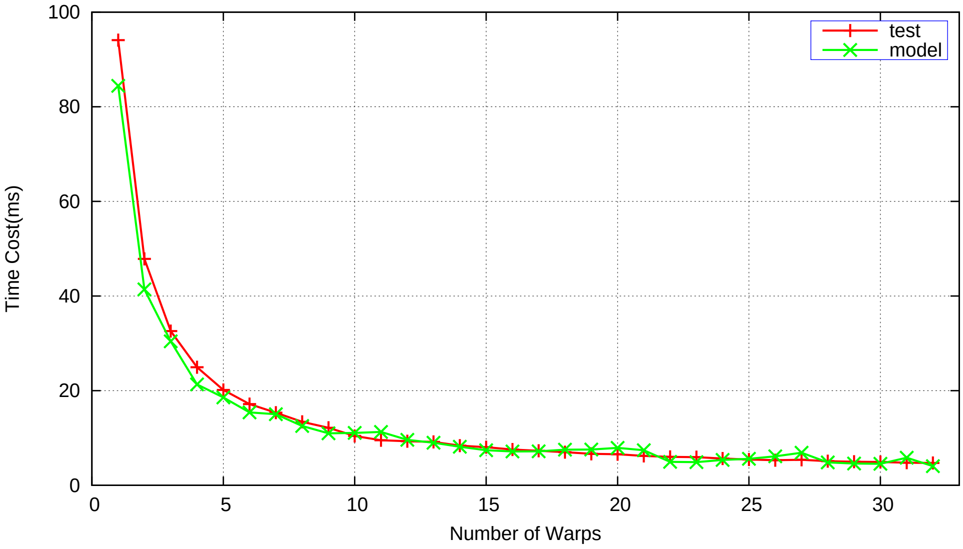


Figure 3.19: Sepia Filter Results in Tesla K20c

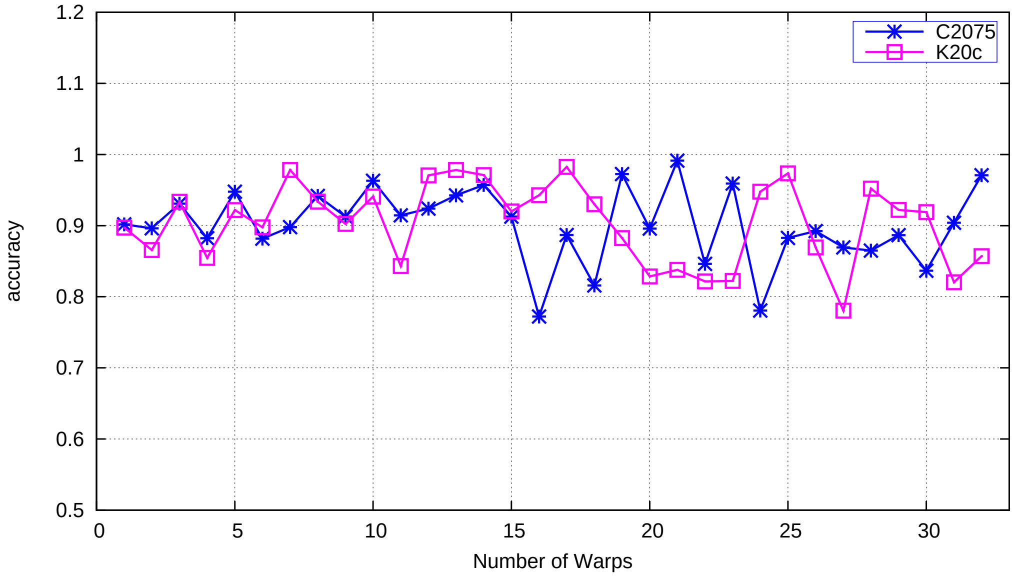


Figure 3.20: The Accuracy of Sepia Filter in Tesla C2075 and Tesla K20c

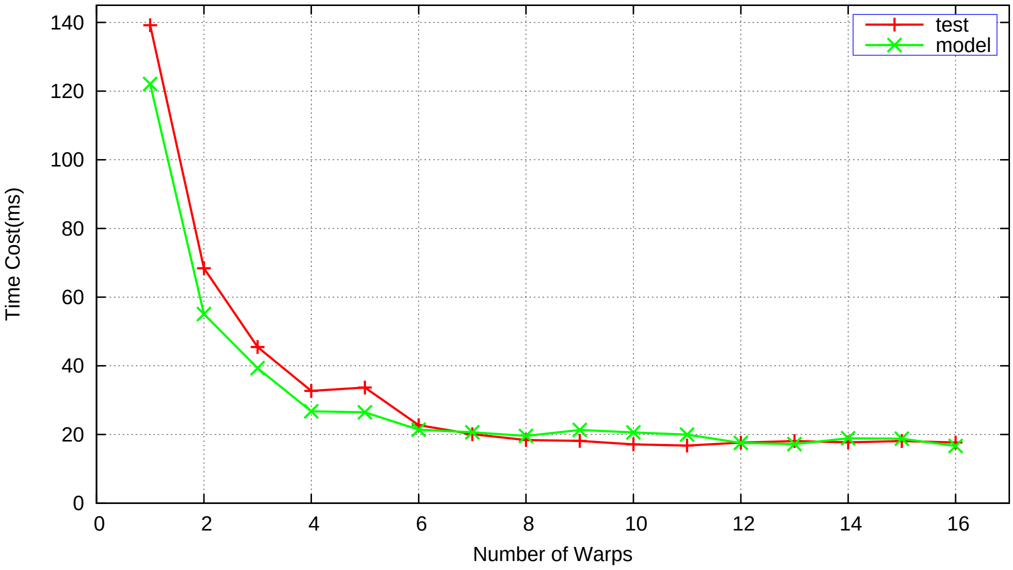


Figure 3.21: Black-scholes Results in GTX 260

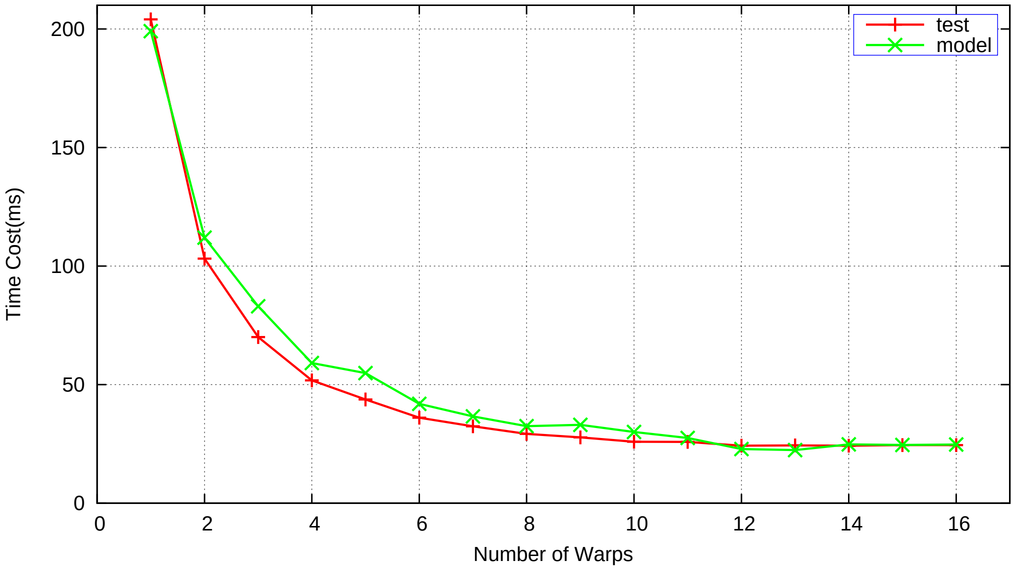


Figure 3.22: Black-scholes Results in Tesla C2050

We also use black-scholes benchmark which has 86.97% high computation instruction proportion to test our work. First of all, we test in the two old GPUs. The execution results are shown in Figure 3.21 and Figure 3.22 and the accuracy results are shown in Figure 3.23.

We can find that the execution decreases along with the increase of warp numbers and finally comes to a limitation. This is because there are enough computing resources for all threads when the number of threads is small. Therefore, the increase of thread number can improve the performance as more data are processed in parallel. However, when the number of threads is very large, there are no enough computing resources to support all thread execution. This results in resources waiting for some threads.

Table 3.5: The Arithmetic Means of Accuracy for Each Benchmark

benchmarks	linear filter	sepia filter	black-scholes	matrix
GTX260(100%)	91.18	93.94	89.48	90.73
C2050(100%)	89.52	94.47	90.78	84.87
C2075(100%)	90.17	91.11	90.44	86.62
K20c(100%)	89.46	91.3	89.97	85.86

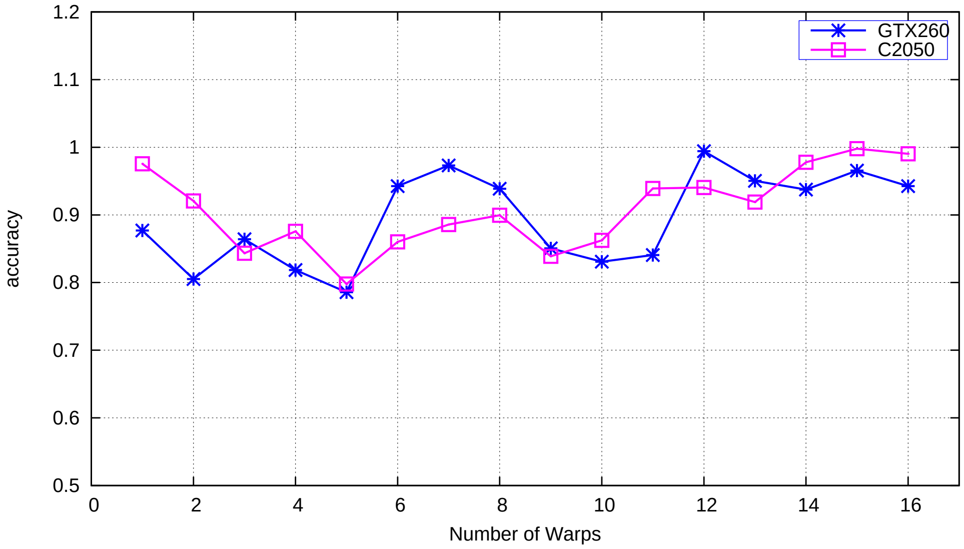


Figure 3.23: The Accuracy of Black-scholes in Two GPUs

Our performance model can well predict the kernel execution no matter the number of thread is large or small. The average accuracy of GTX 260 is 89.48% and the average accuracy of Tesla C2050 is 90.78%. Here the accuracy of Tesla C2050 is better than GTX 260. That is because the influence of cache hit is very low as the memory access instruction proportion of black-scholes is very low.

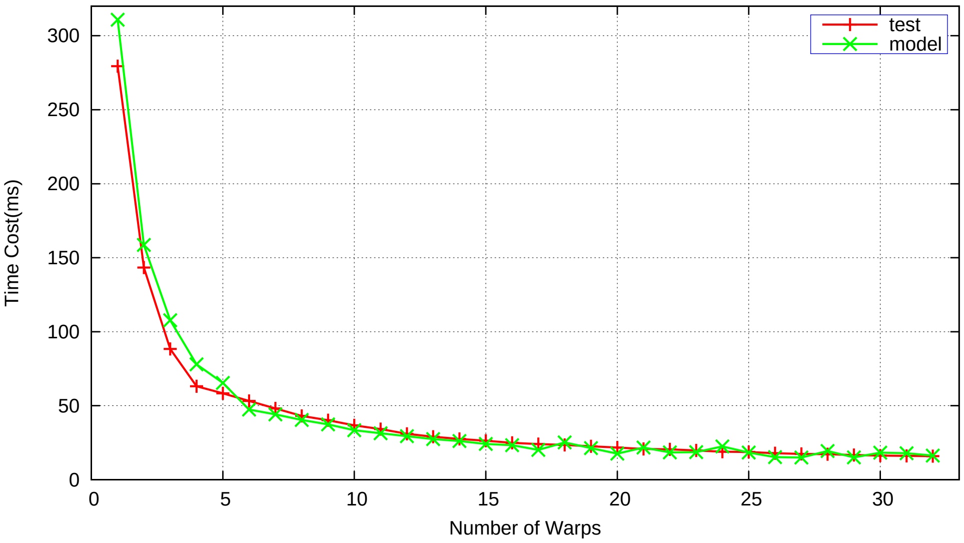


Figure 3.24: Black-scholes Results in C2075

Then we repeat the black-scholes experiments in Tesla C2075 and Tesla K20c. The execution time results are shown in Figure 3.24 and Figure 3.25 and the accuracy results are shown in Figure 3.26. The average accuracy of Tesla C2075 is 90.44% and the average accuracy of Tesla K20c is 89.97%.

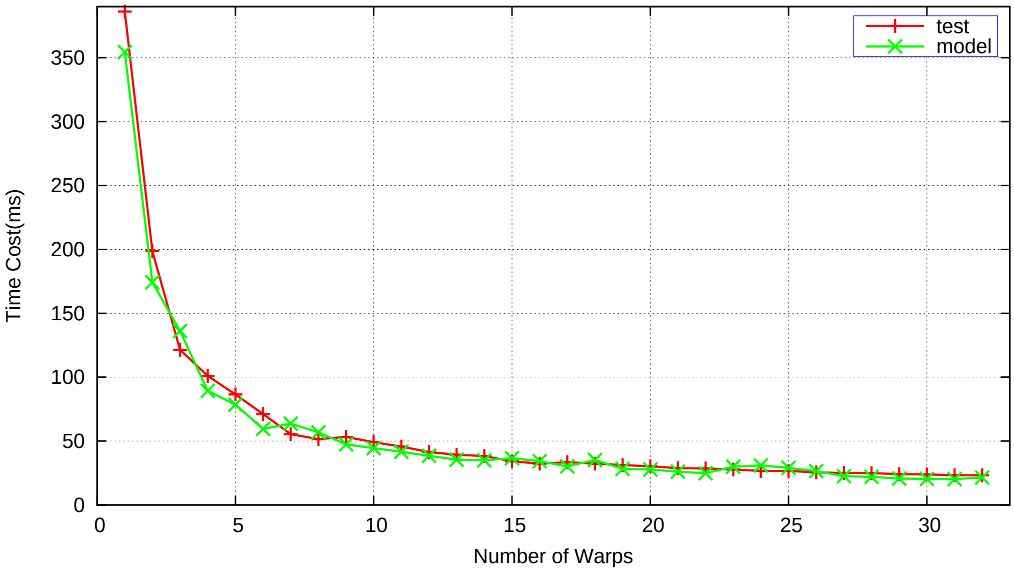


Figure 3.25: Black-scholes Results in Tesla K20c

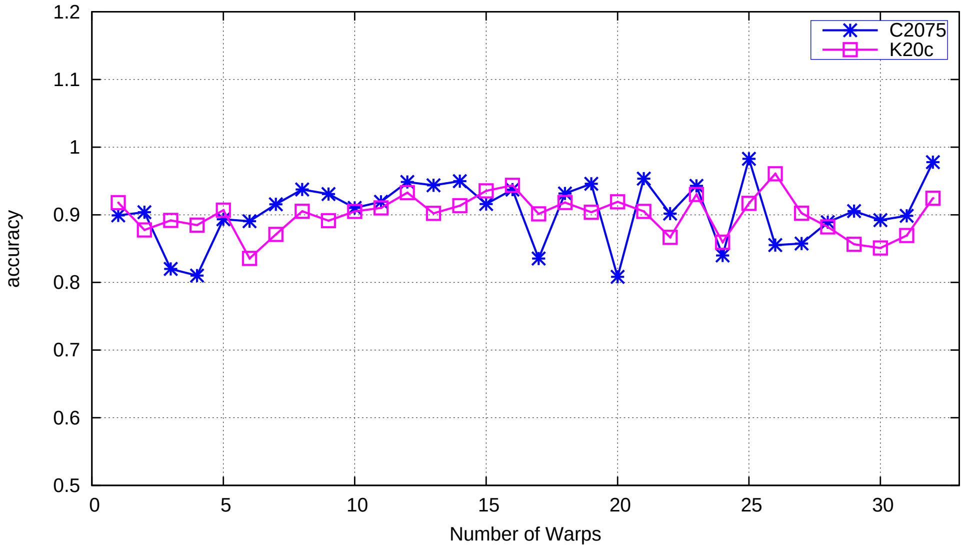


Figure 3.26: The Accuracy of Black-scholes in Tesla C2075 and Tesla K20c

Finally we use matrix multiplication benchmark to test our work. The computation instruction proportion of matrix multiplication is 28.2% which is the lowest in all benchmarks used in this thesis. The execution time results of GTX 260 and Tesla C2050 are shown in Figure 3.27 and Figure 3.28 and the accuracy results are shown in Figure 3.29.

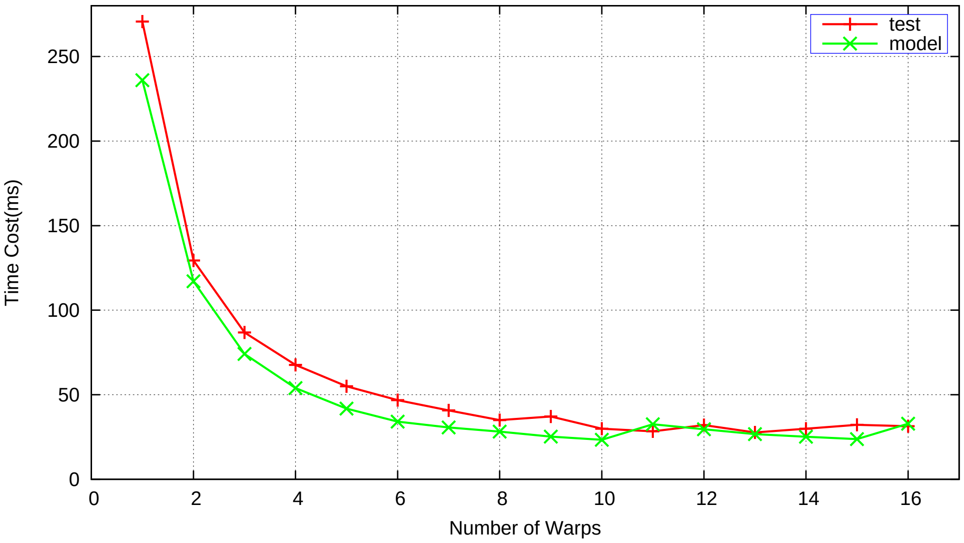


Figure 3.27: Matrix Multiplication Results in GTX260

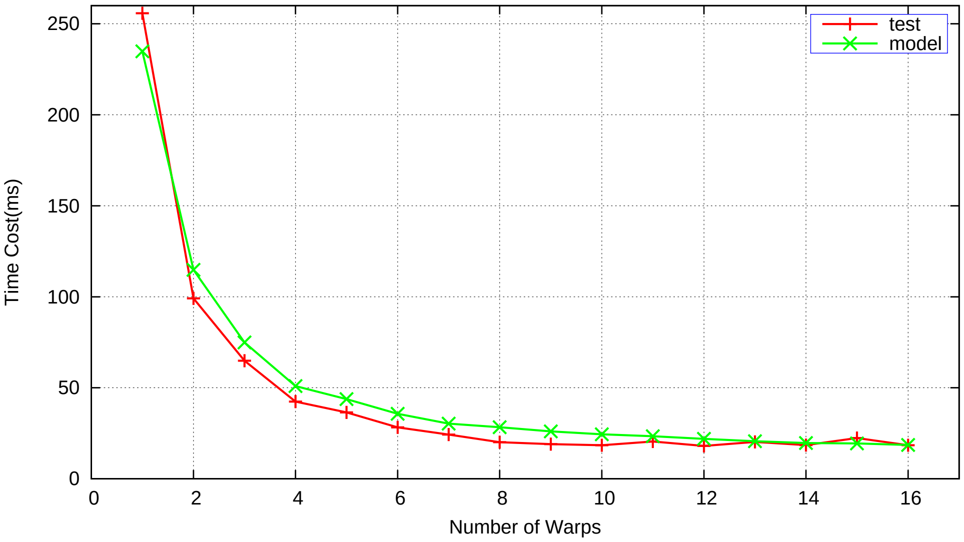


Figure 3.28: Matrix Multiplication Results in Tesla C2050

The average accuracy of GTX 260 is 90.73% while the average accuracy of Tesla C2050 is 84.87%. As matrix multiplication benchmark is memory access bound, the cache hits can greatly affect the performance. Without cache simulation, our performance model can hardly have the same prediction results in GPUs with cache comparing to GPUs without cache.

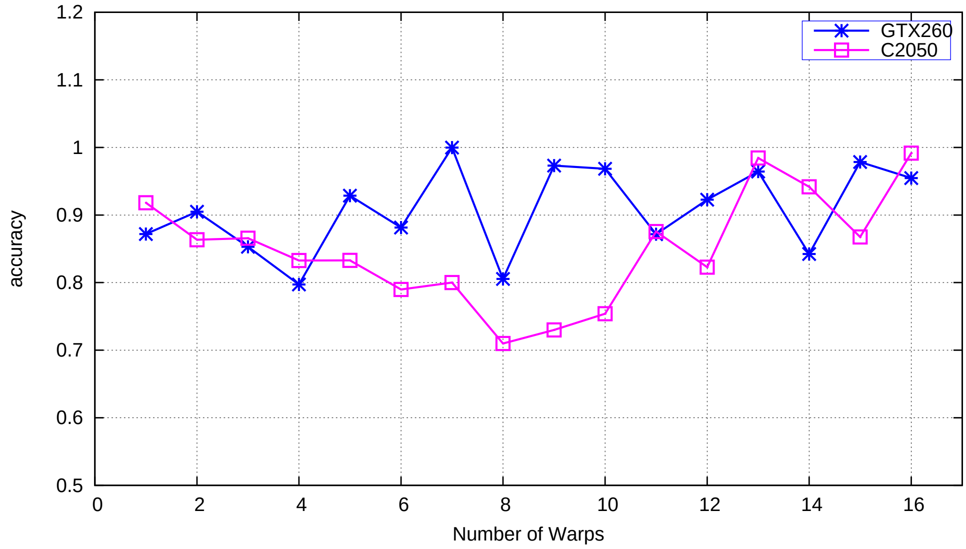


Figure 3.29: The Accuracy of Matrix Multiplication in Two GPUs

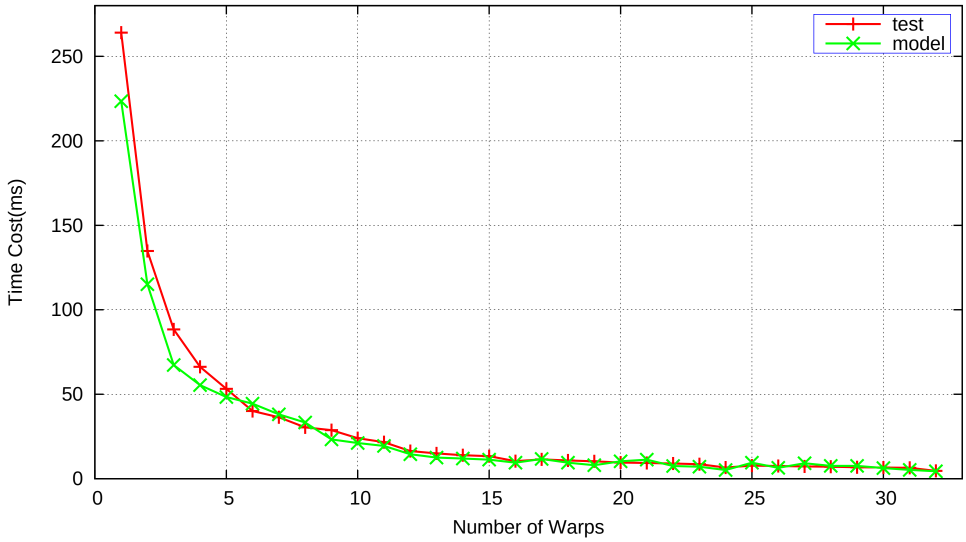


Figure 3.30: Matrix Multiplication Results in Tesla C2075

We repeat the matrix multiplication benchmark in Tesla C2075 and Tesla K20c. The execution time results are shown in Figure 3.30 and Figure 3.31 and the accuracy results are shown in Figure 3.32. The average accuracy of Tesla C2075 is 86.62% and the average accuracy of Tesla K20c is 85.86%. The accuracy of Tesla C2075 and Tesla K20c is not as good as the accuracy of GTX 260 due to cache.

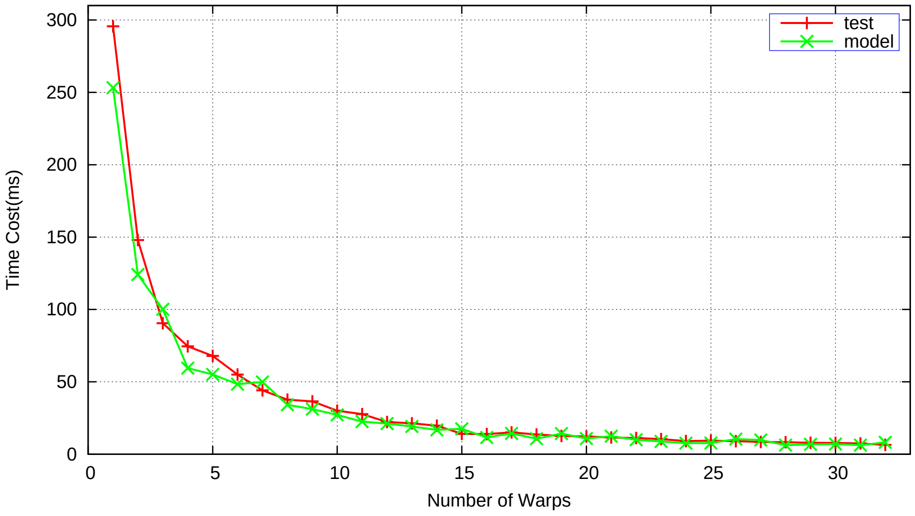


Figure 3.31: Matrix Multiplication Results in Tesla K20c

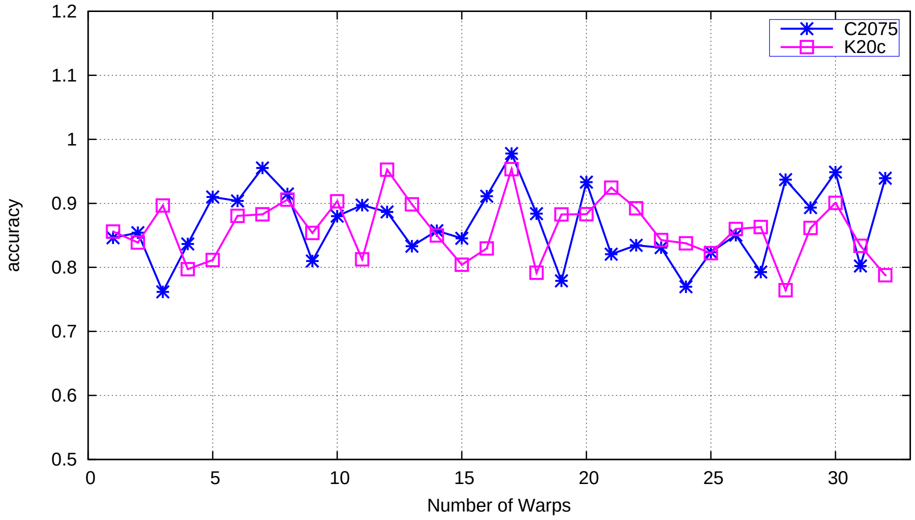


Figure 3.32: The Accuracy of Matrix Multiplication in Tesla C2075 and Tesla K20c

Chapter 4

A Task Partitioning and Scheduling Method

Contents

To have good partitioning and scheduling, we need to take the characteristic of applications into consideration. Also we need to notice that the limited bandwidth between CPU and GPU is shared by all GPUs in multiple GPU architecture. Therefore, we propose an application classification and a time optimal data transfer algorithm besides the task partitioning and scheduling method in this section.

4.1 Application Classification

It is important to analyze the features of applications for achieving high performance. The GPGPU applications are classified into several basic types. For the classification, the computation-to-communication ratio of applications and the send-to-receive ratio of applications are taken into consideration as shown in Figure 4.1.

From the computation-to-communication ratio aspect, the applications can be classified into *kernel bound applications* and *data transfer bound applications*. The kernel bound applications has longer time cost of kernel execution than the time cost of data transfer while the data transfer bound applications has longer time cost of data transfer than the time cost of kernel execution.

From the send-to-receive ratio aspect, the applications can be classified into *send heavy*, *receive heavy* and *general heavy*. For send heavy applications, the time cost of data receive is very short while the time cost of data send is very long. For receive heavy applications, the time cost of data send is very short while the time cost of data send is very long. For general heavy applications, both the time cost of data send and receive are not short.

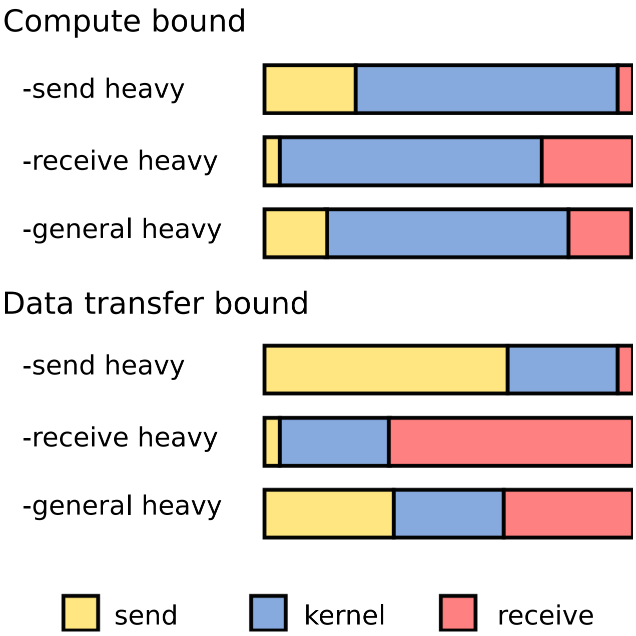


Figure 4.1: The Classification of Applications

If programmers are familiar with their applications, then they can decide the application type by themselves. However, no all programmers are familiar with their applications. In this case, they can use the performance analytical model in Section 3 to help them with the type decision. First of all, we can estimate the time cost of data sending and receiving based on the bandwidth between the CPU and GPU. Then we can use the performance analytical model to predict the time cost of kernel execution. With the time cost of each part, it is easy to decide the application

Table 4.1: The Type of Applications

benchmark	input size	send	kernel	receive	type
linear filter	8000^2	3.7%	92.1%	4.2%	compute bound and general heavy
sepia filter	8000^2	14.2%	69.5%	16.3%	compute bound and general heavy
black filter	5×10^5	48.5%	45.3%	6.2%	data transfer bound and send heavy
matrix	$358400 \times 512 \times 14$	11.9%	87.7%	0.4%	compute bound and send heavy
matrix	$16384 \times 4 \times 7186$	0.6%	79.3%	20.1%	compute bound and receive heavy

type. We use the performance analytical model to decide the application type which will be used in the experiments and the results in Tesla C2075 are shown in Table 4.1.

4.2 TPSM for Single GPU Architecture

Our goal is to hide the communication latency by partitioning the application into subtasks and overlapping the data send, data receive and kernel execution part of these subtasks. According to different application types, TPSM provides corresponding solution with awareness of application features. Based on the application classification, TPSM provides six scheduling sub-methods for the corresponding six basic types. Any application can be described as a combination of the six basic types. Notice that our TPSM is not a theoretically optimal solution but a heuristic solution.

The basic idea of TPSM is to partition one application into small *tasks* with different size as shown in Figure 4.2. Each task can be considered as a small application which includes data send, kernel execution and data receive. The total scheduling process includes many steps. In each step, we can launch data send, kernel execution and data receive from different subtasks. As our target applications are divisible, the kernel execution is proportional to data size. Therefore, we can decide the subtask size by the send data size. To best hide the communication latency, the subtask size in each step is important. We will introduce some exponential increase or decrease for the subtask size. The exponential increase means the current subtask size in current is a time of the subtask size in previous step.

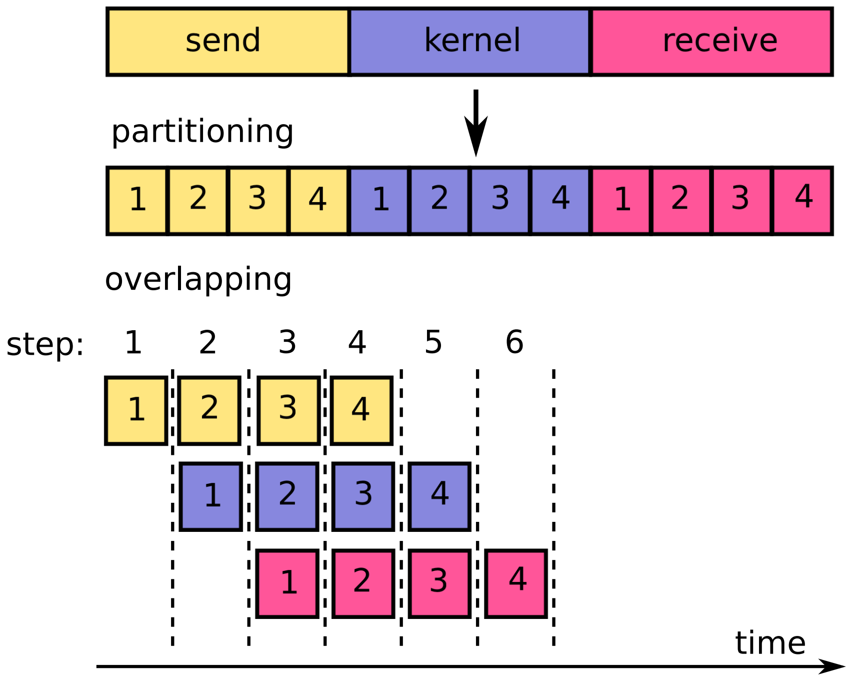


Figure 4.2: The Basic Idea of TPSM

4.2.1 Partitioning and Scheduling for Kernel Bound Applications

For kernel bound applications, the kernel execution is the main part. Therefore, the total time cost of the application is mainly decided by the kernel execution part. To minimize the total time cost, we should make sure the time cost of kernel execution is not shorter than the time cost of the bi-directional data transfer in each step and make the time cost of first send and last receive as small as possible.

4.2.1.1 Send Heavy

For kernel bound and send heavy applications, the receive part is so small that we can receive all the data in the last step. To make the first send as small as possible, we set the first subtask size to a minimum size which can just keep all threads working and then exponentially increase the subtask size as shown in Figure 4.3.

Suppose that the base value of the exponential increase is a and the send data size in step 1 is ds_{min}. Then the send data size in step i is $ds_{min} \times a^{i-1}$. We can calculate ts_i and tk_i in step i as follow:

$$ts_i = ds_{min} \times a^{i-1}/S_S, \tag{4.1}$$

$$tk_i = T_k \times ds_{min} \times a^{i-2}/D_s. \tag{4.2}$$

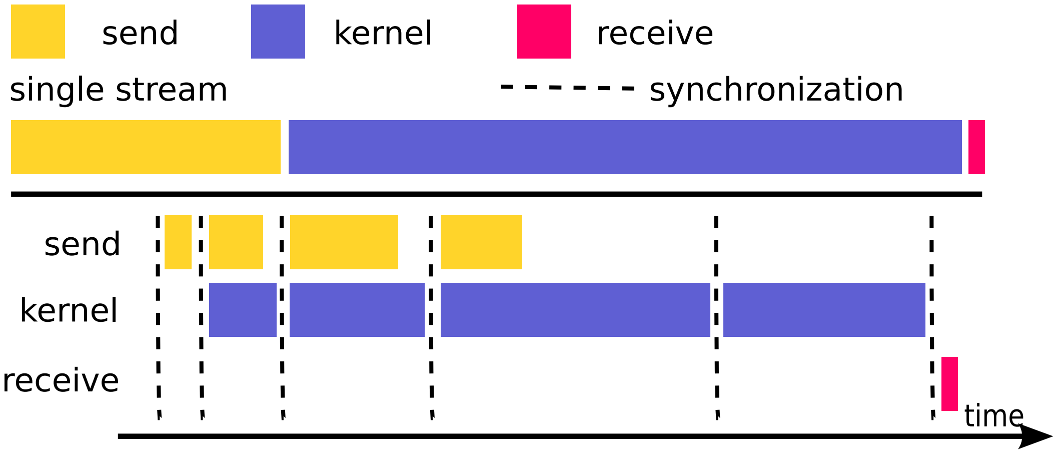


Figure 4.3: TPSM for Kernel Bound and Send Heavy in Single GPU Architecture

D_s: the total data size to send from host to device (given from programmers);
T_k: the total time cost of kernel execution in single GPU (given from programmers);
ds_{min}: minimum send data size to feed all threads working (Normally it is the send data size that can enable each thread computing and have one output element);
S_S: the bandwidth of only sending;
ts_i: the time cost of data sending in step i;
tk_i: the time cost of kernel execution in step i.

We suppose that programmers are familiar with their applications so that they can provide the information such as the total send data size and kernel time cost. As the total time cost is the sum of kernel execution, first send, last receive and the synchronization overhead. With fixed first send, last receive and kernel execution, the base value a can affect the total time cost by affecting the synchronization overhead. With larger a, there will be less synchronization times. Therefore, the optimal base value a is the maximum value which can also make sure the time cost of kernel execution is not shorter than the time cost of data send in each step. Then we can calculate the a that matches the condition as follow:

$$ds_{min} \times a^{i-1}/S_S \leq T_k \times ds_{min} \times a^{i-2}/D_s, \tag{4.3}$$

$$a \leq S_S \times T_k/D_s. \tag{4.4}$$

Therefore, $a = S_S \times T_k/D_s$ is the optimal base value.

Proof of Asymptotic Optimality

The scheduling length of our sub-method is as follow:

$$T_{total} = ds_{min}/S_S + T_k + D_r/R_R + t_{str} \times log_a(D_s(a-1)/ds_{min} + 1). \tag{4.5}$$

D_r: the total data size to receive from device to host;
R_R: the bandwidth of only receiving;
t_{str}: the time cost of single synchronization operation.
As the application is kernel bound and send heavy, we can assume the receive data size to be constant. The data size of sending is proportional to the time cost of kernel execution, assuming $D_s = p \times T_k$ (p is constant). Then we have a is constant. As ds_{min} and t_{str} are also constant, then we have:

$$T_{total} \leq T_k + O(log_a(p \times T_k)). \tag{4.6}$$

As the optimal scheduling length is $T_{opt} = T_k$, then we have

$$\lim_{T_k \to \infty} T_{total}/T_{opt} \leq 1 + \lim_{T_k \to \infty} O(log_a(p \times T_k))/T_k = 1. \tag{4.7}$$

Therefore, the sub-method of TPSM for kernel bound and send heavy in single GPU architecture is asymptotic optimal.

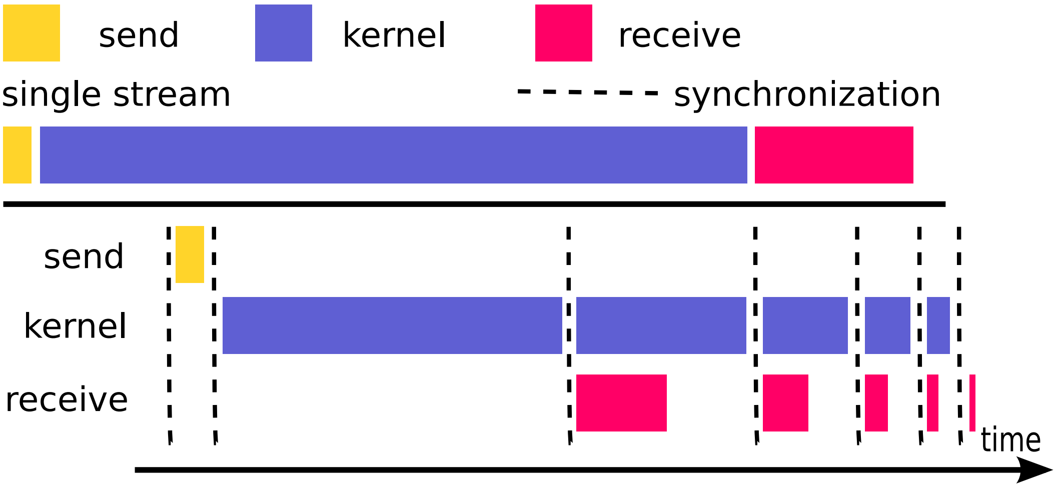


Figure 4.4: TPSM for Kernel Bound and Receive Heavy in Single GPU Architecture

4.2.1.2 Receive Heavy

For kernel bound and receive heavy applications, the send part is so small that we can send all the data in the first step. To make the last receive as small as possible, we set the first subtask size in a large size and then exponentially decrease the subtask size to make the last subtask size just feed all threads working as shown in Figure 4.4.

We set a minimum receive data dr_{min} which feeds all threads can return results and suppose that the base value of the exponential decrease is a and the number of subtask is m. Then the receive data size in the step i is $dr_{min} \times a^{m+2-i}$. We can calculate the time cost of kernel execution and data receive in the step i as follow:

$$tr_i = dr_{min} \times a^{m+2-i}/R_R, \tag{4.8}$$

$$tk_i = T_k \times dr_{min} \times a^{m+1-i}/D_r. \tag{4.9}$$

D_r: the total data size to receive from device to host;
R_R: the bandwidth of only receiving;
dr_{min}: minimum receive data size to feed all threads working;
tr_i: the time cost of data receiving in step i.
Similarly, the base value a can affect the total time cost by affecting the synchronization times. Therefore, the optimal base value a should be the maximum value while make sure the time cost of kernel execution is not shorter than the time cost of data receive in each step. Then we can calculate the base value a that matches the condition as follow:

$$dr_{min} \times a^{m+2-i}/B_R \leq T_k \times dr_{min} \times a^{m+1-i}/D_r, \tag{4.10}$$

$$a \leq R_R \times T_k/D_r. \tag{4.11}$$

Then we can choose $a = R_R \times T_k/D_r$ as the optimal base value to schedule.

Proof of Asymptotic Optimality

The scheduling length of our sub-method is as follow:

$$T_{total} = D_s/S_S + T_k + dr_{min}/R_R + t_{str} \times log_a(D_r(a-1)/dr_{min} + 1). \tag{4.12}$$

As the application is kernel bound and receive heavy, we can assume the send data size to be constant. The data size of receiving is proportional to the time cost of kernel execution, assuming $D_r = p \times T_k$ (p is constant). Then $a = R_R \times T_k/D_r$ is constant. As dr_{min} and t_{str} are also constant, then we have:

$$T_{total} \leq T_k + O(log_a(p \times T_k)). \tag{4.13}$$

As the optimal scheduling length is $T_{opt} = T_k$, then we have:

$$\lim_{T_k \to \infty} T_{total}/T_{opt} \leq 1 + \lim_{T_k \to \infty} = 1. \tag{4.14}$$

Therefore, the sub-method of TPSM for kernel bound and receive heavy in single GPU architecture is asymptotic optimal.

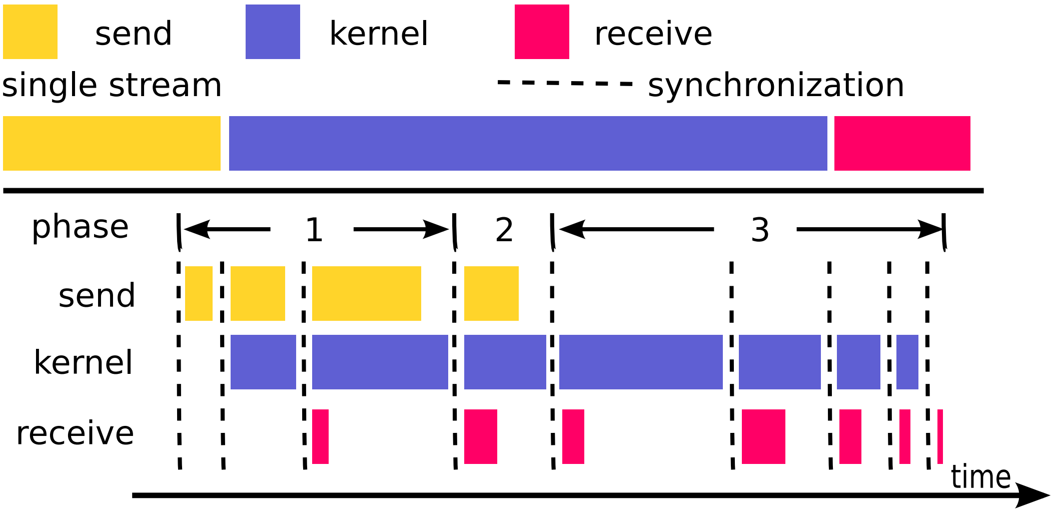


Figure 4.5: TPSM for Kernel Bound and General Heavy in Single GPU Architecture

4.2.1.3 General Heavy

For kernel bound and general heavy applications, both the send and receive parts are not small so that we should not send or receive all data in one step. We divide the scheduling into three phases: (1)*send phase*, (2)*middle phase* and (3)*receive phase* as shown in Figure 4.5. To minimize the total time cost, we should ensure the time cost of kernel execution is not shorter than the time cost of the bi-directional data transfer in each step of each phase. Therefore, we begin with a small subtask size and exponentially increase the subtask size in the send phase. In the middle phase, there is only one step in which we send all the left data and receive all the output data from the last step of the send phase. We set the time cost of kernel execution not shorter than the data transfer part by setting a suitable subtask size. Finally, we begin with a large subtask size and then exponentially decrease it to make the last receive as small as possible in the receive phase.

In the send phase, we assume the base value of the increase is a. Then the data send size in step i of the phase send is $ds_{min} \times a^{i-1}$. We can use the following equations to calculate the time cost of kernel execution in step i:

$$tk_i = (ds_{min} \times T_k / D_s) \times a^{i-2}. \tag{4.15}$$

The data receive size in step i is $(ds_{min} \times D_r / D_s) \times a^{i-3}$. Then the time cost of data send and data receive in step i is as follow when only consider uni-directional data transfer:

$$ts_i = ds_{min} \times a^{i-1} / S_{SR}, \tag{4.16}$$

$$tr_i = ds_{min} \times D_r \times a^{i-2}/(D_s \times R_{SR}). \tag{4.17}$$

S_{SR}: the bandwidth of sending while GPU is sending and receiving at the same time;
R_{SR}: the bandwidth of receiving while GPU is sending and receiving at the same time.
We also overlap the data send and data receive as well. Due to the different time cost of data sending and receiving, the data transfer process will begin with simultaneous bi-directional data transfer and end with uni-directional data transfer. If the time cost of data sending is longer than the time cost of data receiving, then the total data transfer time is the sum of the first simultaneous bi-directional data transfer and the time cost of the rest data sending. On the contrary, the total data transfer time is the sum of the first simultaneous bi-directional data transfer and the time cost of the rest data receiving. We can use the following equation to calculate it:

$$tcom_i = \begin{cases} tr_i + (ds_{min} \times a^{i-1} - tr_i \times S_{SR})/S_S \\ \qquad\qquad\qquad if \quad tr_i \leq ts_i, \\ \\ ts_i + (ds_{min} \times a^{i-3} \times D_r/D_s - ts_i \times R_{SR})/R_R \\ \qquad\qquad\qquad if \quad tr_i > ts_i. \end{cases} \tag{4.18}$$

$tcom_i$:the total communication time cost between the host and the device in step i of the send phase.
In each step of the send phase, we need to keep $tcom_i \leq tk_i$. With equation (4.15) and (4.18), we can get A and B that match the condition $tcom_i \leq tk_i$ as follow:

$$\begin{cases} A = \{a|\beta - \sqrt{\beta^2 - \gamma} \leq a \leq \beta + \sqrt{\beta^2 - \gamma} \ \& \ a \geq \delta\}; \\ \\ B = \{a|\lambda - \sqrt{\lambda^2 - \mu} \leq a \leq \lambda + \sqrt{\lambda^2 - \mu} \ \& \ a \leq \delta\}; \\ \\ where\ \delta = \sqrt{D_r \times S_{SR}/(D_s \times R_{SR})}, \\ \qquad \beta = S_S \times T_k/2D_s, \\ \qquad \gamma = (S_S - S_{SR}) \times D_r/(D_s \times R_{SR}), \\ \qquad \lambda = T_k \times S_{SR} \times R_R/(2D_s \times (R_R - R_{SR})), \\ \qquad \mu = D_R \times S_{SR}/(D_s \times (R_R - R_{SR})). \end{cases} \tag{4.19}$$

Then we can choose the maximum base value from set A and B as the optimal base value for the send phase:

$$a = \max\{a|a \in A \ or \ a \in B\}. \tag{4.20}$$

Because the application is kernel bound application. The kernel is longer than the data transfer part. Therefore, there exists solution a in equation (4.20).

As the base value a in send phase is fixed, then we can calculate the number of subtasks in the send phase as follow:

$$P = \lfloor \log_a(D_s(a-1)/ds_{min} + 1) \rfloor. \tag{4.21}$$

The size of remaining send data is as follow:

$$d_s = D_s - ds_{min} \times (a^P - 1)/(a-1). \tag{4.22}$$

The size of receive data in the step of the middle phase is:

$$d_r = (ds_{min} \times D_r \times a^{P-2})/D_s. \tag{4.23}$$

d_s: the data size of send in the step of the middle phase;
d_r: the data size of receive in the step of the middle phase.
Then we can calculate the total time cost of the bi-directional data transfer in the middle phase as follow:

$$tcom_{p2} = \begin{cases} d_r + (d_s - d_r \times S_{SR})/S_S \\ \qquad\qquad if\ d_s/S_{SR} > d_r/R_{SR}, \\ \\ d_s + (d_r - d_s \times R_{SR})/R_R \\ \qquad\qquad if\ d_s/S_{SR} \leq d_r/R_{SR}. \end{cases} \tag{4.24}$$

$tcom_{p2}$: the total time cost of the bi-directional data transfer in the middle phase. To ensure the kernel part is the main part in the middle phase, we set the subtask size to $tom_{p2} \times D_s/T_k$.

In the receive phase, the time cost of the remaining kernel execution is not shorter than the time cost of the remaining data receive as the application is kernel bound application. Therefore, we set a large size for the first subtask size in the receive phase and then exponentially decrease the subtask size to make the final receive part as small as possible as shown in Figure 4.5. Still suppose that the minimum receive data is dr_{min} and the base value is b. Then we can calculate the time cost of data receive and kernel execution in step i in the receive phase as follow:

$$\begin{aligned} tr_i &= dr_{min} \times b^{m+1-i}/R_R, \\ tk_i &= dr_{min} \times T_k \times b^{m-i}/D_r. \end{aligned} \tag{4.25}$$

To ensure $tr_i \leq tk_i$, we can get the base value as follow:

$$b \leq T_k \times R_R/D_r \tag{4.26}$$

Then we choose $b = T_k \times R_R/D_r$ as the optimal base value for the scheduling in the receive phase.

Proof of Asymptotic Optimality

The scheduling length of our sub-method is as follow:

$$\begin{aligned} T_{total} \leq ds_{min}/S_S + T_k + dr_{min}/R_R + t_{str} \times log_a(D_s(a-1)/ds_{min}+1) \\ +t_{str} \times log_b(D_r(b-1)/dr_{min}+1). \end{aligned} \tag{4.27}$$

As the application is kernel bound and general heavy, then we assume that $D_s = pT_k,\ D_r = qT_k$ where p and q are constant. As $ds_{min},\ dr_{min},\ a,\ b$ are constant, the scheduling length of our work can be:

$$T_{total} \leq T_k + O(log_a(pT_k)) + O(log_b(qT_k)). \tag{4.28}$$

Therefore, as the optimal scheduling length is $T_{opt} = T_k$, then we have:

$$\lim_{T_k \to \infty} T_{total}/T_{opt} \leq 1 + \lim_{T_k \to \infty} O(log_a(pT_k))/T_k + \lim_{T_k \to \infty} O(log_b(qT_k))/T_k = 1. \tag{4.29}$$

Therefore, the TPSM sub-method for kernel bound and general heavy in single GPU architecture is asymptotic optimal.

4.2.2 Partitioning and Scheduling for Data Transfer Bound Applications

For data transfer bound applications, the bi-directional data transfer between the host and the device is the main part. Therefore, we prefer to hide the kernel execution by overlapping the data send, data receive and kernel execution. we should also ensure that the time cost of data transfer part is not shorter than the time cost of kernel execution in each step.

4.2.2.1 Send Heavy

For data transfer bound and send heavy application, the receive part is much smaller than the send part. To achieve higher overlap and make the last receive as small as possible, we begin with a large subtask size. Then we exponentially decrease the subtask size as shown in Figure 4.6. With the base value a, the number of subtasks is:

$$m = \lceil \log_a(D_s \times (a-1)/ds_{min}+1) \rceil. \tag{4.30}$$

In step 1 and 2, we have:

$$\begin{aligned} x_{s1} &= ds_{min} \times a^{m-1}, \\ x_{s2} &= ds_{min} \times a^{m-2}. \\ x_{r1} &= x_{r2} = 0. \end{aligned} \tag{4.31}$$

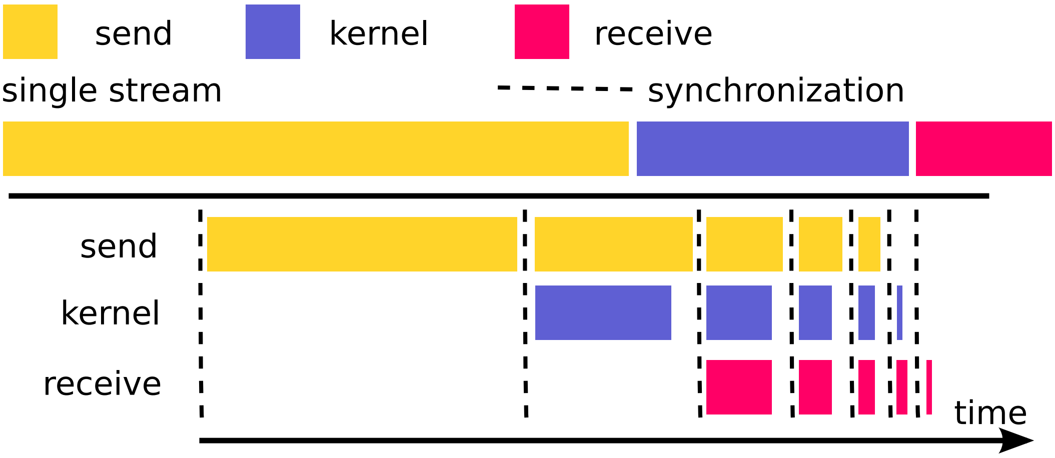


Figure 4.6: TPSM for Data Transfer Bound and Send Heavy in Single GPU Architecture

x_{s1}: the data size to send in step 1;
x_{s2}: the data size to send in step 2;
x_{r1}: the data size to receive in step 1;
x_{r2}: the data size to receive in step 2.
In steps $m+1$ and $m+2$,we have:

$$\begin{aligned} x_{s(m+1)} = x_{s(m+2)} = 0, \\ x_{r(m+1)} = a \times dr_{min}, \\ x_{r(m+2)} = dr_{min} = ds_{min} \times D_r/D_s. \end{aligned} \tag{4.32}$$

$x_{s(m+1)}$: the data size to send in step $m+1$;
$x_{s(m+2)}$: the data size to send in step $m+2$;
$x_{r(m+1)}$: the data size to receive in step $m+1$;
$x_{r(m+2)}$: the data size to receive in step $m+2$.
For step i $(3 \leq i \leq m)$, we have the data size to send and receive for one GPU as follow:

$$\begin{aligned} x_{si} = ds_{min} \times a^{m-i}, \\ x_{ri} = dr_{min} \times a^{m+2-i}. \end{aligned} \tag{4.33}$$

With the data size of send and receive in each step, we can use the time optimal data transfer model to calculate the minimum time cost of data transfer in step

$1, 2, m+1, m+2$ and $i(3 \le i \le m)$ as follow:

$$\begin{aligned} t_1 &= x_{s1}/S_S = ds_{min} \times a^{m-1}/S_S \\ t_2 &= x_{s2}/S_S = ds_{min} \times a^{m-2}/S_S \\ t_{m+1} &= x_{r(m+1)}/R_R = a \times dr_{min}/R_R \\ t_{m+2} &= x_{r(m+2)}/R_R = dr_{min}/R_R \\ dr_{min} &= ds_{min} \times D_r/D_s \end{aligned} \tag{4.34}$$

It is similar to the calculation of bi-directional data transfer in equation (4.18). The time cost of bi-directional data transfer with data send and receive data size (x_{si}, x_{ri}) can be calculated as follow:

$$t_i = H(x_{si}, x_{ri}) = \begin{cases} x_{si}/S_{SR} + (x_{ri} - (x_{si}/S_{SR}) \times R_{SR})/R_R \\ \qquad\qquad if \quad x_{si}/S_{SR} \le x_{ri}/R_{SR}, \\ \\ x_{ri}/R_{SR} + (x_{si} - (x_{ri}/R_{SR}) \times S_{SR})/S_S \\ \qquad\qquad if \quad x_{si}/S_{SR} > x_{ri}/R_{SR}. \end{cases} \tag{4.35}$$

t_i: the time cost of data transfer for step i;
dr_{min}: the minimum data size to receive in the last step.
Then we can use the following equation to calculate the total time cost:

$$T_{total} = (m+2) \times t_{str} + \sum_{i=1}^{m+2} t_i = W(a). \tag{4.36}$$

t_{str}: the time cost of one synchronization between GPUs;
t_i: the time cost of data transfer in step i $(3 \le i \le m)$.
Since $W(a)$ is not differentiable on many points, we approximately minimize it by a heuristics method. We found that the solution base value a is normally close to the ratio of the time cost of data transfer and the time cost of kernel execution. Therefore, we can use algorithm 1 to find the solution.

Notice that with larger step number, we can have a better base value a close to the optimal one. Meanwhile, larger step number means more calculation and more time cost. Therefore, we suggest to set n to $10 \sim 30$.

Proof of Asymptotic Optimality

The scheduling length of our sub-method is as follow:

$$\begin{aligned} T_{total} &= dr_{min} \times (a+1)/R_R + H(D_s, D_r - dr_{min} \times (a+1)) + t_{str} \times (m+2) \\ &\le dr_{min} \times (a+1)/R_R + H(D_s, D_r) + t_{str} \times O(log_a(D_s)). \end{aligned} \tag{4.37}$$

As it is data transfer bound and send heavy, we assume that $D_s = p \times D_r$ where $p >> 1$. Therefore, the optimal scheduling length is $T_{opt} = H(D_s, D_r) = D_r \times (1/R_{SR}+p/S_S-S_{SR}/(R_{SR}S_S))$. As a and $H(p, 1)$ can be considered to be constant, when the receive data size is infinite, then we have:

$$\lim_{D_r \to \infty} T_{total}/T_{opt} \leq 1 + \lim_{D_r \to \infty} O(log_a(p \times D_r))/(D_r \times H(p, q)) = 1. \tag{4.38}$$

Therefore, the sub-method of TPSM for data transfer bound and send heavy is asymptotic optimal.

4.2.2.2 Receive Heavy

For data transfer bound and receive heavy applications, the the send part is much smaller than the receive part. Notice that we will adopt a solution which is not symmetric with TPSM for data transfer bound and send heavy applications. This is because for we can send any data size of send data while we can only receive the

Algorithm 1: Algorithm for a good base value a

```
Input: $n$: the number of search steps,
        $b$: the ratio of the time cost of data transfer and the time cost of
kernel execution,
        $W(a)$: the function to calculate the total time cost from $a$
Output: $a$: the base value $a$
$a_1 = \min(1, b/2), a_2 = b, y_3 = 2b$;
for (i=0; i<n; i++) do
    calculate $W(a_1)$, $W(a_2)$, $W(a_3)$;
    if ($W(a_1) < W(a_2) \& W(a_1) < W(a_3)$) then
        $a_1 = a_2$;
        $a_2 = (a_1 + a_3)/2$;
        continue;
    if ($W(a_2) < W(a_1) \& W(a_2) < W(a_3)$) then
        $a_2 = (a_1 + a_3)/2$;
        continue;
    if ($W(a_3) < W(a_1) \& W(a_3) < W(a_2)$) then
        $a_3 = a_2$;
        $a_2 = (a_1 + a_3)/2$;
        continue;
return $a_2$;
```

data size of receive data that the corresponding kernel execution is finished. The send data is independent from kernel execution while receive data is dependent on kernel execution. If use solely exponential increase or decrease method, there might exist case in some steps that there is no data send and the kernel is longer than the time cost of data receive. We can not receive the data which is not generated yet.

We divide the scheduling into three phases:(1)*send phase*, (2)*middle phase* and (3)*receive phase* as shown in Figure 4.7. To minimize the total time cost, we should ensure the time cost of data transfer is not shorter than the time cost of kernel execution in each step of each phase while we also should overlap the send part and receive part as much as possible. Therefore, we begin with a small subtask size and exponentially increase the subtask size in the send phase. As a transitional phase, there is only one step in the middle phase. In this step, we have to finish all the remaining data send and receive all the output data generated by the kernel execution in the last step of the send phase. Then we launch a sub-kernel with the time cost of kernel execution equal to the time cost of the data transfer in this step. As it is data transfer bound and receive heavy application, the time cost of the remaining data receive is not shorter than the time cost of the remaining kernel execution in the receive phase. Therefore, we begin with a small subtask size to make the first kernel small and then exponentially increase the subtask size.

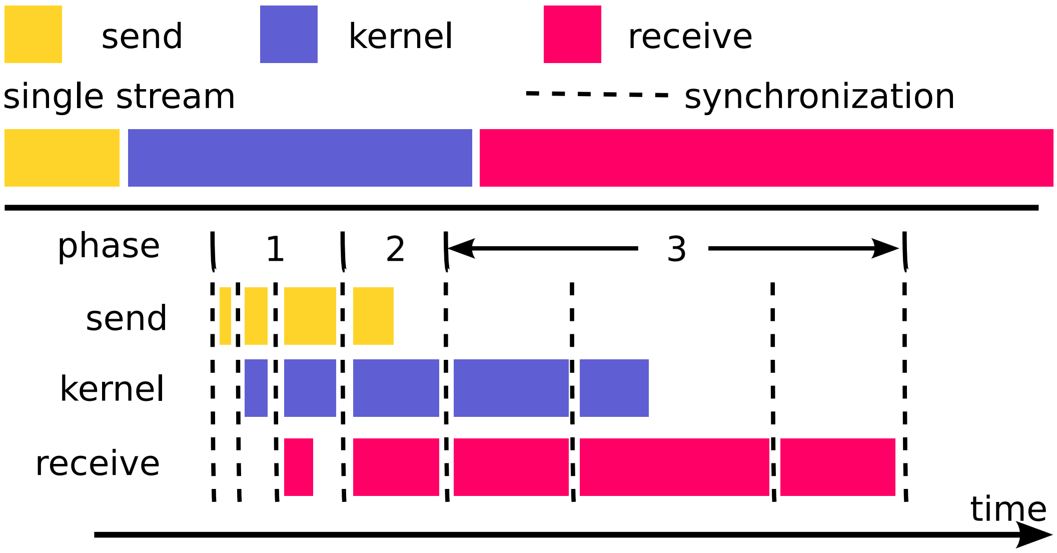


Figure 4.7: TPSM for Data Transfer Bound and Receive Heavy in Single GPU Architecture

In the send phase, assume the base value of the increase is a, then the data size of send and receive for each GPU is:

$$\begin{aligned} x_{si} &= ds_{min} \times a^{i-1}, \\ x_{ri} &= ds_{min} \times a^{i-3} \times D_r/D_s. \end{aligned} \tag{4.39}$$

Then we can use equation (4.35) to calculate the time cost of data transfer in step i. As the time cost of kernel execution in step i is:

$$tk_i = x_{s(i-1)}/D_s \times T_k = ds_{min} \times a^{i-2} \times T_k/D_s. \tag{4.40}$$

To minimize the total time cost, we should ensure $tk_i \leq t_i$ in each step of the send phase. Then we can have the following set of base a that matches the condition:

$$\begin{aligned} &S = \{a | a \times T_k/D_s \leq F(a)\} \\ &where\ F(a) = \begin{cases} a^2(R_R - R_{SR})/(S_{SR}R_R) + D_r/(D_sR_R) \\ \qquad\qquad if \quad a^2 \leq D_rS_{SR}/(D_sR_{SR}), \\ \\ a^2/S_S + (D_rS_S - D_rS_{SR})/(D_sR_{SR}S_S) \\ \qquad\qquad if \quad a^2 > D_rS_{SR}/(D_sR_{SR}). \end{cases} \end{aligned} \tag{4.41}$$

Then we choose $a = \max\{a|a \in S\}$ as the optimal base value for the scheduling in the send phase. As the application is data transfer bound and send heavy, there exits a solution a that matches the equation.

In the middle phase, we will send all the remaining send data and receive all the output data generated by the kernel execution of the last step in the send phase. The data size of sending and receiving in the middle phase is as follow:

$$\begin{aligned} d_s &= D_s - ds_{min} \times (a^P - 1)/(a-1), \\ d_r &= (ds_{min} \times D_r \times a^{P-2})/D_s. \end{aligned} \tag{4.42}$$

P: the number of subtasks in the send phase which can be calculated with equation (4.30).

The time cost of the bi-directional data transfer t_{mid} can be calculated with equation (4.35). To ensure the data transfer part is the main part in the middle phase, we set the subtask size to $t_{mid} \times D_s/T_k$.

As the remaining receive part is not shorter than the remaining kernel execution part, we receive all the data generated from the middle phase in the first step of the receive phase as shown in Figure 4.7. As the kernel time in the middle phase is t_{mid}, the time cost of data receiving is:

$$t_{r1} = (t_{mid} \times D_r)/(T_k \times R_R). \tag{4.43}$$

t_{r1}: the time cost of kernel execution in the first step of the receive phase.

In the first step of the receive phase, we set the kernel execution equal to the data receiving. Then the subtask size in the first step of the receive phase is as follow:

$$x_1 = t_{r1} \times D_s/T_k \tag{4.44}$$

In step i of the receive phase ($i \geq 2$), we receive all the output data generated from the kernel execution in step $i-1$. The time cost of data receiving in step i of the receive phase is:

$$t_{ri} = (t_{r(i-1)} \times D_r)/(T_k \times R_R). \tag{4.45}$$

We also set the time cost of kernel execution equal to the data receiving. Therefore, the subtask size in step i of the receive phase is:

$$x_i = t_{ri} \times D_s/T_k. \tag{4.46}$$

Finally, we can finish all the kernel execution in the penultimate step and finish all the data receiving in the last step. By this way, we can hide the kernel execution and use as fewer synchronization operation as possible to minimize the total time cost.

Proof of Asymptotic Optimality

As it is data transfer bound and receive heavy, we assume $D_r = p \times D_r$ where $p >> 1$. The number of synchronization operations in the receive phase can be considered as constant e. The scheduling length of our sub-method is as follow:

$$T_{total} = ds_{min}(a+1)/S_S+H(D_s-ds_{min}(a+1), D_r)+t_{str}(log_a(D_s(a-1)/ds_{min}+1)+1+e). \tag{4.47}$$

Similar to the proof of asymptotic optimality for data transfer bound and send heavy, we have:

$$T_{total} \leq H(D_s, D_r) + t_{str} \times O(log_a(D_s)). \tag{4.48}$$

The optimal scheduling length is $T_{opt} = H(D_s, D_r)$ and a can be considered as constant. When the input size is infinite, then we have:

$$\lim_{D_s \to \infty} T_{total}/T_{opt} \leq 1 + \lim_{D_s \to \infty} t_{str} \times O(log_a(D_s))/H(D_s, D_r) = 1. \tag{4.49}$$

Therefore, the sub-method of TPSM for data transfer bound and receive heavy in single GPU architecture is asymptotic optimal.

4.2.2.3 General Heavy

For data transfer bound and general heavy applications, both the data send and receive part are not short. Therefore, the key is to overlap the data send and receive

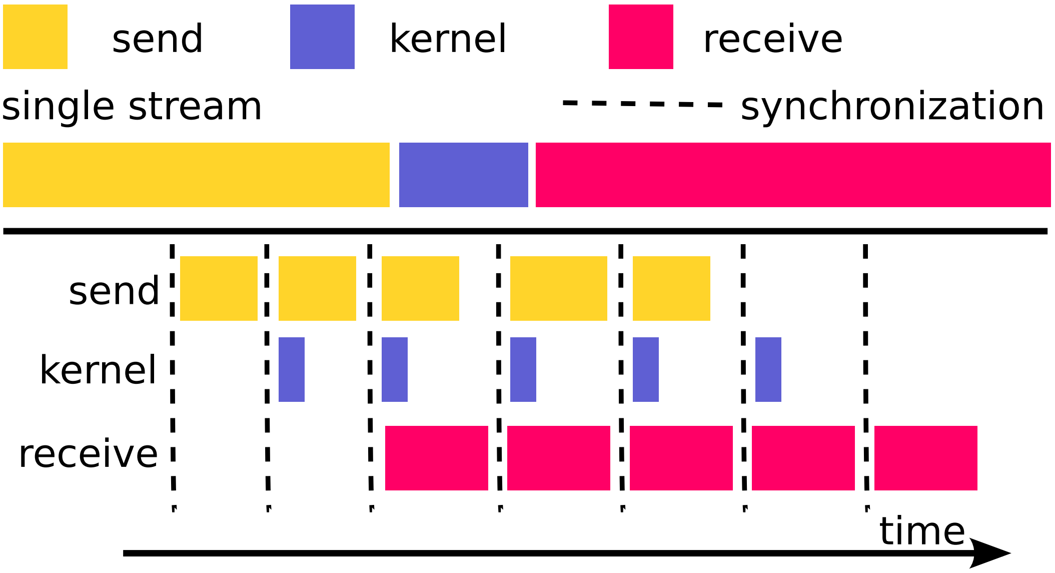


Figure 4.8: TPSM for Data Transfer Bound and General Heavy in Single GPU Architecture

part as much as possible. We set the subtasks to equal size for easy scheduling and use the time optimal data transfer algorithm to schedule the data transfer as shown in Figure 4.8.

Suppose that the number of subtasks is m, then we have the data size of send and receive for one GPU in step i as follow:

$$x_{si} = D_s/m, \qquad x_{ri} = D_r/m. \tag{4.50}$$

By using time optimal data transfer model, we can calculate the optimal time cost of data transfer for all GPUs in each step as follow($3 \leq i \leq m$):

$$\begin{aligned} t_1 = t_2 = D_s/(mS_S), \\ t_{m+1} = t_{m+2} = D_r/(mR_R), \\ t_i = H(D_s/m, D_r/m) \quad (function\ H()\ is\ defined\ in\ equation\ (4.35)). \end{aligned} \tag{4.51}$$

With the time cost of each step, we can calculate the total time cost as follow:

$$\begin{aligned} T_{total} = (m+2)t_{str} + \sum_{i=1}^{m+2} t_i \\ = 2D_s/(mS_S) + 2D_r/(mR_R) + (m-2)H(D_s/m, D_r/m) + t_{str}(m+2) = W(m). \end{aligned} \tag{4.52}$$

The total time cost of the application is a function of the number of subtask. Then we calculate the derivative of the function $W(m)$ to get the m for the scheduling which makes the total time cost minimum as follow:

$$m = \sqrt{2(D_s/S_S + D_r/R_R - H(D_s, D_r))/t_{str}}. \tag{4.53}$$

Proof of Asymptotic Optimality

As the application is data transfer bound and general heavy, we assume $D_r = pD_s$ (p is constant) and $c = \sqrt{2(1/S_S + p/R_R - H(1,p))/t_{str}}$. As $m = c\sqrt{D_s}$, then we can calculate the scheduling length as follow:

$$\begin{aligned} T_{total} = (c\sqrt{D_s} + 2)t_{str} + 2\sqrt{D_s}/(cS_S) + 2\sqrt{pD_s}/(cR_R) + (D_s - 2/(c\sqrt{D_s}))H(1,p) \\ \leq D_s H(1,p) + O(\sqrt{D_s}). \end{aligned} \tag{4.54}$$

As the optimal scheduling length is $T_{opt} = H(D_s, D_r) = D_s \times H(1,p)$ and $H(1,p)$ is constant, then we have:

$$\lim_{D_s \to \infty} T_{total}/T_{opt} \leq 1 + \lim_{D_s \to \infty} O(\sqrt{D_s})/(D_s \times H(1,p)) = 1. \tag{4.55}$$

Therefore, our sub-method for data transfer bound and general heavy in single GPU architecture is asymptotic optimal.

4.3 TPSM for Symmetric Multiple GPUs Architecture

For multiple GPU platform (multiple GPUs in one node), the bandwidths are shared by all GPU devices and the send and receive bandwidth of one GPU can be greatly affected by the states of other GPU devices as shown in Table 4.2. $G0$ and $G1$ are the GPU device ID and $N, S, R, S\&R$ are four data transfer states which will be introduced in this section. Therefore, local optimal of data transfer in each GPU does not mean global optimal of data transfer for the whole system. To minimize the total time cost of data transfer between the host and the device, it is important to take the impact between GPUs into consideration. For symmetric multiple GPU architecture (install the same GPUs rather different types of GPUs), we allocate the same load for each GPU so that the size of data transfer of different GPUs are equal.

4.3.1 Time Optimal Data Transfer Algorithm for Symmetric Multiple GPUs Architecture

In this section, we introduce a time optimal data transfer algorithm for multiple GPU platform with symmetric architecture. We also build a model for the algorithm to predict the optimal communication time cost between the host and the device with a fixed communication data size.

Table 4.2: Bandwidth of Dual Tesla C2075 under Different States

	G0-N	G0-S	G0-R	G0-S&R
G1-N	-	$S0 = 5.99$	$R0 = 6.48$	$S0 = 3.86$ $R0 = 4.2$
G1-S	$S1 = 5.99$	$S0 = 5.11$ $S1 = 5.11$	$R0 = 4.26$ $S1 = 3.79$	$S0 = 2.9$ $R0 = 2.99$ $S1 = 2.99$
G1-R	$R1 = 6.48$	$S0 = 3.79$ $R1 = 4.26$	$R0 = 3.38$ $R1 = 3.38$	$S0 = 2.41$ $R0 = 2.42$ $R1 = 2.76$
G1-S&R	$S1 = 3.86$ $R1 = 4.2$	$S0 = 2.99$ $S1 = 2.9$ $R1 = 2.99$	$R0 = 2.76$ $S1 = 2.41$ $R1 = 2.42$	$S0 = 2.05$ $R0 = 2.06$ $S1 = 2.05$ $R1 = 2.06$

4.3.1.1 Definition and Notations

In the multiple GPU system, there are two kind of states:*device state* and *system state*. *Device state* is the communication state of single GPU. *System state* is the communication state of all the GPU devices in the system. Therefore, the system state includes the device states of all GPU devices. We define four states for device state as follow:

- N: no communication;
- S: data sending from host to device;
- R: data receiving from host to device;
- $S\&R$: simultaneous data sending and receiving.

With n GPU devices where D_i is the device state of the ith GPU, we define Γi_q as the send($\Gamma = S$) or receive($\Gamma = R$) bandwidth of ith GPU under the system state of $q \in U$, where U is defined as follow:

$$U = \{D_0/D_1/\cdots/D_{n-1} \mid D_i \in \{N, S, R, SR\}, 0 \le i \le n-1\}. \qquad (4.56)$$

q: the system state;
U: the union of the system state;
D_i: the device state of the ith GPU;
n: the total number of GPU devices in the system.
And we define Γ_q to present the total send or receive bandwidth of all GPU devices

which means:

$$\Gamma_q = \sum_{i=0}^{n-1} \Gamma i_q. \tag{4.57}$$

For example, $S3_{S/S/SR/SR}$ means the sending bandwidth of $GPU3$ under the system state of $GPU0$ only sending, $GPU1$ only sending, $GPU2$ simultaneously sending and receiving and $GPU3$ simultaneously sending and receiving.

4.3.1.2 Time Optimal Data Transfer Algorithm

The process of data communication between the host and the device can be described by a set of system states and the corresponding time cost in each system state. We denote the scheduling as T_Q. For example, a scheduling T_Q, where $Q = \{S/S, R/R\}$, means we begin with two GPUs sending data for T_{q1} and then make two GPUs receiving data for T_{q2}. Therefore, the scheduling length of T_Q is equal to the sum of the scheduling time in each system state as follow:

$$|T_Q| = \sum_{i=1}^{m} T_{q_i}, \quad q_i \in Q \subseteq U. \tag{4.58}$$

Q: a set of system states used in the process of data communication;
m: the number of system states in Q;
T_{q_i}: the scheduling time of the system state q_i in Q.
Therefore, the data size of send and receive for $GPUi$ is:

$$condition1: \begin{cases} X_{s_i} = \sum_{q_i \in U} T_{q_i} \times Si_{q_i}, \ T_{q_i} \geq 0, \\ \\ X_{r_i} = \sum_{q_i \in U} T_{q_i} \times Ri_{q_i}, \ T_{q_i} \geq 0. \end{cases} \tag{4.59}$$

X_{s_i}: the data size of sending for $GPUi$;
X_{r_i}: the data size of receiving for $GPUi$;
Si_{q_i}: the send bandwidth of $GPUi$ under system state q_i;
Ri_{q_i}: the receive bandwidth of $GPUi$ under system state q_i.
As the system is symmetric architecture, we assume that the data size of sending and receiving of each GPU device is equal. Therefore, the total data size of sending and receiving of all GPUs should be:

$$condition2: \begin{cases} X_s = \sum_{i=0}^{n-1} X_{s_i} = n \times X_{s_i}, \ (0 \leq i \leq n-1), \\ \\ X_r = \sum_{i=0}^{n-1} X_{r_i} = n \times X_{r_i}, \ (0 \leq i \leq n-1). \end{cases} \tag{4.60}$$

X_s: the total data size of sending for the system;
X_r: the total data size of receiving for the system.

A *time optimal* scheduling is a scheduling which gives the minimum scheduling length among all scheduling that satisfy the condition 1 and 2 with a fixed (X_s, X_r). The time optimal scheduling is a problem of linear program which is determined by the input of (X_s, X_r). There could be more than one optimal scheduling.

We can find a time optimal scheduling that gives symmetric time cost for symmetric states. Therefore, we consider the time cost of each sate between symmetric states is equal. In this case, we can reduce the number of system states by reducing the duplicated symmetric system states. For example, S/R (GPU0 is sending and GPU1 is receiving) and R/S (GPU0 is receiving and GPU1 is sending) are duplicated. These two system states can be described just by one S/R. There are 256 system states for 4 GPU system. By reducing the symmetric system states, only 35 system states are useful for the scheduling. From data size aspect, we denote transferred data by a scheduling T_Q as X_s and X_r which can be illustrated as a point (X_s, X_r) in two-dimensional coordinate system. We can find out the time optimal scheduling by solving its dual problem. The dual problem of time optimal scheduling problem is data size optimal scheduling problem which is defined as follows. As shown in Figure 4.9, we define a scheduling T_Q as a data size optimal scheduling, which maximizes transferred data sizes in a fixed time, if there is no scheduling T'_Q

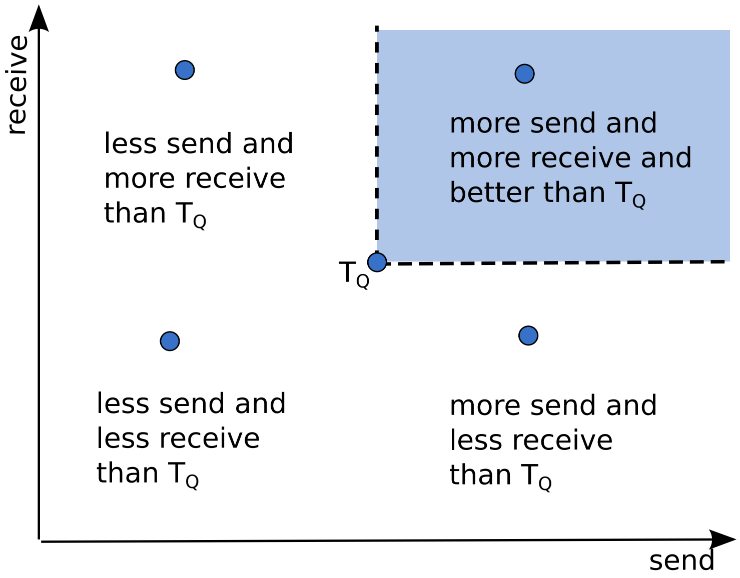

receive
less send and more receive than T_Q
more send and more receive and better than T_Q
T_Q
less send and less receive than T_Q
more send and less receive than T_Q
send

Figure 4.9: Data Size Optimality Problem

that satisfies condition 1 and 2 and one of the following conditions:

$$\begin{cases} (1)\ X_s(T'_Q) > X_s(T_Q)\ \mathit{and}\ X_r(T'_Q) > X_r(T_Q); \\ \\ (2)\ X_s(T'_Q) > X_s(T_Q)\ \mathit{and}\ X_r(T'_Q) = X_r(T_Q); \\ \\ (3)\ X_s(T'_Q) = X_s(T_Q)\ \mathit{and}\ X_r(T'_Q) > X_r(T_Q). \end{cases} \tag{4.61}$$

Next we discuss the solution of data size optimal scheduling problem. When only one system state is used, say state p, then the transferred data corresponds to a point $(T \times S_q, T \times R_q)$. When two system states p and q are used, then we can have the following equation as the scheduling length is fixed:

$$T_p = \alpha T,\ T_q = (1-\alpha)T,\ 0 \leq \alpha \leq 1. \tag{4.62}$$

T: the fixed total time cost of the scheduling.
Then we have the transferred data sizes:

$$\begin{aligned} X_s = T_p \times S_p + T_q \times S_q = \alpha T \times S_p + (1-\alpha)T \times S_q, \\ X_r = T_p \times R_p + T_q \times R_q = \alpha T \times R_p + (1-\alpha)T \times R_q. \end{aligned} \tag{4.63}$$

Thus we have the total data size of send and receive as follow:

$$(X_s, X_r) = \alpha(T \times S_p, T \times R_p) + (1-\alpha)(T \times S_q, T \times R_q). \tag{4.64}$$

Therefore, the transferred data of the scheduling in the set of data size optimal scheduling corresponds to the line segment that connects point p and q. We call such a scheduling a convex mixture of states p and q.

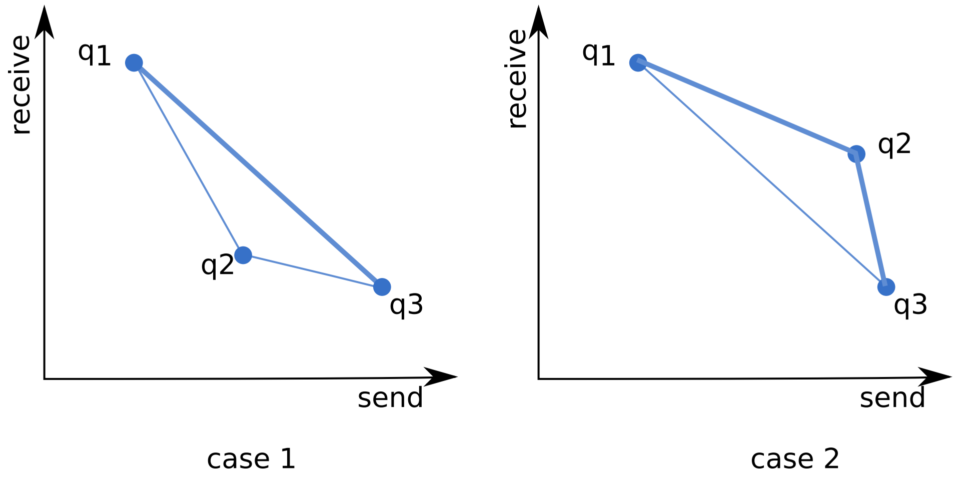


Figure 4.10: Convex Mixture

Let us consider three system states as shown in Figure 4.10. In *case* 1, the combination of q_1 and q_3 is better than any combination of q_1 and q_2 and any combination of q_2 and q_3. Therefore, the transferred data of the scheduling in the set of data size optimal scheduling corresponds to the line segment q_1q_3 in *case* 1. Similarly in *case* 2, the combination of q_1 and q_3 is worse than both the combination of q_1 and q_2 and the combination of q_2 and q_3. Thus the transferred data of the scheduling in the set of data size optimal scheduling corresponds to the line segments q_1q_2 and q_2q_3 in *case* 2.

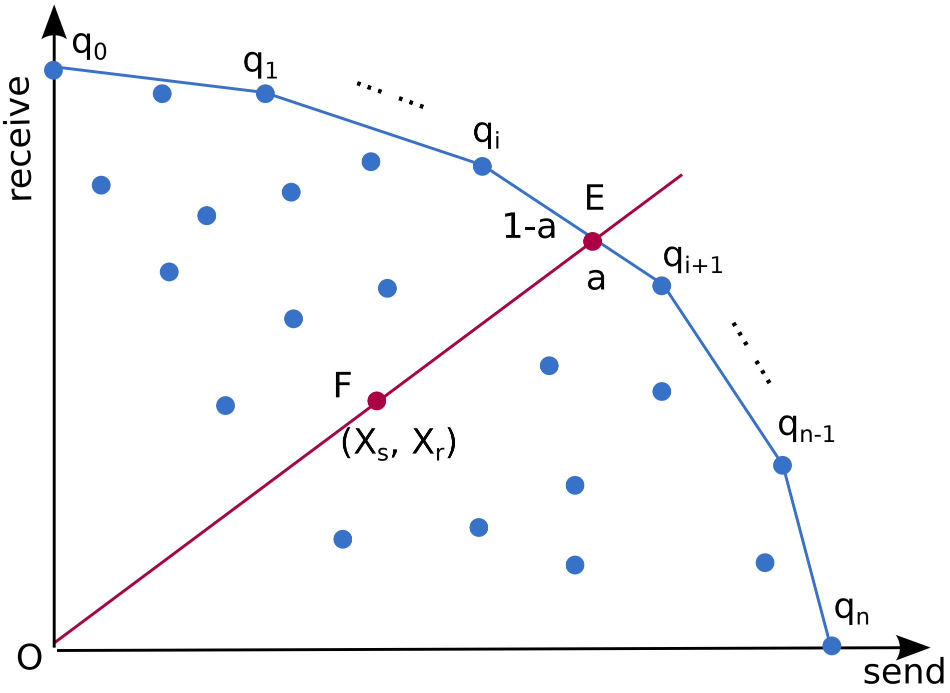


Figure 4.11: Upper-right Convex Hull Data Transfer Size with Scheduling Length T

Considering more system states as shown in Figure 4.11, we can find that any scheduling is not better than the scheduling on the upper right convex hull. The data size optimal scheduling is a combination of only two system states. By changing the scheduling time T, the upper right convex hull linearly scales about the origin. Therefore, with a fixed (X_s, X_r), we can draw a line that passes the origin and point F which will intersect with upper right convex hull at point E as shown in Figure 4.11. Let q_i and q_{i+1} be the end points of the crossing line segment of the convex hull. Then the time optimal scheduling is the scheduling at the point E which is a convex mixture of system states q_i and q_{i+1}. The time cost of the time optimal scheduling is:

$$T_{optimal} = T \times |OF|/|OE|. \tag{4.65}$$

$|OF|$: the length of the line segment of OF;
$|OE|$: the length of the line segment of OE.
The scheduling time in the system state q_i is $a \times T_{optimal}$ while the scheduling time in the system state q_{i+1} is $(1-a) \times T_{optimal}$ $(0 \le a \le 1)$.

Next we will introduce a model to predict the minimum time cost with a fixed data size of send and receive based on the time optimal transfer algorithm. As shown in Figure 4.11, suppose that the two used system states are p and q. The coordinates of the system state p and q are as follow:

$$\begin{aligned} (x_p, y_p) &= (T \times B_{ps}, T \times B_{pr}), \\ (x_q, y_q) &= (T \times B_{qs}, T \times B_{qr}). \end{aligned} \tag{4.66}$$

B_{ps}: the total bandwidth of data sending of all GPUs in system state p;
B_{pr}: the total bandwidth of data receiving of all GPUs in system state p;
B_{qs}: the total bandwidth of data sending of all GPUs in system state q;
B_{qr}: the total bandwidth of data receiving of all GPUs in system state q.
We also can use the following equation to calculate the x coordinate of point E:

$$x_e = (x_q \times y_p - x_p \times y_q) \times X_s/(X_r \times (x_q - x_p) + X_s \times (y_q - y_p)). \tag{4.67}$$

Suppose that T is 1 second and take equation (4.66) and (4.67) into equation (4.65). Then we can get the scheduling length of T_{pq} using system states p and q:

$$\begin{aligned} T_{pq} = X_s/x_e = (X_r(B_{qs} - B_{ps}) + X_s(B_{qr} - B_{pr})) \\ /(B_{qs} \times B_{pr} - B_{ps} \times B_{qr}) = F_{pq}(X_s,\ X_r). \end{aligned} \tag{4.68}$$

For n points $p_1, p_2, ..., p_n$ as shown in Figure 4.11, suppose that their coordinates are $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$. As it is upper-right convex hull so that we have $x_1 < x_2 < ... < x_n$ and $y_1 > y_2 > ... > y_n$. Therefore we can calculate the length of the time optimal scheduling with an input of (X_s, X_r):

$$T_{opt} = G(X_s, X_r) = \begin{cases} F_{p_1p_2}(X_s, X_r) \\ \qquad if\ y_2/x_2 \le X_r/X_s < y_1/x_1; \\ ...... \\ F_{p_ip_{i+1}}(X_s, X_r) \\ \qquad if\ y_{i+1}/x_{i+1} \le X_r/X_s < y_i/x_i; \\ ...... \end{cases} \tag{4.69}$$

$G(X_s, X_r)$: the minimum time cost to send and receive a data size of (X_s, X_r).

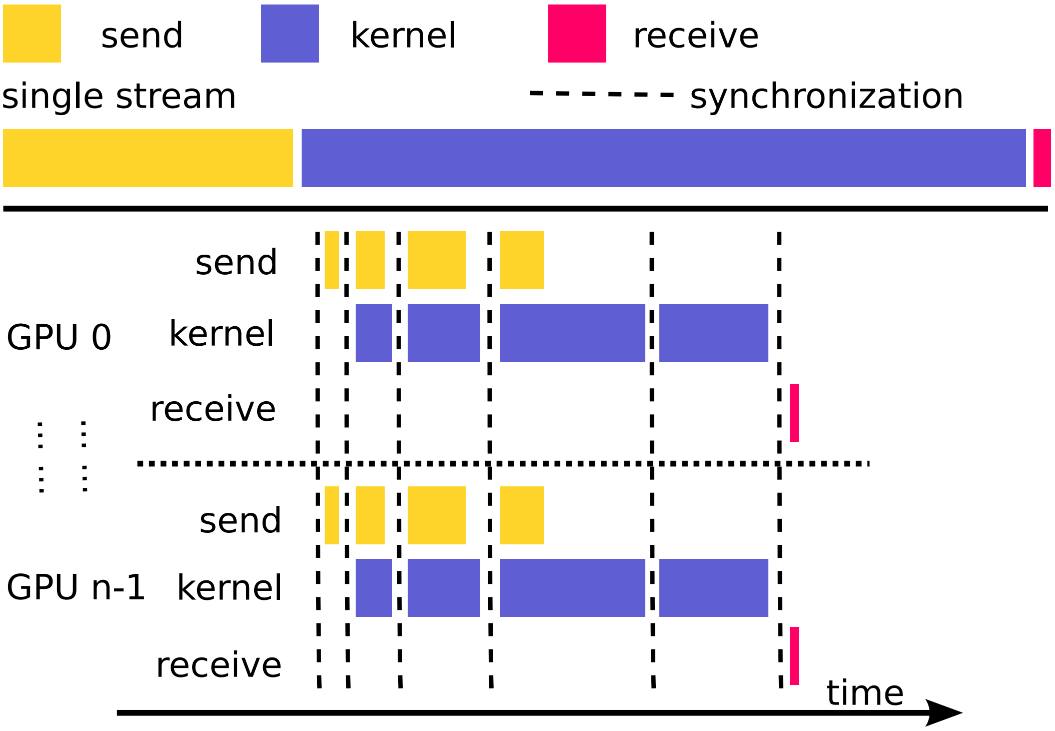


Figure 4.12: TPSM for Kernel Bound and Send Heavy in Symmetric Multiple GPU Architecture

For example, the length of the time optimal scheduling with an input of (X_s, X_r) in dual Tesla C2075 platform is as follow (Table 4.2):

$$T_{opt} = G(X_s, X_r) = \begin{cases} 0.1417x_r - 0.0978x_s & when\ x_r/x_s \leq 0.5076 \\ \\ 0.1479x_r - 0.0947x_s & when\ x_r/x_s > 0.5076 \end{cases} \tag{4.70}$$

4.3.2 TPSM

For symmetric architecture, we use symmetric scheduling method for all GPUs. The TPSM for symmetric architecture is based on TPSM for single GPU. In symmetric architecture, the load in each GPU is equal size. Notice that our TPSM is not a theoretically optimal solution but a heuristic solution.

4.3.2.1 TPSM for Kernel Bound and Send Heavy

For kernel bound and send heavy applications, the scheduling is similar to the TPSM of kernel bound and send heavy in single GPU case. We average allocate the application on each GPU. The receive part is so small that we can receive all the data in the last step. To make the first send as small as possible, we set the first subtask size to a minimum size which can just keep all threads working and then exponentially increase the subtask size as shown in Figure 4.12.

Suppose that the base value of the exponential increase is a and the send data size in step 1 is ds_{min} in each GPU. Then the send data size in step i is $ds_{min} \times a^{i-1}$. We can calculate ts_i and tk_i in step i in each GPU as follow:

$$ts_i = ds_{min} \times a^{i-1}/S_S, \tag{4.71}$$

$$tk_i = (T_k/n) \times ds_{min} \times a^{i-2}/(D_s/n). \tag{4.72}$$

D_s: the total data size to send from host to device for all GPUs;
T_k: the total time cost of kernel execution with single GPU;
ds_{min}: minimum send data size to feed all threads working in each GPU;
S_S: the send bandwidth of one GPU in the system state of all GPUs sending, equal to $Si_{S/S/..}$ $(i = 0, 1, ..)$ (notice that there are only data sending in all steps except the last step);
ts_i: the time cost of data sending in step i in each GPU;
tk_i: the time cost of kernel execution in step i in each GPU;
n: the number of GPU devices in the system.

The total time cost is the sum of kernel execution, first send, last receive and the synchronization overhead. With fixed first send, last receive and kernel execution, the base value a can affect the total time cost by affecting the synchronization overhead. With larger a, there will be less synchronization times. Therefore, the optimal base value a is the maximum value which can also make sure the time cost of kernel execution is not shorter than the time cost of data send in each step. Then we can calculate the a that match the condition as follow:

$$ds_{min} \times a^{i-1}/S_S \leq (T_k/n) \times ds_{min} \times a^{i-2}/(D_s/n), \tag{4.73}$$

$$a \leq S_S \times T_k/D_s. \tag{4.74}$$

Therefore, $a = S_S \times T_k/D_s$ is the optimal base value.

Proof of Asymptotic Optimality

The scheduling length of our sub-method is:

$$T_{total} = ds_{min}/S_S + T_k/n + D_r/(nR_R) + t_{str} \times log_a(D_s(a-1)/(nds_{min})+1). \tag{4.75}$$

D_r: the total data size to receive from device to host;
t_{str}: the time cost of one synchronization operation.

As the application is kernel bound and send heavy, the receive is small so that we assume it is constant. Because ds_{min} and t_{str} are also constant and D_s is proportional to T_k (assume $D_s = p \times T_k$ where p is constant), the scheduling length with our work is:

$$T_{total} \leq T_k/n + O(log_a(pT_k)) \tag{4.76}$$

When the input data size is infinite, the time cost of kernel is also infinite. As the optimal scheduling length is $T_{opt} = T_k/n$ and a is equal to $S_S T_k/D_s = S_S/p$ which is constant, then we have:

$$\lim_{T_k \to \infty} T_{total}/T_{opt} \leq 1 + \lim_{T_k \to \infty} O(log_a(pT_k))/(T_k/n) = 1. \tag{4.77}$$

Therefore, the sub-method of TPSM for kernel bound and send heavy is asymptotic optimal.

4.3.2.2 TPSM for Kernel Bound and Receive Heavy

For kernel bound and receive heavy applications, the send part is so small that we can send all the data in the first step. To make the last receive as small as possible, we set the first subtask size in a large size and then exponentially decrease the subtask size to make the last subtask size just feed all threads working as shown in Figure 4.13.

We set a minimum receive data dr_{min} in each GPU which feeds all threads can return results and suppose that the base value of the exponential decrease is a and the number of subtask is m. Then the receive data size in the step i in each GPU is $dr_{min} \times a^{m+2-i}$. We can calculate the time cost of kernel execution and data receive in the step i as follow:

$$tr_i = dr_{min} \times a^{m+2-i}/R_R, \tag{4.78}$$

$$tk_i = (T_k/n) \times dr_{min} \times a^{m+1-i}/(D_r/n). \tag{4.79}$$

D_r: the total data size to receive from device to host for all GPUs;
R_R: the receive bandwidth of one GPU in the system state of all GPUs receiving, equal to $Ri_{R/R/..}(i = 0, 1, ..)$;
dr_{min}: minimum receive data size to feed all threads working in each GPU;
tr_i: the time cost of data receiving in step i in each GPU.

Similarly to the case of kernel bound and send heavy, the base value a can affect the total time cost by affecting the synchronization times. Therefore, the optimal base value a should be the maximum value while make sure the time cost of kernel execution is not shorter than the time cost of data receive in each step. Then we can calculate the base value a that matches the condition as follow:

$$dr_{min} \times a^{m+2-i}/R_R \leq (T_k/n) \times dr_{min} \times a^{m+1-i}/(D_r/n), \tag{4.80}$$

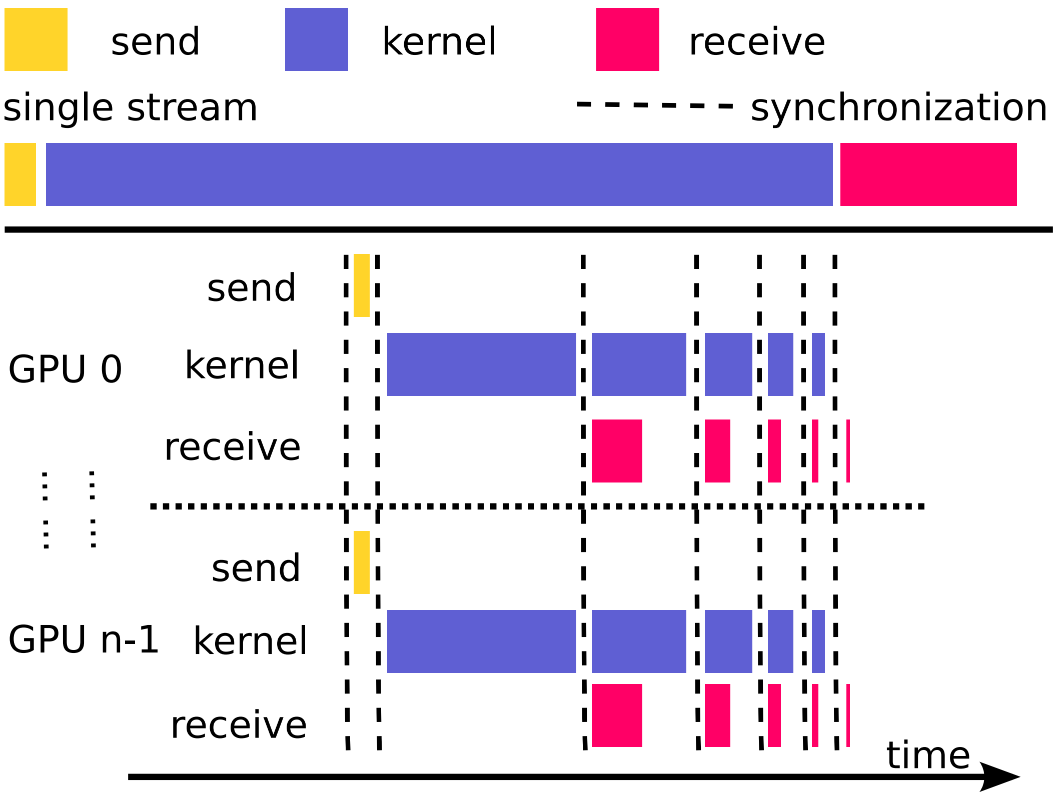


Figure 4.13: TPSM for Kernel Bound and Receive Heavy in Symmetric Multiple GPU Architecture

$$a \leq R_R \times T_k/D_r. \tag{4.81}$$

Then we can choose $a = R_R \times T_k/D_r$ as the optimal base value to schedule.

Proof of Asymptotic Optimality

The scheduling length with our work is:

$$T_{total} = D_s/(nS_S) + T_k/n + dr_{min}/R_R + t_{str}log_a(D_r(a-1)/(ndr_{min}) + 1). \tag{4.82}$$

As the application is kernel bound and receive heavy, the send part is small so that we assume D_s is constant. As dr_{min} is constant and D_r is proportional to T_k (assume $D_r = p \times T_k$ where p is constant), the scheduling length with our work is:

$$T_{total} \leq T_k/n + O(log_a(pT_k)). \tag{4.83}$$

When the input data size is infinite, the time cost of kernel is also infinite. As the

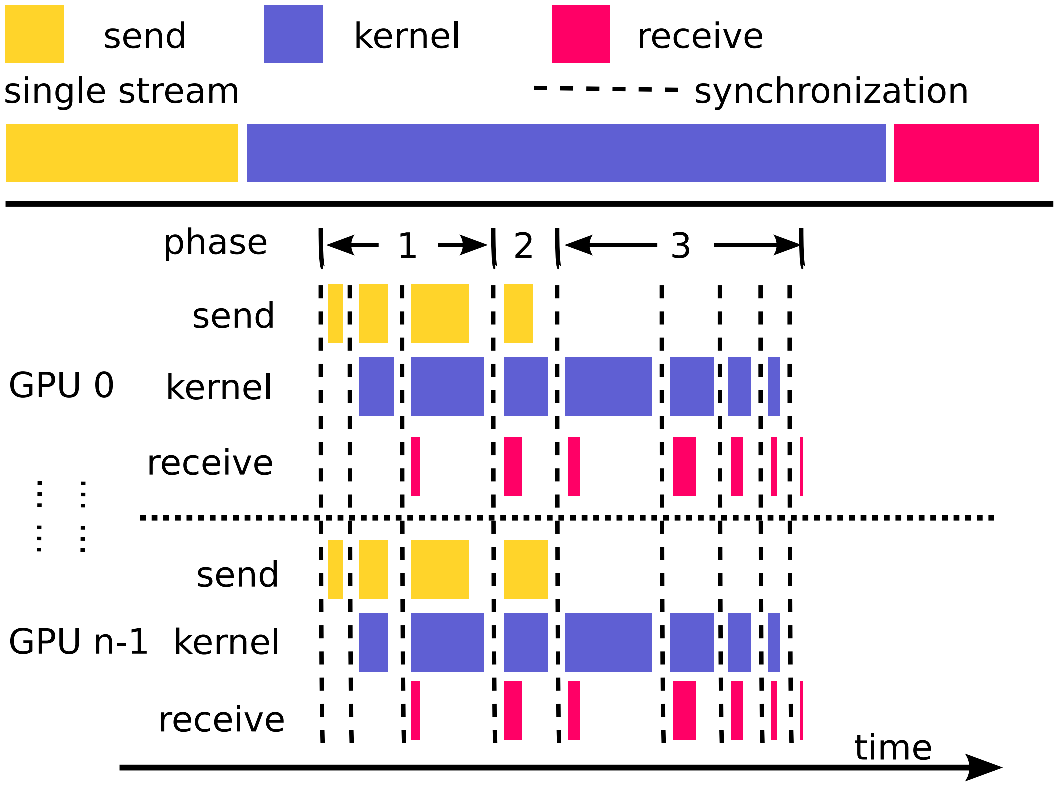


Figure 4.14: TPSM for Kernel Bound and General Heavy in Symmetric Multiple GPU Architecture

optimal scheduling length is $T_{opt} = T_k/n$ and a is constant, then we have:

$$\lim_{T_k \to \infty} T_{total}/T_{opt} \leq 1 + \lim_{T_k \to \infty} O(log_a(pT_k))/(T_k/n) = 1. \tag{4.84}$$

Therefore, the sub-method of TPSM for kernel bound and receive heavy is asymptotic optimal.

4.3.2.3 TPSM for Kernel Bound and General Heavy

For kernel bound and general heavy applications in symmetric architecture, both the send and receive parts are not small so that we should not send or receive all data in one step. We divide the scheduling into three phases as the same as in single GPU case: (1)*send phase*, (2)*middle phase* and (3)*receive phase* as shown in Figure 4.14. To minimize the total time cost, we should ensure the time cost of kernel execution is not shorter than the time cost of the bi-directional data transfer in each step of each phase. Therefore, we begin with a small subtask size and exponentially increase the subtask size in the send phase. In the middle phase, there is only one step in which we send all the left data and receive all the output data from the last step of the send phase. We set the time cost of kernel execution not shorter than the data transfer part by setting a suitable subtask size. Finally, we begin with a large subtask size and then exponentially decrease it to make the last receive as small as possible in the receive phase.

In the send phase, we assume the base value of the increase is a. Then the data send size in step i of the phase send is $ds_{min} \times a^{i-1}$. We can use the following equations to calculate the time cost of kernel execution in step i:

$$tk_i = (ds_{min} \times T_k/D_s) \times a^{i-2}. \tag{4.85}$$

The data receive size in step i in each GPU is $(ds_{min} \times D_r/D_s) \times a^{i-3}$. Then the time cost of data send and data receive in step i in each GPU is as follow when only consider uni-directional data transfer:

$$ts_i = ds_{min} \times a^{i-1}/S'_{SR}, \tag{4.86}$$

$$tr_i = ds_{min} \times D_r \times a^{i-2}/(D_s \times R'_{SR}). \tag{4.87}$$

S'_{SR}: the send bandwidth of one GPU in the system state of all GPU sending and receiving, equal to $Si_{SR/SR/..}(i = 0, 1, ..)$;
R'_{SR}: the receive bandwidth of one GPU in the system state of all GPU sending and receiving, equal to $Ri_{SR/SR/..}(i = 0, 1, ..)$.
As we also overlap the data send and data receive, then the time cost of the actual bi-directional data transfer should be as follow (Although we have time optimal data transfer algorithm, we do not use it in kernel bound applications. Because the time cost of kernel execution part is longer than the time cost of data transfer part. In each step except the first and the last step, the time cost of kernel execution is longer than the time cost of data transfer part. Reducing the time cost of data transfer part can not lead to the decrease of total time cost. On the contrary, using time optimal data transfer algorithm for kernel bound applications might increase the total time cost as it needs to some calculation at the beginning of each step):

$$tcom_i = \begin{cases} tr_i + (ds_{min} \times a^{i-1} - tr_i \times S'_{SR})/S_S \\ \qquad\qquad if \quad tr_i \le ts_i, \\ \\ ts_i + (ds_{min} \times a^{i-3} \times D_r/D_s - ts_i \times R'_{SR})/R_R \\ \qquad\qquad if \quad tr_i > ts_i. \end{cases} \tag{4.88}$$

$tcom_i$:the total communication time cost between the host and the device in step i of the send phase in each GPU.

In each step of the send phase, we need to keep $tcom_i \leq tk_i$. With equation (4.85) and (4.88), we can get A and B that match the condition $tcom_i \leq tk_i$ as follow:

$$\begin{cases} A = \{a|\beta - \sqrt{\beta^2 - \gamma} \leq a \leq \beta + \sqrt{\beta^2 - \gamma} \ \& \ a \geq \delta\}; \\ \\ B = \{a|\lambda - \sqrt{\lambda^2 - \mu} \leq a \leq \lambda + \sqrt{\lambda^2 - \mu} \ \& \ a \leq \delta\}; \\ \\ where\ \delta = \sqrt{D_r \times S'_{SR}/(D_s \times R'_{SR})}, \\ \qquad \beta = S_S \times T_k/2D_s, \\ \qquad \gamma = (S_S - S'_{SR}) \times D_r/(D_s \times R'_{SR}), \\ \qquad \lambda = T_k \times S'_{SR} \times R_R/(2D_s \times (R_R - R'_{SR})), \\ \qquad \mu = D_R \times S'_{SR}/(D_s \times (R_R - R'_{SR})). \end{cases} \tag{4.89}$$

Then we can choose the maximum base value from set A and B as the optimal base value for the send phase:

$$a = \max\{a|a \in A\ or\ a \in B\}. \tag{4.90}$$

As the base value a in send phase is fixed, then we can calculate the number of subtasks in the send phase as follow:

$$P = \lfloor \log_a((D_s/n)(a-1)/ds_{min} + 1) \rfloor. \tag{4.91}$$

The size of remaining send data is as follow:

$$d_s = D_s/n - ds_{min} \times (a^P - 1)/(a-1). \tag{4.92}$$

The size of receive data in the step of the middle phase is:

$$d_r = (ds_{min} \times (D_r/n) \times a^{P-2})/(D_s/n). \tag{4.93}$$

d_s: the data size of send in the step of the middle phase;
d_r: the data size of receive in the step of the middle phase.
Then we can calculate the total time cost of the bi-directional data transfer in the middle phase as follow:

$$tcom_{p2} = \begin{cases} d_r + (d_s - d_r \times S'_{SR})/S_S \\ \qquad\qquad if\ d_s/S'_{SR} > d_r/R'_{SR}, \\ \\ d_s + (d_r - d_s \times R'_{SR})/R_R \\ \qquad\qquad if\ d_s/S'_{SR} \leq d_r/R'_{SR}. \end{cases} \tag{4.94}$$

$tcom_{p2}$: the total time cost of the bi-directional data transfer in the middle phase. To ensure the kernel part is the main part in the middle phase, we set the subtask size to $tom_{p2} \times D_s/T_k$.

In the receive phase, the time cost of the remaining kernel execution is not shorter than the time cost of receiving the remaining receive data as kernel bound applications. Therefore, we begin with a large subtask size and then exponentially decrease the subtask size as shown in Figure 4.14. Still suppose that the minimum receive data in each GPU is dr_{min}, the base value is b and the number of subtasks in the receive phase is m, then we have the time cost of data receive and kernel execution in step i of the receive phase as follow:

$$\begin{aligned} tr_i &= dr_{min} \times b^{m+1-i}/R_R, \\ tk_i &= dr_{min} \times (T_k/n) \times b^{m-i}/(D_r/n). \end{aligned} \tag{4.95}$$

To make the total time cost minimum, we should ensure $tr_i \leq tk_i$. Then we can get the base value as follow:

$$b \leq T_k \times R_R/D_r. \tag{4.96}$$

Then $b = T_k \times R_R/D_r$ is the optimal base value.

Proof of Asymptotic Optimality

As the application is kernel bound and general heavy, then we assume that $D_s = p \times T_k$ and $D_r = q \times T_k$ where p and q are positive constant. Then the scheduling length with TPSM is:

$$\begin{aligned} T_{total} \leq ds_{min}/S_S + T_k/n + dr_{min}/R_R + \\ t_{str}(log_a(D_s(a-1)/(nds_{min})+1) + log_b(D_r(b-1)/(ndr_{min})+1)). \end{aligned} \tag{4.97}$$

As ds_{min}, dr_{min}, a and b are constant, then the scheduling length with our work can be:

$$T_{total} \leq T_k/n + O(log(pT_k)) + O(log(qT_k)). \tag{4.98}$$

When the input data size is infinite, the time cost of kernel is also infinite. As the optimal scheduling length is $T_{opt} = T_k/n$, then we have:

$$\lim_{T_k \to \infty} T_{total}/T_{opt} \leq 1 + \lim_{T_k \to \infty} O(log(pT_k))/(T_k/n) + \lim_{T_k \to \infty} O(log(qT_k))/(T_k/n) = 1. \tag{4.99}$$

Therefore, the sub-method of TPSM for kernel bound and general heavy is asymptotic optimal.

4.3.2.4 TPSM for Data Transfer Bound and Send Heavy

For data transfer bound and send heavy application, the receive part is much smaller than the send part. To achieve higher overlap and make the last receive as small as possible, we begin with a large subtask size. Then we exponentially decrease the

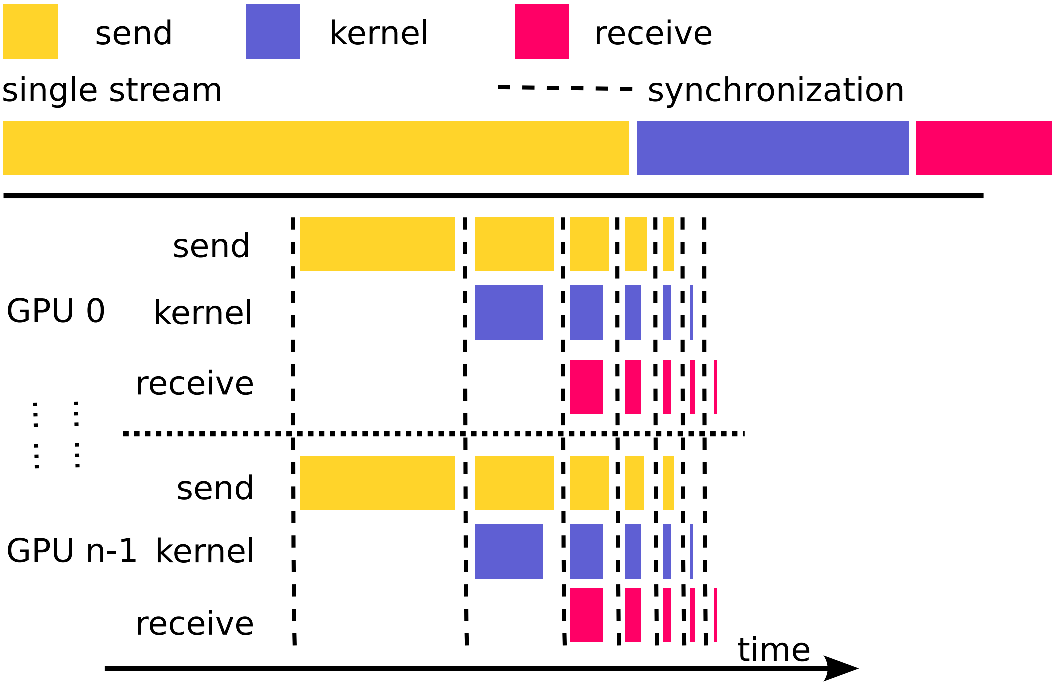


Figure 4.15: TPSM for Data Transfer Bound and Send Heavy in Symmetric Multiple GPU Architecture

subtask size as shown in Figure 4.15. With the base value a, the number of subtasks is:

$$m = \lceil \log_a((D_s/n) \times (a-1)/ds_{min} + 1) \rceil. \tag{4.100}$$

In step 1 and 2, we have:

$$\begin{aligned} x_{s1} &= ds_{min} \times a^{m-1}, \\ x_{s2} &= ds_{min} \times a^{m-2}. \\ x_{r1} &= x_{r2} = 0. \end{aligned} \tag{4.101}$$

x_{s1}: the data size to send in step 1 in each GPU;
x_{s2}: the data size to send in step 2 in each GPU;
x_{r1}: the data size to receive in step 1 in each GPU;
x_{r2}: the data size to receive in step 2 in each GPU.
In steps $m+1$ and $m+2$ in each GPU,we have:

$$\begin{aligned} x_{s(m+1)} &= x_{s(m+2)} = 0, \\ x_{r(m+1)} &= a \times dr_{min}, \\ x_{r(m+2)} &= dr_{min} = ds_{min} \times D_r/D_s. \end{aligned} \tag{4.102}$$

$x_{s(m+1)}$: the data size to send in step $m+1$ in each GPU;
$x_{s(m+2)}$: the data size to send in step $m+2$ in each GPU;

$x_{r(m+1)}$: the data size to receive in step $m+1$ in each GPU;
$x_{r(m+2)}$: the data size to receive in step $m+2$ in each GPU.
For step i ($3 \leq i \leq m$), we have the data size to send and receive for one GPU as follow:

$$\begin{aligned} x_{si} &= ds_{min} \times a^{m-i}, \\ x_{ri} &= dr_{min} \times a^{m+2-i}. \end{aligned} \tag{4.103}$$

With the data size of send and receive in each step, we can use the time optimal data transfer model to calculate the minimum time cost of data transfer in step $1, 2, m+1, m+2$ and $i(3 \leq i \leq m)$ in each GPU as follow:

$$\begin{aligned} t_1 &= G(nx_{s1}, 0), \\ t_2 &= G(nx_{s2}, 0), \\ t_{m+1} &= G(0, nx_{r(m+1)}), \\ t_{m+2} &= G(0, nx_{r(m+2)}), \\ t_i = G(nx_{si}, nx_{ri}) &= ds_{min} \times a^{m-i} \times n \times G(1, a^2 \times D_r/D_s). \end{aligned} \tag{4.104}$$

t_i: the time cost of data transfer for step i in each GPU;
n: the number of GPU devices in the system.
Then we can use the following equation to calculate the total time cost for all GPU as the kernel execution of each GPU is parallelized:

$$\begin{aligned} T_{total} = (m+2) \times t_{str} + t_1 + t_2 + t_{m+1} + t_{m+2} \\ + ds_{min} \times n \times G(1, a^2 \times D_r/D_s) \times \sum_{i=3}^{m} a^{m-i} \\ = W(a). \end{aligned} \tag{4.105}$$

t_{str}: the time cost of one synchronization between GPUs;
t_i: the time cost of data transfer in step i ($3 \leq i \leq m$).
Since $W(a)$ is not differentiable on many points, we approximately minimize it by the same method for TPSM of data transfer bound and send heavy (Algorithm 1).

Proof of Asymptotic Optimality

As it is data transfer bound and send heavy, we assume that $D_s = p \times D_r$ where $p >> 1$. The scheduling length with TPSM for data transfer bound and send heavy is:

$$T_{total} = (1+a)dr_{min}/R_R + G(D_s, D_r - ndr_{min}(a+1)) + t_{str} log_a(D_r(a-1)/(ndr_{min})+1). \tag{4.106}$$

From equation (4.68) and (4.69) we know that $G(x, y) = q \times x + k \times y$ where q and k are non-negative constant. D_s is much larger than D_r so that point (D_s, D_r) is very close to point $(D_s, D_r - ndr_{min}(a+1))$ in 2-D coordinate system. The system states (discussed in section 4.3.1) used for both cases are the same. Therefore, function $F(x, y)$ in equation (4.69) for (D_s, D_r) and $(D_s, D_r - ndr_{min}(a+1))$ is the same. Then the scheduling length can be as follow:

$$T_{total} \leq G(D_s, D_r) + O(log_a D_r) = (pq+k)D_r + O(log_a D_r). \tag{4.107}$$

When the input data size is infinite, the output data size is also infinite. As the optimal scheduling length is $T_{opt} = G(D_s, D_r) = (pq+k)D_r$ and a is constant, then we have:

$$\lim_{D_r \to \infty} T_{total}/T_{opt} \leq 1 + \lim_{D_r \to \infty} O(log_a D_r)/((pq+k)D_r) = 1. \tag{4.108}$$

Therefore, the sub-method of TPSM for data transfer bound and send heavy is asymptotic optimal.

4.3.2.5 TPSM for Data Transfer Bound and Receive Heavy

For data transfer bound and receive heavy applications, the the send part is much smaller than the receive part. We divide the scheduling into three phases:(1)*send phase*, (2)*middle phase* and (3)*receive phase* as shown in Figure 4.16. To minimize the total time cost, we should ensure the time cost of data transfer is not shorter than the time cost of kernel execution in each step of each phase while we also should overlap the send part and receive part as much as possible. Therefore, we begin with a small subtask size and exponentially increase the subtask size in the send phase. As a transitional phase, there is only one step in the middle phase. In this step, we have to finish all the remaining data send and receive all the output data generated by the kernel execution in the last step of the send phase. Then we launch a sub-kernel with the time cost of kernel execution equal to the time cost of the data transfer in this step. As it is data transfer bound and receive heavy application, the time cost of the remaining data receive is not shorter than the time cost of the remaining kernel execution in the receive phase. Therefore, we begin with a small subtask size to make the first kernel small and then exponentially increase the subtask size.

In the send phase, assume the base value of the increase is a, then the data size of send and receive for each GPU is:

$$\begin{aligned} x_{si} &= ds_{min} \times a^{i-1}, \\ x_{ri} &= ds_{min} \times a^{i-3} \times D_r/D_s. \end{aligned} \tag{4.109}$$

With the time optimal data transfer model, we can calculate the time cost of data transfer in step i as follow:

$$t_i = G(nx_{si}, nx_{ri}) = n \times ds_{min} \times a^{i-3} G(a^2, D_r/D_s). \tag{4.110}$$

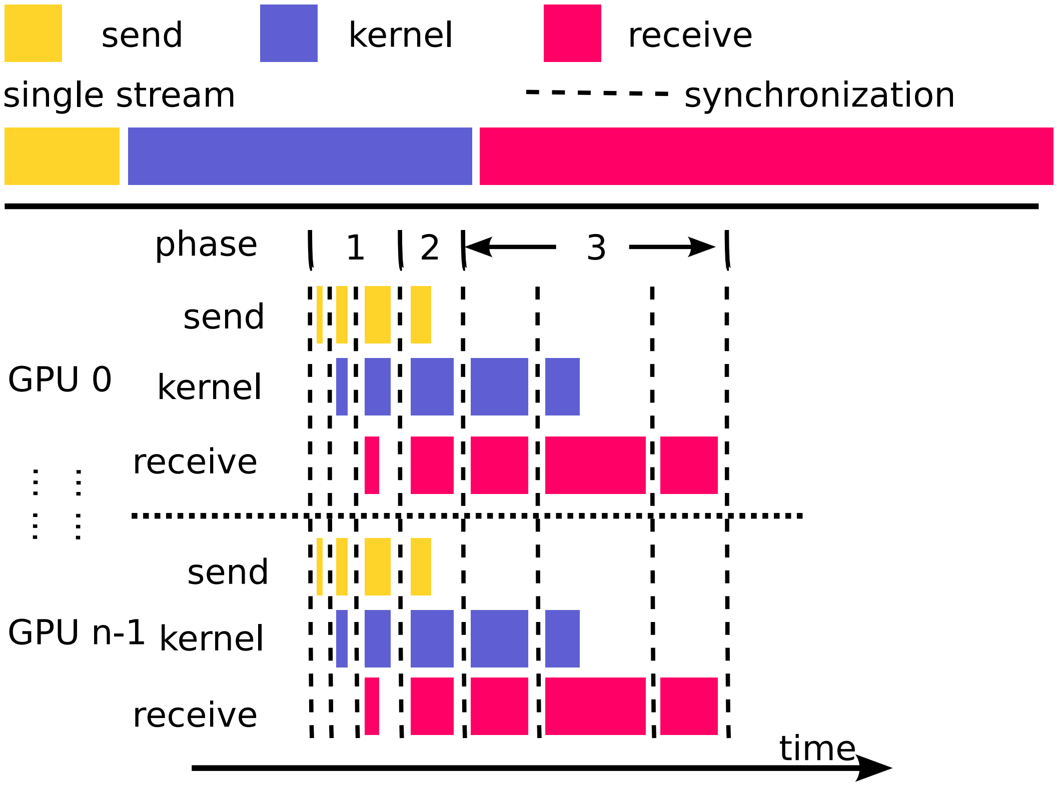


Figure 4.16: TPSM for Data Transfer Bound and Receive Heavy in Symmetric Multiple GPU Architecture

As the time cost of kernel execution in step i is:

$$tk_i = x_{s(i-1)}/(D_s/n) \times (T_k/n) = ds_{min} \times a^{i-2} \times T_k/D_s. \tag{4.111}$$

To minimize the total time cost, we should ensure $tk_i \leq t_i$ in each step of the send phase. Then we can have the following set of base a that matches the condition:

$$S = \{a | a \times T_k \leq n \times G(D_s \times a^2, D_r)\}. \tag{4.112}$$

Then we choose $a = \max\{a | a \in S\}$ as the optimal base value for the scheduling in the send phase. As it is data transfer bound and receive bound heavy applications, there exits a solution in the above equation.

In the middle phase, we will send all the remaining send data and receive all the output data generated by the kernel execution of the last step in the send phase. The data size of sending and receiving in the middle phase is as follow:

$$\begin{aligned} d_s &= D_s/n - ds_{min} \times (a^P - 1)/(a - 1), \\ d_r &= (ds_{min} \times (D_r/n) \times a^{P-2})/(D_s/n). \end{aligned} \tag{4.113}$$

P: the number of subtasks in the send phase which can be calculated by equation (4.91).

The optimal time cost of the bi-directional data transfer is:

$$t_{mid} = G(nd_s, nd_r) \quad (G() \; is \; defined \; in \; equation (4.69).) \tag{4.114}$$

To ensure the data transfer part is the main part in the middle phase, we set the subtask size to $t_{mid} \times D_s/T_k$.

As the remaining receive part is not shorter than the remaining kernel execution part, we receive all the data generated from the middle phase in the first step of the receive phase as shown in Figure 4.16. As the kernel time in the middle phase is t_{mid}, the data size of data receiving in the first step of the receive phase in each GPU is:

$$x_{r1} = t_{mid} \times (D_r/n)/(T_k/n) = t_{mid} \times D_r/T_k. \tag{4.115}$$

x_{r1}: the data size of data receiving in the first step of the receive phase in each GPU.

Then the time cost of data receiving in the first step of the receive phase is:

$$t_{r1} = G(0, nx_{r1}) = nt_{mid} \times D_r \times G(0,1)/T_k. \tag{4.116}$$

t_{r1}: the time cost of kernel execution in the first step of the receive phase.

In the first step of the receive phase, we set the kernel execution equal to the data receiving. Then the subtask size in the first step of the receive phase is as follow:

$$x_1 = t_{r1} \times (D_s/n)/(T_k/n) = t_{r1} \times D_s/T_k. \tag{4.117}$$

In step i of the receive phase ($i \geq 2$), we receive all the output data generated from the kernel execution in step $i-1$. The data size of data receiving in step i of the receive phase in each GPU is:

$$x_{ri} = t_{r(i-1)} \times (D_r/n)/(T_k/n) = t_{r(i-1)} \times D_r/T_k. \tag{4.118}$$

The time cost of data receiving in step i of the receive phase is:

$$t_{ri} = G(0, nx_{ri}) = nt_{r(i-1)} \times D_r \times G(0,1)/T_k. \tag{4.119}$$

We also set the time cost of kernel execution equal to the data receiving. Therefore, the subtask size in step i of data receiving is:

$$x_i = t_{ri} \times (D_s/n)/(T_k/n) = t_{ri} \times D_s/T_k. \tag{4.120}$$

Finally, we can finish all the kernel execution in the penultimate step and finish all the data receiving in the last step. By this way, we can hide the kernel execution and use as fewer synchronization operation as possible to minimize the total time cost.

Proof of Asymptotic Optimality

As it is data transfer bound and receive heavy, we assume that $D_r = p \times D_s$ where $p >> 1$. As the data receiving is longer than the kernel execution in receive phase and we set the time cost of kernel execution equal to the time cost of data receiving in most steps of receive phase, the number of synchronization operations can be considered as constant e. The scheduling length with TPSM for data transfer bound and receive heavy is:

$$\begin{aligned} T_{total} = (1+a)ds_{min}/S_S + G(D_s - n(1+a)ds_{min}, D_r) \\ +t_{str}(log_a(D_s(a-1)/(nds_{min})+1)+1+e). \end{aligned} \tag{4.121}$$

Similar to the proof asymptotic optimality for data transfer bound and send heavy , we have $G(x, y) = q \times x + k \times y$ where q and k are non-negative constant. Then the scheduling length can be as follow:

$$T_{total} \leq G(D_s, D_r) + O(log_a D_s) = (q+pk)D_s + O(log_a D_s). \tag{4.122}$$

As the optimal scheduling length is $T_{opt} = G(D_s, D_r) = (q+pk)D_s$ and a is constant, then we have:

$$\lim_{D_s \to \infty} T_{total}/T_{opt} = 1 + \lim_{D_s \to \infty} O(log_a D_s)/((q+pk)D_s) = 1. \tag{4.123}$$

Therefore, the sub-method of TPSM for data transfer bound and receive heavy is asymptotic optimal.

4.3.2.6 TPSM for Data Transfer Bound and General Heavy

For data transfer bound and general heavy applications, both the data send and receive part are not short. Therefore, the key is to overlap the data send and receive part as much as possible. We set the subtasks to equal size for easy scheduling and use the time optimal data transfer algorithm to schedule the data transfer as shown in Figure 4.17.

Suppose that the number of subtasks is m, then we have the data size of send and receive for one GPU in step i as follow:

$$\begin{aligned} x_{si} = D_s/(n \times m), \\ x_{ri} = D_r/(n \times m). \end{aligned} \tag{4.124}$$

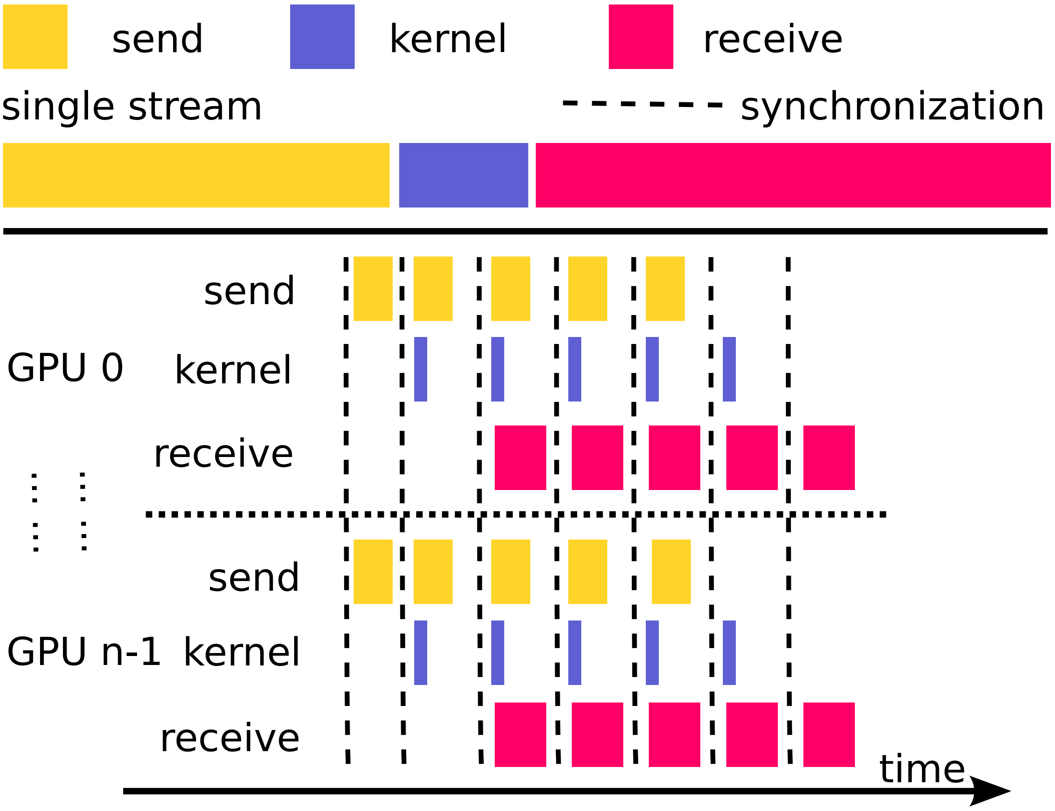


Figure 4.17: TPSM for Data Transfer Bound and General Heavy in Symmetric Multiple GPU Architecture

By using time optimal data transfer model, we can calculate the optimal time cost of data transfer for all GPUs in each step as follow($3 \leq i \leq m$):

$$\begin{aligned} t_1 = t_2 &= G(n \times D_s/(n \times m), 0), \\ t_{m+1} = t_{m+2} &= G(0, n \times D_r/(n \times m)), \\ t_i &= G(n \times D_s/(n \times m), n \times D_r/(n \times m)). \end{aligned} \tag{4.125}$$

With the time cost of each step, we can calculate the total time cost as follow:

$$\begin{aligned} T_{total} &= (m+2)t_{str} + t_1 + t_2 + \sum_{i=3}^{m} t_i + t_{m+1} + t_{m+2} \\ &= (m+2)t_{str} + 2G(D_s/m, 0) + 2G(0, D_r/m) + (m-2)G(D_s/m, D_r/m) \\ &= (m+2)t_{str} + 2(G(D_s, 0) + G(0, D_r))/m + G(D_s, D_r)(m-2)/m = W(m). \end{aligned} \tag{4.126}$$

The total time cost of the application is a function of the number of subtask. Then we calculate the derivative of the function $W(m)$ to get the m for the scheduling which makes the total time cost minimum as follow:

$$m = \sqrt{2(G(D_s, 0) + G(0, D_r) - G(D_s, D_r))/t_{str}} \tag{4.127}$$

Proof of Asymptotic Optimality

The scheduling length with TPSM for data transfer bound and general heavy is:

$$T_{total} = (m + 2)t_{str} + 2D_s/(nmS_S) + 2D_r/(nmR_R) + (m - 2)G(D_s/m, D_r/m). \tag{4.128}$$

Assume $D_r = pD_s$ and $c = \sqrt{2(G(1, 0) + G(0, p) - G(1, p))/t_{str}}$, so $G(1, p)$ and c are constant. Take $m = c\sqrt{D_s}$ into the equation (4.128), then the scheduling length is:

$$\begin{aligned} T_{total} &= (c\sqrt{D_s} + 2)t_{str} + 2\sqrt{D_s}/(cS_S) + 2p\sqrt{D_s}/(cR_R) + (D_s - 2\sqrt{D_s}/c)G(1, p) \\ &\leq D_sG(1, p) + O(\sqrt{D_s}). \end{aligned} \tag{4.129}$$

As the optimal scheduling length is $T_{opt} = G(D_s, D_r) = D_sG(1, p)$, then we have:

$$\lim_{D_s \to \infty} T_{total}/T_{opt} \leq 1 + \lim_{D_s \to \infty} O(\sqrt{D_s}/(D_sG(1, p))) = 1. \tag{4.130}$$

Therefore, the sub-method of TPSM for data transfer bound and general heavy is asymptotic optimal.

4.4 TPSM for Non-symmetric Multiple GPUs Architecture

In non-symmetric multiple GPUs architecture, there are mainly two differences comparing to symmetric multiple GPUs architecture. First of all, the bandwidth is not equal for each GPU in non-symmetric case. As shown in Table 4.3, the bandwidth table of non-symmetric architecture is not symmetric as in symmetric architecture. We can not directly use the time optimal data transfer algorithm for symmetric architecture in the non-symmetric architecture. Therefore, we propose a time optimal data transfer for non-symmetric GPUs architecture based on the algorithm for symmetric GPUs architecture. Secondly, the computing capacity of each GPU in non-symmetric architecture is different while it is equal in symmetric architecture. There is a load balance problem in non-symmetric architecture. To solve this problem, we propose a TPSM for non-symmetric architecture based on TPSM for symmetric architecture which can well handle the load balance between GPUs.

4.4.1 Time Optimal Data Transfer Algorithm for Non-symmetric GPUs Architecture

As in 2-D coordinate system for symmetric architecture, the X-axis presents data size of sending and the Y-axis presents data size of receiving. Because the data size of each GPU are the same, it is not necessary to differentiate the data size of sending or receiving from each GPU. However, the data size of sending and receiving of each GPU device can be different for non-symmetric architecture. Therefore, one separate axis is required to present the data sending or receiving of each GPU. So we need a 2n-D coordinate system for a non-symmetric n-GPU architecture. For a non-symmetric multiple GPUs system with n GPUs, there are 4^n valid system states and no duplicated system states. For each system state, it can be described by a 2n-D coordinate like $(s_1, r_1, ..., s_n, r_n)$ where s_i and r_i are the bandwidth of sending and receiving of GPU_i. So we have a point set $U = \{P_1, P_2, P_3, ..., P_{2^n-1}\}$ in a 2n-D coordinate system.

Comparing to symmetric architecture, the time optimal data transfer algorithm for non-symmetric architecture is much more complex but the basic idea is the same. First of all, we have to find out the upper right convex hull in the 2N-D coordinate system. In 2-D coordinate system, the upper right convex hull is composed with segments that each segment is decided by two points. In 3-D coordinate system, the upper right convex hull is composed with planes that each plane is surrounded by three points. Therefore, it is easy to find out that the upper right convex hull in N-D

Table 4.3: Bandwidth of Dual Non-symmetric GPUs under Different States(G0:Tesla K20c, G1:Tesla C2075)

	G0-N	G0-S	G0-R	G0-S&R
G1-N	-	$S0 = 6.36$	$R0 = 6.71$	$S0 = 4.08$ $R0 = 4.05$
G1-S	$S1 = 5.18$	$S0 = 5.3$ $S1 = 4.94$	$R0 = 3.61$ $S1 = 4.41$	$S0 = 2.26$ $R0 = 2.25$ $S1 = 4.43$
G1-R	$R1 = 4.51$	$S0 = 3.38$ $R1 = 3.05$	$R0 = 3.18$ $R1 = 2.4$	$S0 = 1.97$ $R0 = 1.9$ $R1 = 2.43$
G1-S&R	$S1 = 3.03$ $R1 = 3.16$	$S0 = 2.58$ $S1 = 2.31$ $R1 = 2.34$	$R0 = 2.53$ $S1 = 1.91$ $R1 = 1.93$	$S0 = 1.65$ $R0 = 1.65$ $S1 = 1.84$ $R1 = 1.85$

coordinate system is composed with geometric shapes that each shape is decided by n points. We can use a set of n points to present the geometric shapes that belong to the upper right convex hull. We define a set of n points as *n-point subset*. If the geometric shapes composed by a set of n points belongs to the upper right convex hull, then we call it as a *hull subset*. Given with m points in N-D coordinate system, there are C_m^n n-point subset. There are many works on finding out the convex hull in high-dimension coordinate system such as [100, 26]. With the approaches, we can get the hull subset union within polynomial time. Notice that we only need to get the hull subset union once.

Algorithm 2: Algorithm for finding out optimal hull subset

```
Input: $S$: the union of hull subsets,
        $E = (x_1, x_2, .., x_n)$: an input point
Output: $A$: the optimal hull subset,
        $a$: time proportion for $n$ system states(N-D vector, $a_i$ is the time
proportion for system states $x_i$)
calculate the function of the ray $OE$:
        $y_1 = at, y_2 = bt, ..., y_n = wt$;
for each $A \in S$ do
    calculate the shape function of $A$:
        $F(x_1, x_2, ..., x_n) = ax_1 + bx_2 + ... + zx_n - 1 = 0$;
    calculate the intersection $F$ between the shape $A$ and the ray $OE$;
    if exits one non-negative solution $a$ for $F = a \times A$ then
        return $A$ and $a$
```

With the hull subset union, we need to find out the optimal hull subset for a given data transfer size. The data size optimal scheduling for n GPU non-symmetric architecture is a combination of n system states which is a n point subset belongs to the hull subset union. By changing the scheduling time, the upper right convex hull linearly scales about the origin. Therefore, with an input data transfer size which can be described as an input point in the 2N-D coordinate system, we can draw a line that passes the origin and the input point named *input line*. This line will intersect with the upper right convex hull. Then the hull subset intersected by the line is the data size optimal scheduling.

We use Algorithm 2 to find out the data size optimal scheduling. For each hull subset in the union, we calculate if the input line cross the space enclosed by the hull subset. If the result is yes, then this is the data size optimal scheduling and we stop the process. Otherwise, we go on for the searching.

4.4.2 TPSM

For non-symmetric multiple GPUs architecture, the total time cost of application is the maximum execution time cost among all the execution time in each GPU. Therefore, it is very important to keep load balance for better performance. With this target, we propose a two layer partitioning based on the TPSM for symmetric multiple GPUs architecture. With the new partitioning method, we can dynamically adjust the load allocation between GPUs and finally to have a load balance allocation.

First of all, we partition the application into equal size blocks as shown in Figure 4.18. Then for each block, we will partition it into n parts with different size in n GPUs system. Each part in the block will be allocated to the related GPU and we will use the TPSM for single GPU to partition and execute. In this section, we only discuss the partitioning above subtask level. The partitioning of each part in single GPU has been discussed in Section 4.2.

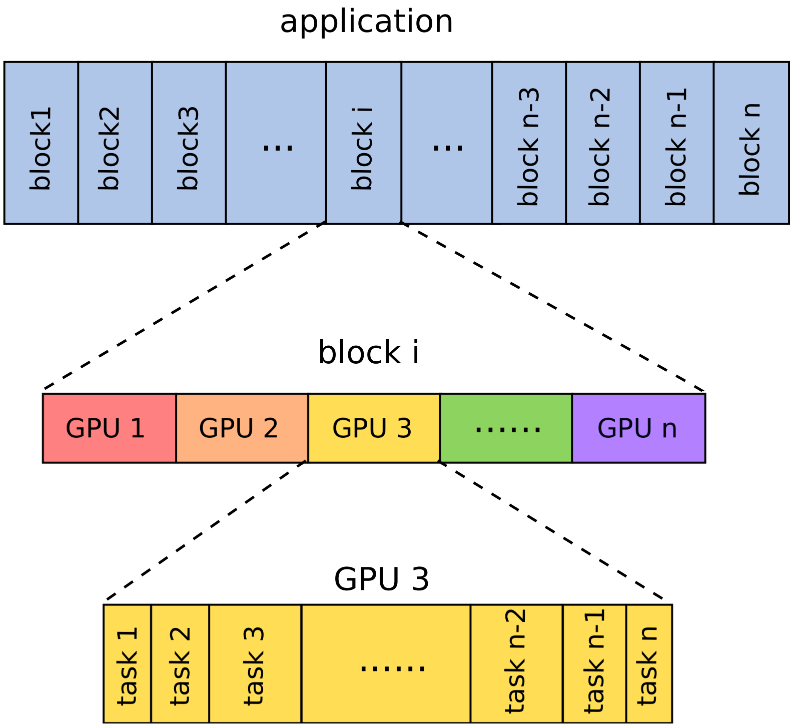


Figure 4.18: Two Layer Partitioning

We can find that the partitioning within one block decides the load allocation between GPUs. The partitioning of previous blocks may be not suitable to keep well balance. Therefore, we partition the application into many blocks as we can adjust the partitioning of latter blocks based on the feedback information from the execution of previous blocks.

The scheduling process is shown in Figure 4.19. For the partitioning of the first block, there is no feedback for reference. We have to utilize existing information to make the partitioning. One method is to use the hardware information such as peak performance which somehow reflect the computing capacity. However, using hardware information may lead to a partitioning which is far from perfect load balance. For example, the single precision peak performance of Tesla K20c is 3.52 Tflops while the single precision peak performance of Tesla C2075 is 1.03 Tflops. Tesla K20c is 3.5 time faster than Tesla C2075. However, we found that Tesla K20c can only achieve around 1.4 speedup with linear filter benchmark which is far from the peak performance. This is because the actual execution of application is often difficult to achieve GPU's peak performance. There are many factors that can affect the performance of application in one GPU and the features of the application is an important aspect. Therefore, we give up the first method and adopt another method.

The second method is using the performance model discussed in Section 3 to pre-

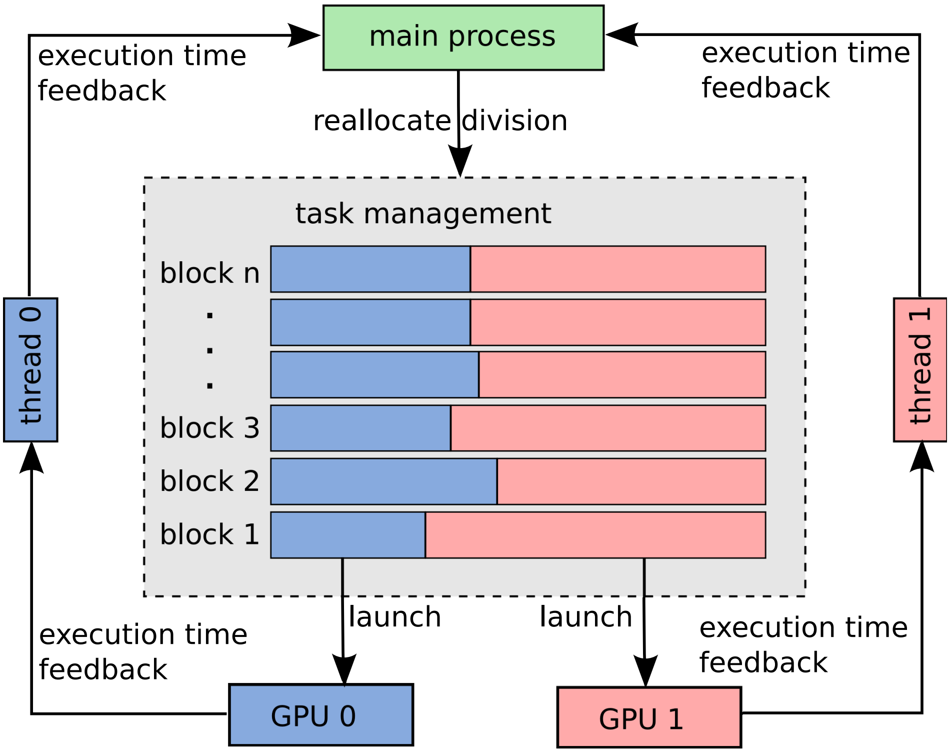


Figure 4.19: Scheduling Process for Non-symmetric Dual GPUs

dict the performance of the application in each GPU and then make the partitioning based on the prediction results. First of all, we use the performance model to predict the execution time of a small sample kernel from the application in each GPU. Then we will partition the application into different parts based on the prediction results.

Taking dual GPUs as an example as shown in Figure 4.19, suppose the prediction results of the execution time for GPU 0 is a and the prediction results of the execution time for GPU 1 is b. Then we will partition the first block into two parts. Suppose the application load is 1. Then the first part size is $b/(a+b)$ and will be allocated to GPU 0 while the second part size is $a/(a+b)$ and will be allocated to GPU 1. Due to the existence of deviation from the prediction comparing to the actual results, the partitioning in the first block may be not well load balance. Therefore, we collect the execution information of the previous block such as kernel time cost and send these feedback information to the threads which is responsible for the controlling of the GPUs as shown in Figure 4.19. Each thread collects the feedback information and sends it to the main process. The main process gathers all feedback information from all GPUs and then reallocate the load in each GPU to make the load between GPUs more balanced. It repeats the above step, and then will soon come to a load balance status between GPUs. Notice that the feedback information collection mechanism will cause some overhead as there are some synchronization operations in CPU during which the GPU is idle. Therefore, we suggest the block size should not be small. Otherwise, the benefit from load balance may be less than the overhead from the synchronization operations.

4.5 Experimental Evaluation

We use four type GPUs to test our work as shown in Table 4.5. All these GPUs have two copy engines. We have two machines named *AT38* and *AT50* as shown in Table 4.4. For both machines, we use CUDA 4.2 and Ubuntu 10.04.3-64-bit edition OS. To test the performance of our work, we use four benchmarks that some are used in Linderman's work[78] and we port them from multi-core platform to multiple GPU platform as shown in Table 4.6.

Table 4.4: Specification of The Host Machines

machine	Device	Cores	Clock speed	Cache	Main Memory
AT38	4 x Intel Xeon X5650	4 x 6	2.67GHz	12MB	6 x 2GB
AT50	2 x Intel Xeon E5-2680	2 x 8	2.7GHz	20MB	8 x 8GB

4.5.1 Symmetric Architecture

4.5.1.1 Configuration

For symmetric architecture, we install two Tesla C2070 and two Tesla C2075 in AT38 and install four Tesla M2090 in AT50. Then we can have three symmetric multiple GPU platform: dual Tesla C2070, dual Tesla C2075 and four Tesla M2090.

As the number of SM of each GPU is different, we use 56 blocks with one warp in each block for Tesla C2070 and Tesla C2075 and use 64 blocks with one warp in each block for Tesla M2090. We compare the time cost of TPSM with good manual code which only use stream in each GPU. The abbreviations used in the Figures are as follow:
$NON-1GPU$: the results of single GPU without TPSM;
$BND-1GPU$: the maximum part among data sending, kernel execution and data receiving of results without TPSM in single GPU;
$TPSM-1GPU$: the results of single GPU with TPSM;
$NON-2GPU$: the results of dual GPUs without TPSM;
$BND-2GPU$: the results of lower bound of dual GPUs;
$TPSM-2GPU$: the results of dual GPUs with TPSM;
$NON-4GPU$: the results of four GPUs without TPSM;
$BND-4GPU$: the results of lower bound of four GPUs;
$TPSM-4GPU$: the results of four GPUs with TPSM.

Table 4.5: Specification of GPUs

Devices(Tesla Series)	C2070(G1)	C2075(G2)	M2090(G3)	K20c
Processor cores	14×32	14×32	16×32	13×192
GPU clock rate	1.15GHz	1.15GHz	1.3GHz	0.71GHz
Memory clock rate	1.5GHz	1.56GHz	1.85GHz	2.6GHz
Memory size	5375MB	6143MB	5375MB	4800MB

Table 4.6: Benchmark Programs used for TPSM

Program	Description	Type
black-scholes[78]	European option pricing	transfer bound
linear filter[78]	image process filter	kernel bound
sepia filter[78]	image process filter	kernel bound
matrix	matrix multiplication	complex type

4.5.1.2 Results

First of all, we apply our work to linear filter benchmark in each platform. We have three input image sizes: 2000^2 pixels, 4000^2 pixels and 8000^2 pixels. The time cost results in Tesla C2070 are shown in Figure 4.20. We also compare the speedups to single GPU case without TPSM and the speedup results are shown in Figure 4.21.

The data transfer part account for around 7% in the total time cost in single GPU case of Tesla C2070. Therefore, we use the TPSM for kernel bound and general heavy. In one GPU case, TPSM can achieve 1.035 $\sim$ 1.045 speedup while the lower bound speedup is 1.069 $\sim$ 1.076. The low speedup is mainly because of the low proportion of data transfer in the total time cost.

In dual GPU case, manual codes can achieve 1.88 $\sim$ 1.94 speedup which is almost twice comparing to single GPU. TPSM can achieve 2.13 $\sim$ 2.17 speedup while lower bound case can achieve 2.14 $\sim$ 2.19. The results of TPSM is very close to the results of lower bound in Tesla C2070.

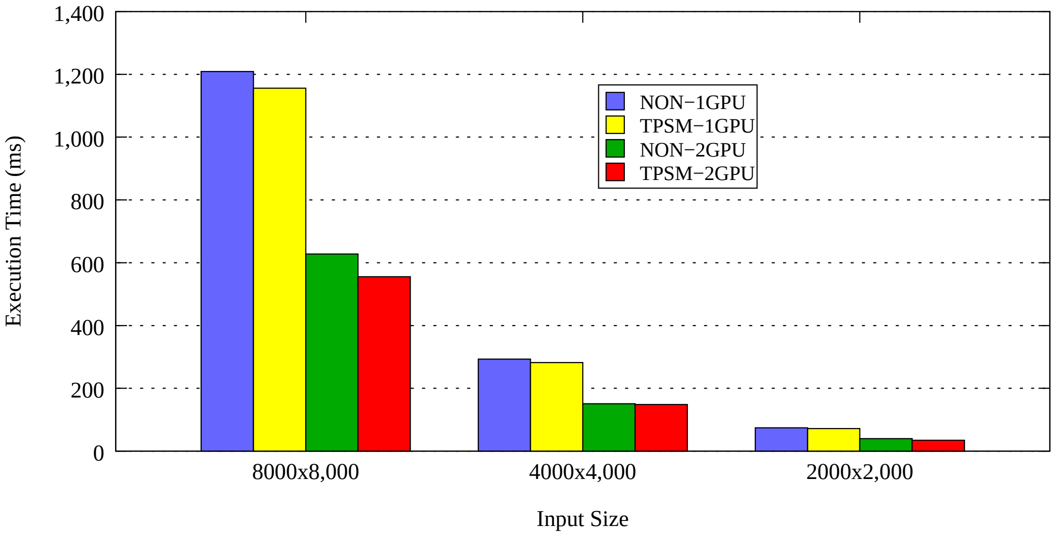


Figure 4.20: Linear Filter Results in Tesla C2070

The time cost results of linear filter benchmark in Tesla C2075 are shown in Figure 4.22, and the speedup results are shown in Figure 4.23. The data transfer part account for around 6.9% in the total time cost in single GPU case of Tesla C2075.

In one GPU case, TPSM can achieve 1.08 $\sim$ 1.16 speedup while the lower bound speedup is 1.065 $\sim$ 1.077. We find that inserting some synchronization operation during the kernel execution can sometimes reduce the time cost of kernel execution. On the other hand, synchronization operation can also increase the time cost of ker-

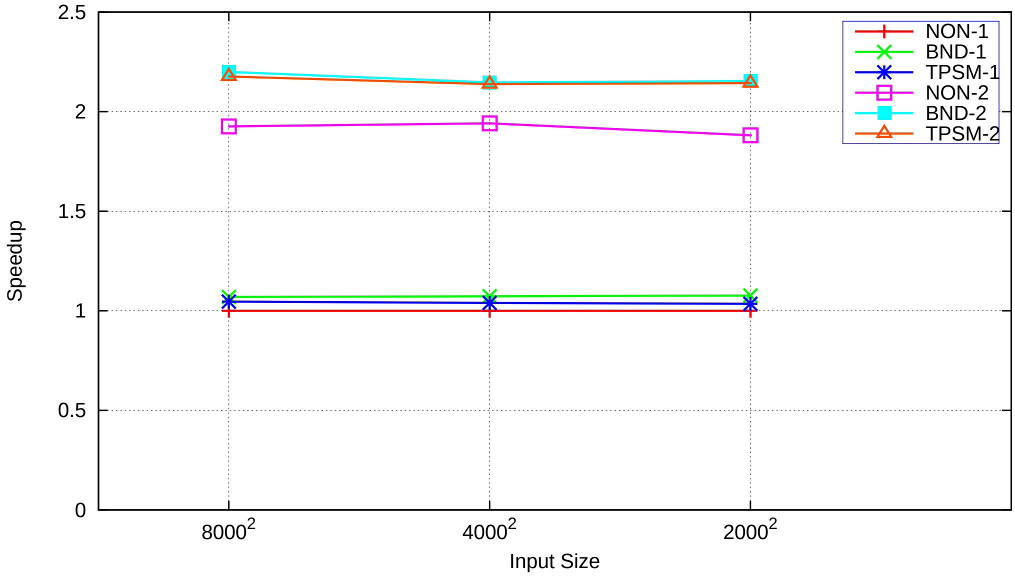


Figure 4.21: Linear Filter Speedup Results in Tesla C2070

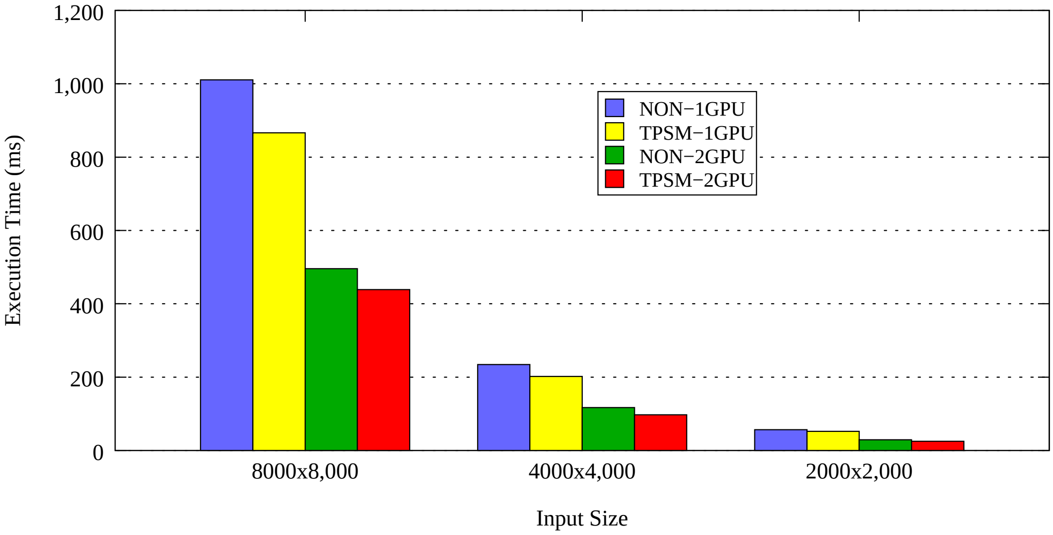


Figure 4.22: Linear Filter Results in Tesla C2075

nel execution. Therefore, it is difficult to utilize inserting synchronization operation to improve kernel performance on purpose.

In dual GPU case, manual codes can achieve 2.14 ∼ 2.23 speedup which is a little more than twice comparing to single GPU. TPSM can achieve 2.25 ∼ 2.4 speedup while lower bound case can achieve 2.14 ∼ 2.23. The results of TPSM is better than the results of lower bound in Tesla C2075. This is also because inserting synchronization operation leads to positive affection on kernel execution.

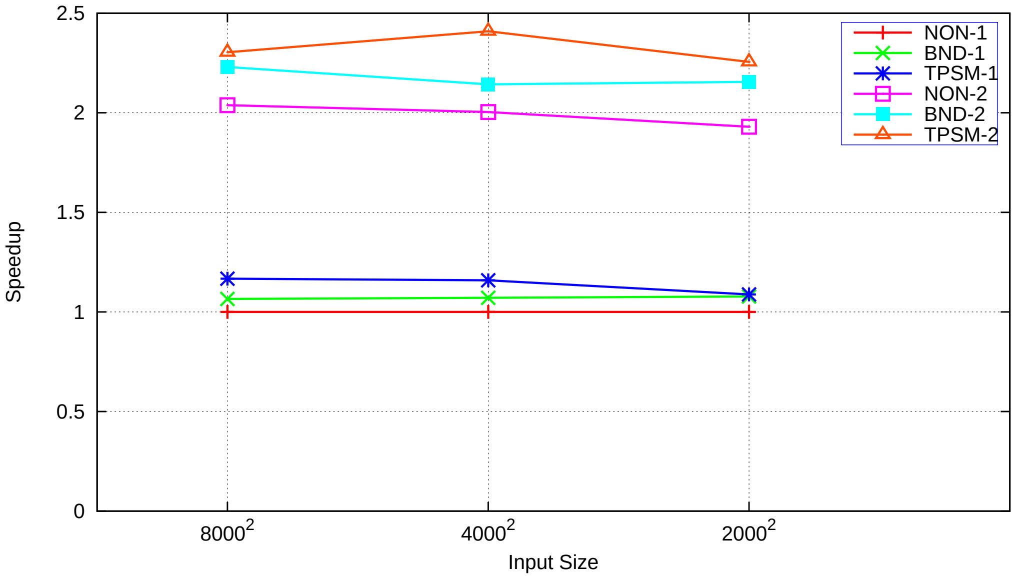


Figure 4.23: Linear Filter Speedup Results in Tesla C2075

The time cost results of linear filter benchmark in Tesla M2090 are shown in Figure 4.24, and the speedup results are shown in Figure 4.25. The data transfer part account for around 6.1% in the total time cost in single GPU case of Tesla M2090. In single GPU case, TPSM can achieve 1.06 ∼ 1.26 speedup while the lower bound speedup is 1.06 ∼ 1.07 for three different input sizes.

In dual GPUs case, manual codes achieve 1.95 ∼ 1.97 speedup. TPSM can achieve 2.13 ∼ 2.22 speedup while the lower bound speedup is 2 ∼ 2.07. In four GPUs case, manual codes achieves 4 and 4.06 speedup for input 4000^2 and 8000^2 pixel images while it only achieves 2.78 speedup for input 2000^2 pixel image. This is mainly because the time cost of kernel execution is so short for the input that the thread launch time cost can not be ignored. As the time cost of thread launch is fixed, it is difficult to achieve four times speedup when the total time cost is small. TPSM achieves 4.72 speedup for input size 8000^2, 4.2 speedup for input size 4000^2 and 3.32 speedup for input size 2000^2 while the lower bound achieves 4.13 speedup

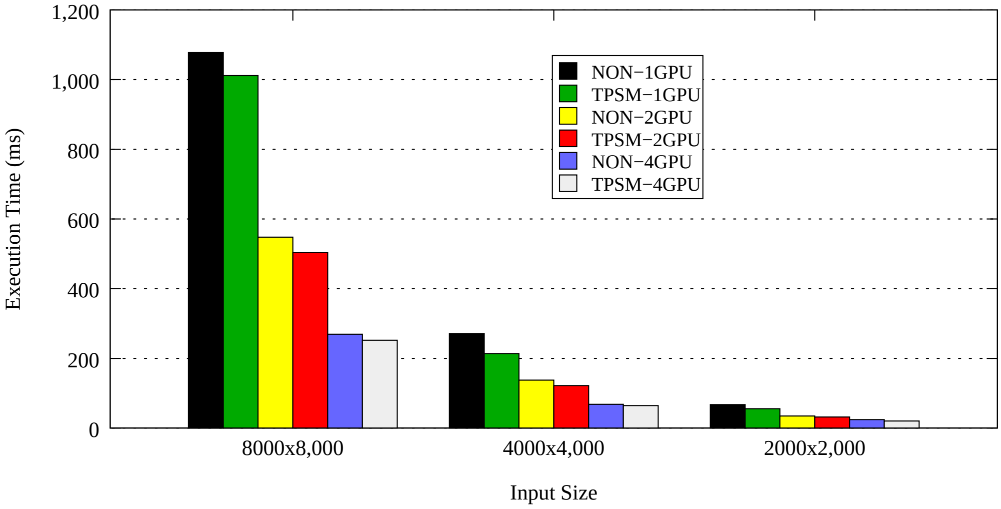


Figure 4.24: Linear Filter Results in Tesla M2090

for input size 8000^2, 4.09 speedup for input size 4000^2 and 2.78 speedup for input size 2000^2. We can see both TPSM and lower bound can not achieve four times speedup for input size 2000^2 as the thread launch overhead is heavy comparing to the short kernel time cost.

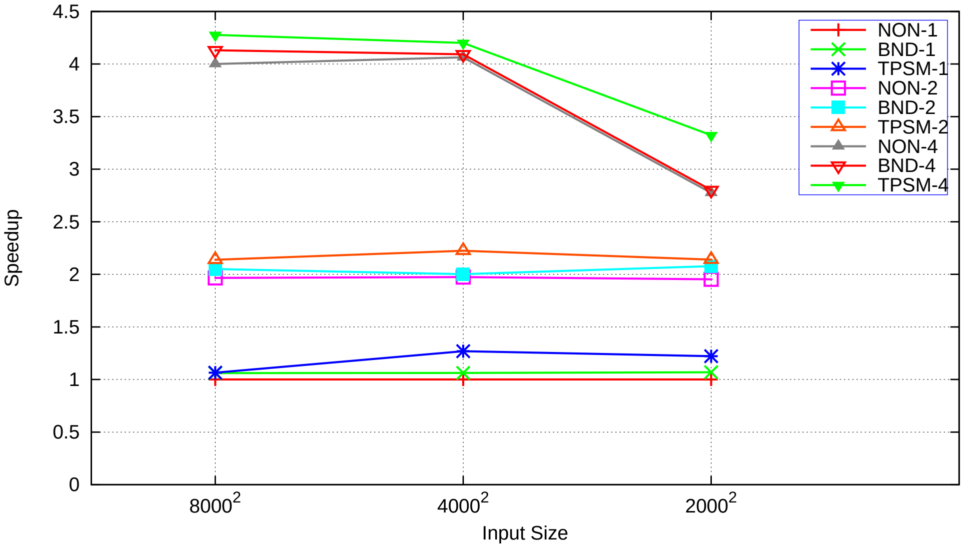


Figure 4.25: Linear Filter Speedup Results in Tesla M2090

We also applied our work to sepia filter benchmark which is similar to linear filter benchmark but has larger data transfer proportion in the total time cost.

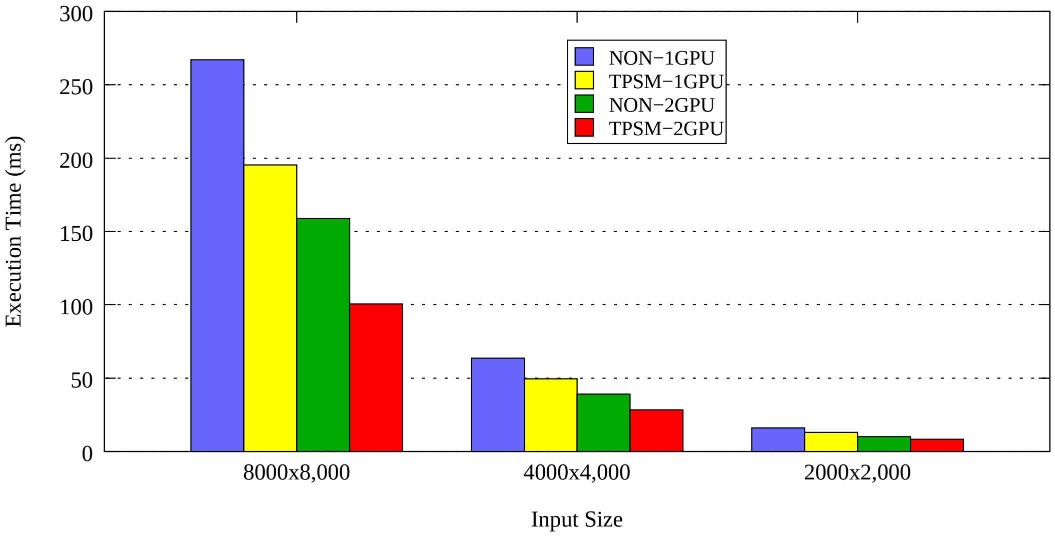


Figure 4.26: Sepia Filter Results in Tesla C2070

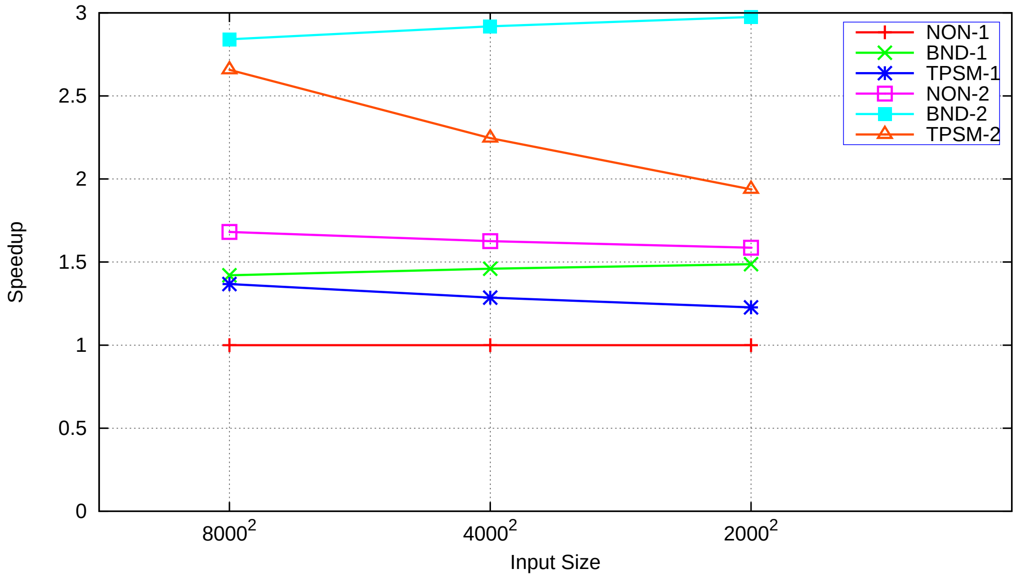


Figure 4.27: Sepia Filter Speedup Results in Tesla C2070

The time cost results of sepia filter benchmark in Tesla C2070 are shown in Figure 4.26, and the speedup results are shown in Figure 4.27. The data transfer part account for around 32.4% in the total time cost in single GPU case of Tesla

C2070. Therefore, we use the TPSM for kernel bound and general heavy. In one GPU case, TPSM can achieve 1.22 ∼ 1.36 speedup while the lower bound speedup is 1.42 ∼ 1.48.

In dual GPU case, manual codes can achieve 1.58 ∼ 1.68 speedup. This is because the bandwidth between CPU and GPU becomes performance bottleneck. The lower bound achieves 2.84 ∼ 2.97 speedup which TPSM can achieve 2.65 speedup for input size 8000^2, 2.25 speedup for input size 4000^2 and only 1.93 speedup for input size 2000^2. We find that with TPSM, the performance of applications with short kernel is not as good as the performance of applications with long kernel. The main problem is still the synchronization operations. As we have mentioned that the synchronization operations can improve the performance of kernel execution but also notice that the operation itself cost some time. Therefore, when the kernel time is long, the affection of the synchronization operation overhead on the speedup is small. When the kernel time is short, the affection of the synchronization operation overhead on the speedup can be very large.

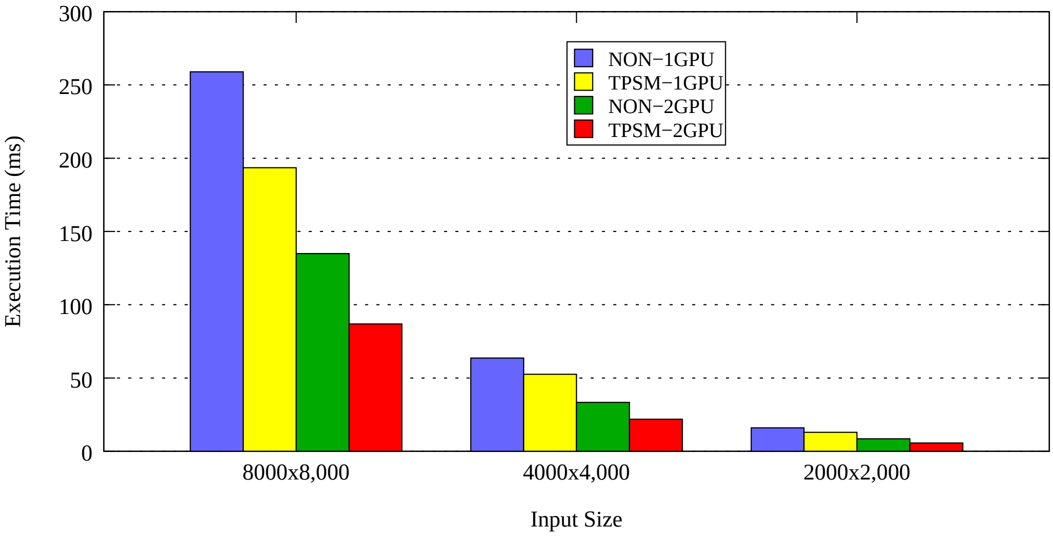


Figure 4.28: Sepia Filter Results in Tesla C2075

We applied sepia filter benchmark in Tesla C2075. The time cost results are shown in Figure 4.28 and the speedup results are shown in Figure 4.29. The data transfer part account for around 31.5% in the total time cost in single GPU case of Tesla C2075. In one GPU case, TPSM can achieve 1.21 ∼ 1.27 speedup while the lower bound speedup is 1.42 ∼ 1.48.

In dual GPU case, manual codes can achieve 1.87 ∼ 1.92 speedup which is almost twice speedup and much lager than the case in Tesla C2070. This is because the bandwidth of platform with Dual Tesla C2075 is larger than the bandwidth of platform with Dual Tesla C2070. Notice that the bandwidth is only decided by the GPU device but also by the motherboard. The lower bound achieves 2.85 ~ 2.97 speedup while TPSM can achieve 2.84 ~ 2.96 speedup. The results of TPSM are very close to the lower bound.

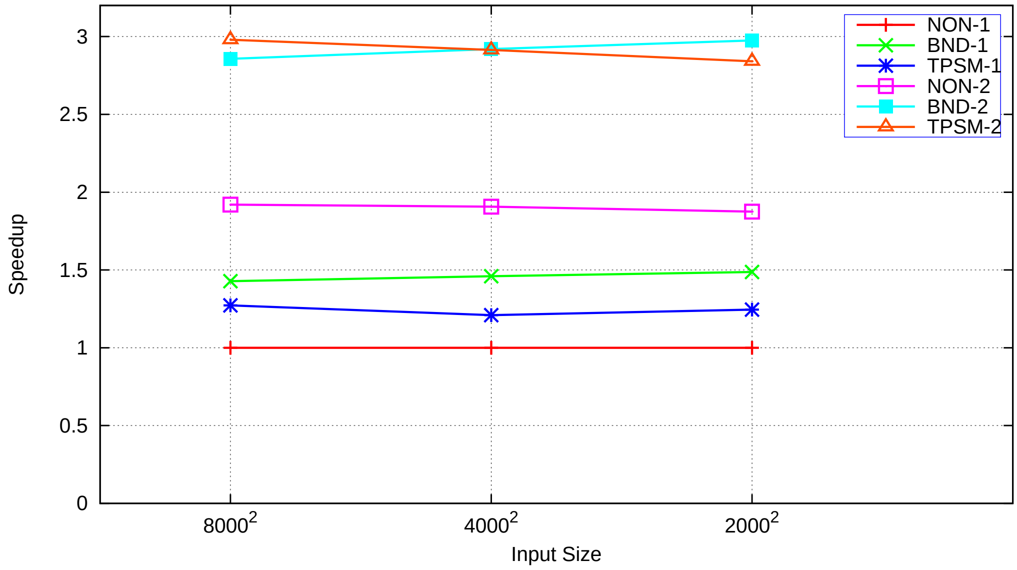


Figure 4.29: Sepia Filter Speedup Results in Tesla C2075

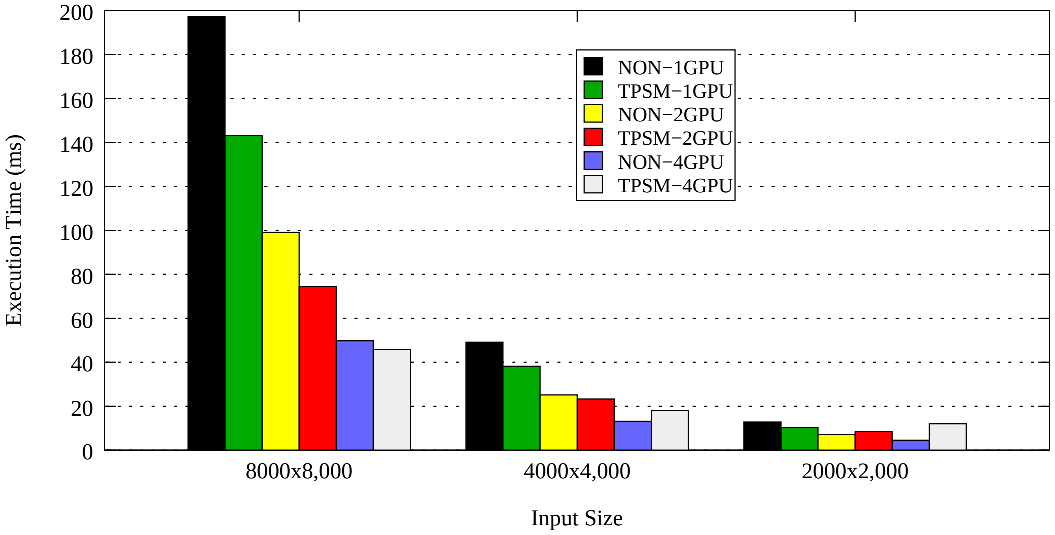


Figure 4.30: Sepia Filter Results in Tesla M2090

We applied sepia filter benchmark in Tesla M2090. The time cost results are shown in Figure 4.30 and the speedup results are shown in Figure 4.31. The data transfer part account for around 34.1% in the total time cost in single GPU case of Tesla M2090. In one GPU case, TPSM can achieve 1.45 ∼ 1.51 speedup while the lower bound speedup is 1.24 ∼ 1.37.

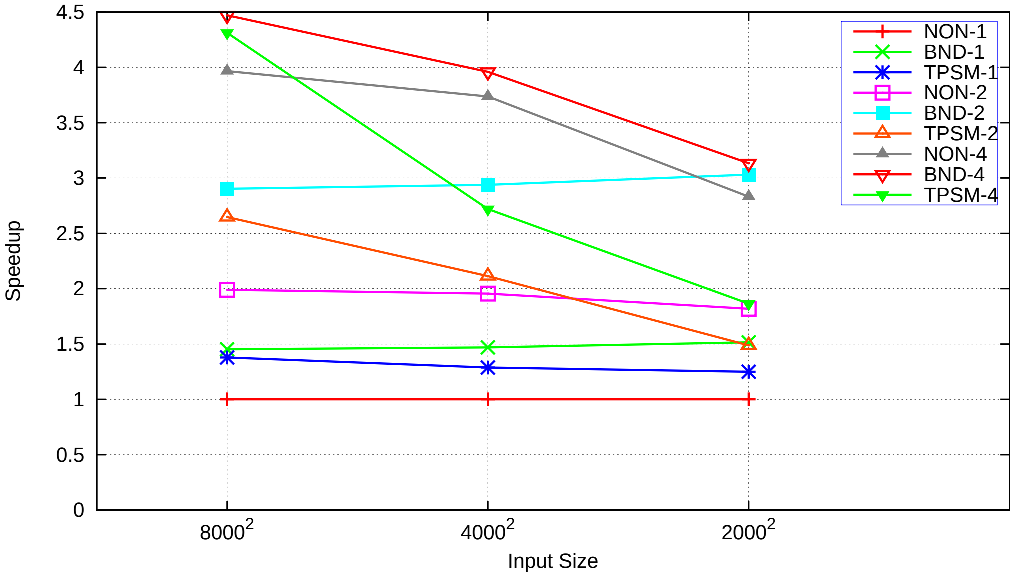


Figure 4.31: Sepia Filter Speedup Results in Tesla M2090

In dual GPU case, manual codes can achieve 1.81 ∼ 1.98 speedup which is almost twice speedup. The lower bound achieves 2.9 ∼ 3.03 speedup. TPSM achieves 2.65 speedup with input size 8000^2, 2.11 speedup with input size 4000^2 and only 1.49 speedup with input size 2000^2 which is even worse than the manual code without TPSM as the total time cost with input size 2000^2 is so small that is only several micro second.

In four GPU case, manual codes can achieve 3.97 speedup with input size 8000^2, 3.74 speedup with input size 4000^2 and 2.83 speedup with input size 2000^2. We can find the results of input size 8000^2 and 4000^2 are very close to four times speedup while input size 2000^2 is less than triple times speedup. This is because the total time cost of input size 2000^2 is very small. We find that it does not achieve higher performance with more GPU devices when the application is very light. The lower bound achieves 3.1 ∼ 4.47 speedup while TPSM achieves 1.81 ∼ 4.31 speedup. TPSM is more suitable for applications whose kernel execution is not too short.

We also use black-scholes benchmark with large data communication to test our work in three different platforms. We have five input size from 10^7 to 5×10^7.

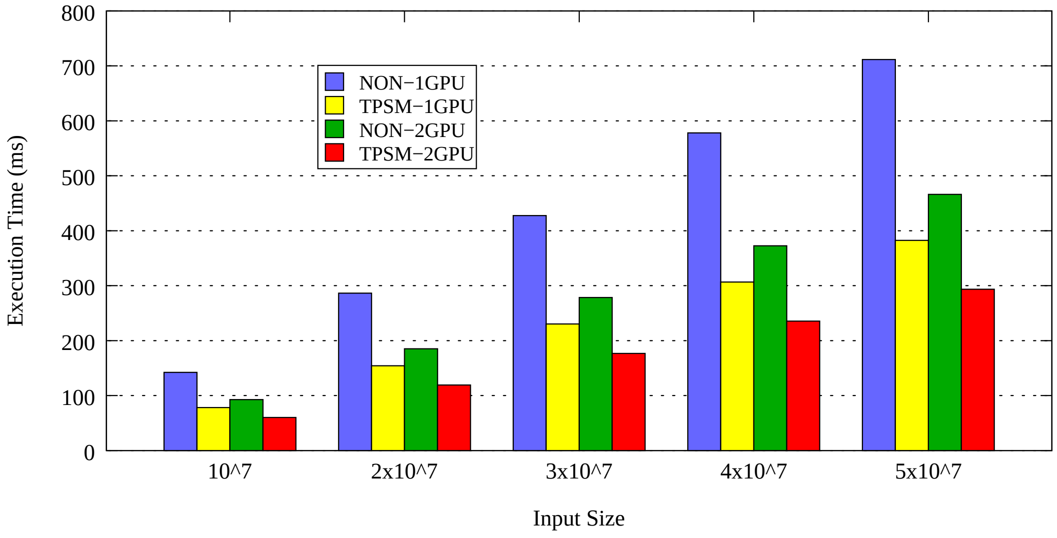


Figure 4.32: Black-Scholes Results in Tesla C2070

The time cost results of Tesla C2070 are shown in Figure 4.32 and the speedup results are shown in Figure 4.33. The data transfer part account for around 52.3% in the total time cost in single GPU case of Tesla C2070. Therefore, we use the TPSM for data transfer bound and receive heavy.

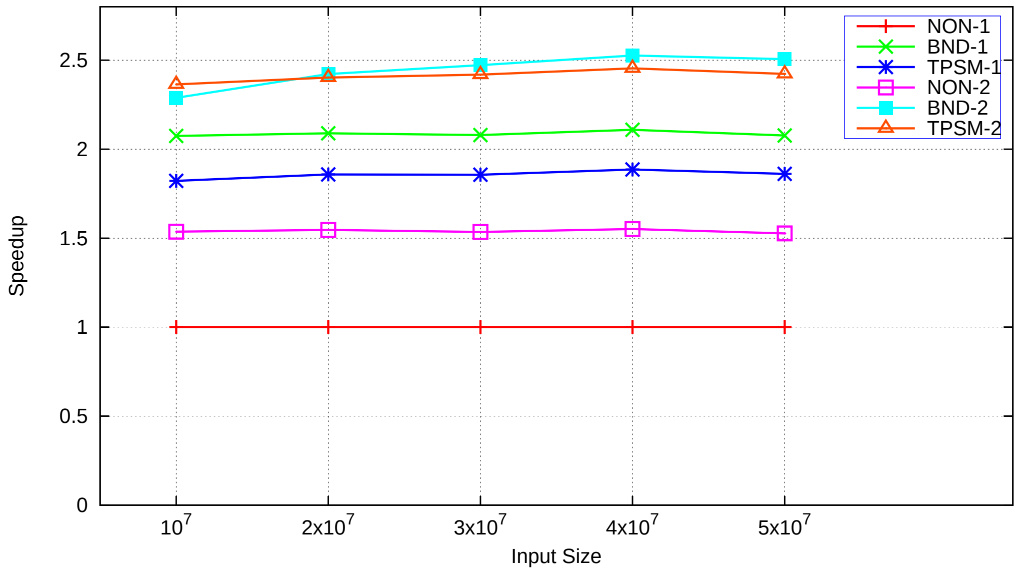


Figure 4.33: Black-Scholes Speedup Results in Tesla C2070

In one GPU case, TPSM can achieve 1.82 ∼ 1.88 speedup while the lower bound speedup is 2.07 ∼ 2.1.

In dual GPU case, manual codes can achieve 1.52 ∼ 1.55 speedup which is far away from ideal twice speedup. This is mainly because of the large communication data size and limited bandwidth between CPU and GPU. The lower bound achieves 2.28 ∼ 2.52 speedup while TPSM can achieve 2.36 ∼ 2.45 speedup. The results of TPSM are very close to the lower bound.

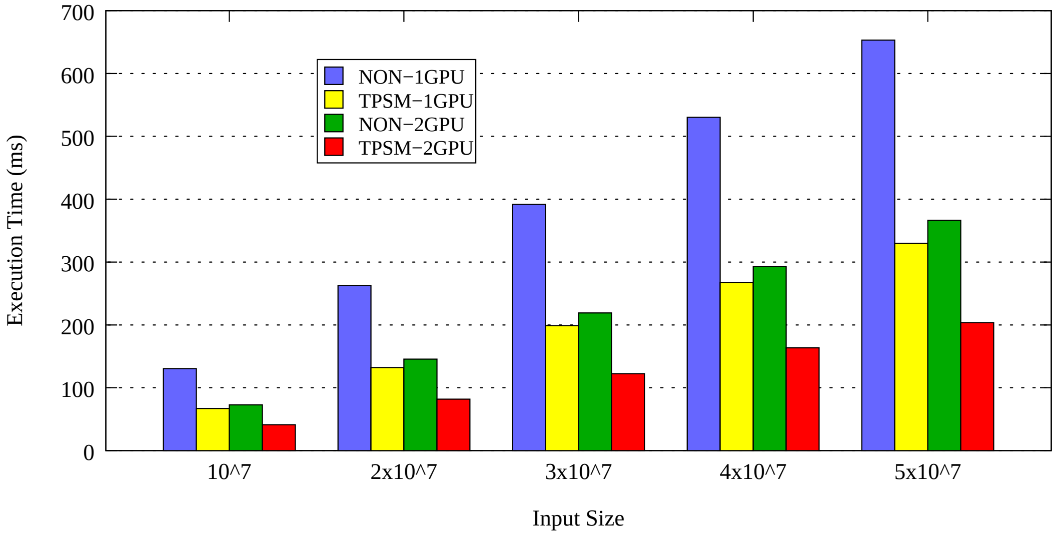


Figure 4.34: Black-Scholes Results in Tesla C2075

Then we applied black-scholes benchmark in Tesla C2075. The time cost results are shown in Figure 4.34 and the speedup results are shown in Figure 4.35.

The data transfer part of black-scholes account for around 51.7% in the total time cost in single GPU case of Tesla C2075. In one GPU case, TPSM can achieve 1.94 ∼ 1.98 speedup while the lower bound speedup is 1.93 ∼ 2.04. TPSM results are very close to the lower bound results and almost achieve twice speedup.

In dual GPU case, manual codes can achieve 1.78 ∼ 1.81 due to bandwidth limitation. The lower bound achieves 2.95 ∼ 3.3 speedup while TPSM can achieve 3.16 ∼ 3.24 speedup. TPSM results still are very close to the lower bound results.

We also applied black-scholes benchmark in Tesla M2090. The time cost results are shown in Figure 4.36 and the speedup results are shown in Figure 4.37. The data transfer part account for around 48.6% in the total time cost in single GPU case of Tesla C2075. In one GPU case, TPSM can achieve 1.52 ∼ 1.62 speedup while the lower bound speedup is 1.97 ∼ 2.04.

In dual GPU case, manual codes can achieve 1.97 ∼ 2.04 which is almost twice speedup. The lower bound achieves 3.43 ∼ 3.49 speedup while the speedup of TPSM

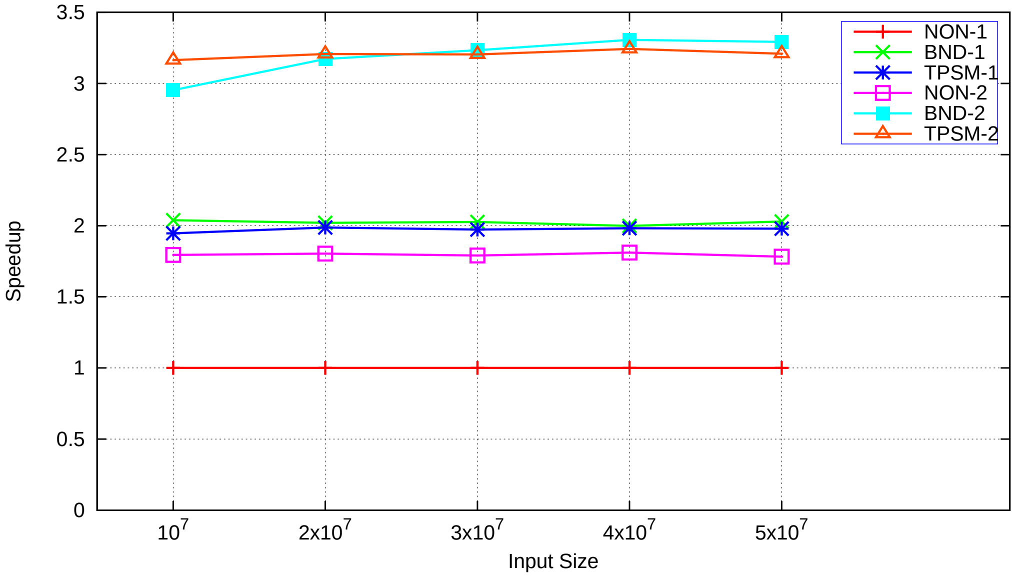


Figure 4.35: Black-Scholes Speedup Results in Tesla C2075

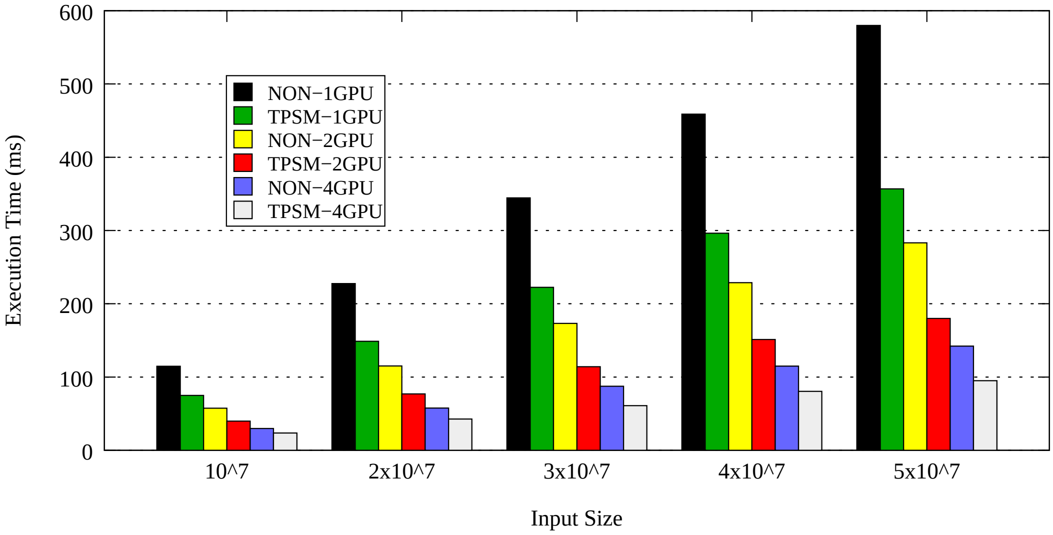


Figure 4.36: Black-Scholes Results in Tesla M2090

gradually increases from 2.88 to 3.22 along with the increase of the input size. This is mainly because of the fixed overhead for modeling.

In four GPU case, manual codes can achieve 3.86 ∼ 4.07 speedup which is around four times speedup. The lower bound achieves 6.35 ∼ 6.44 speedup while the speedup of TPSM increases from 4.86 to 6.1 along with the increase of the input size. For the maximum input size, TPSM result is very close to the lower bound result.

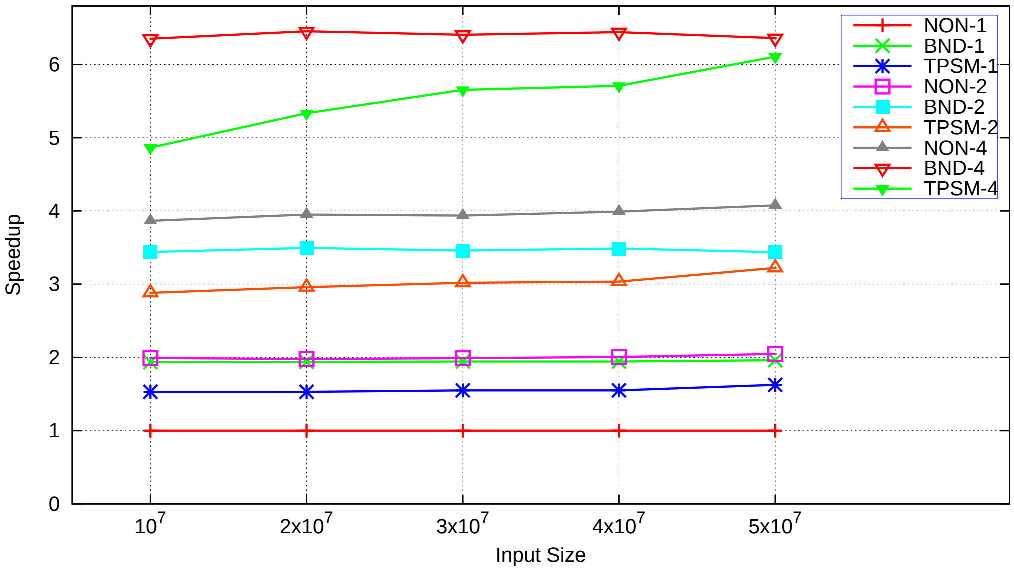


Figure 4.37: Black-Scholes Speedup Results in Tesla M2090

We applied matrix multiplication to test our work in three different platforms. As the number of SMs in three platforms is different, we use different input size for the platforms.

For Tesla C2070 and Tesla C2075, they have 14 SMs and each SM has 32 cores. For Tesla M2090, they have 13 SMs and each SM has 192 cores. Therefore, we set six input sizes $81920\times2\times3584$, $16384\times4\times7168$, $8192\times16\times7168$, $2048\times128\times7168$, $1433600 \times 256 \times 7$ and $358400 \times 512 \times 14$ for Tesla C2070 and Tesla C2075. We set seven input sizes $81920\times2\times4096$, $20480\times4\times8192$, $16384\times4\times8192$, $8192\times16\times8192$, $1638400 \times 256 \times 8$, $409600 \times 512 \times 16$ and $204800 \times 1024 \times 8$ for Tesla M2090. Here the input size $81920 \times 2 \times 3584$ means we calculate $A \times B = C$, A size is 81920×2, B size is 2×3584 and C size is 81920×3584.

First of all, we applied matrix multiplication in Tesla C2070. The time cost results are shown in Figure 4.38 and the speedup results are shown in Figure 4.39. The data transfer part accounts for 2.1% ∼ 24.4% with different input size in the

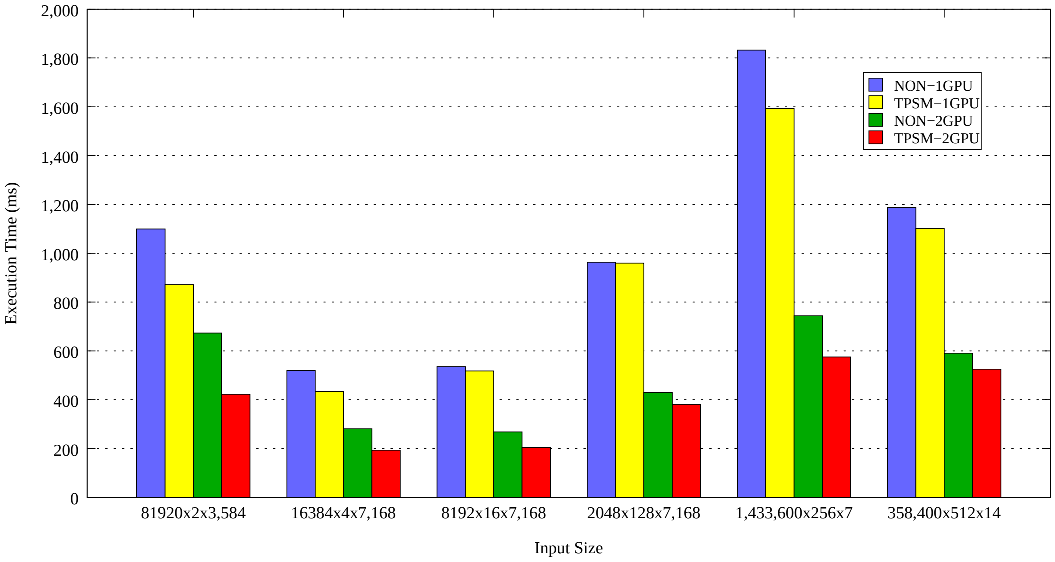


Figure 4.38: Matrix Multiplication Results in Tesla C2070

total time cost in single GPU. Therefore, we use the TPSM for kernel bound.

In single GPU case, TPSM can achieve 1.01 ∼ 1.32 speedup while the lower bound speedup is 1.004 ∼ 1.26. TPSM results are very close to the results of the lower bound.

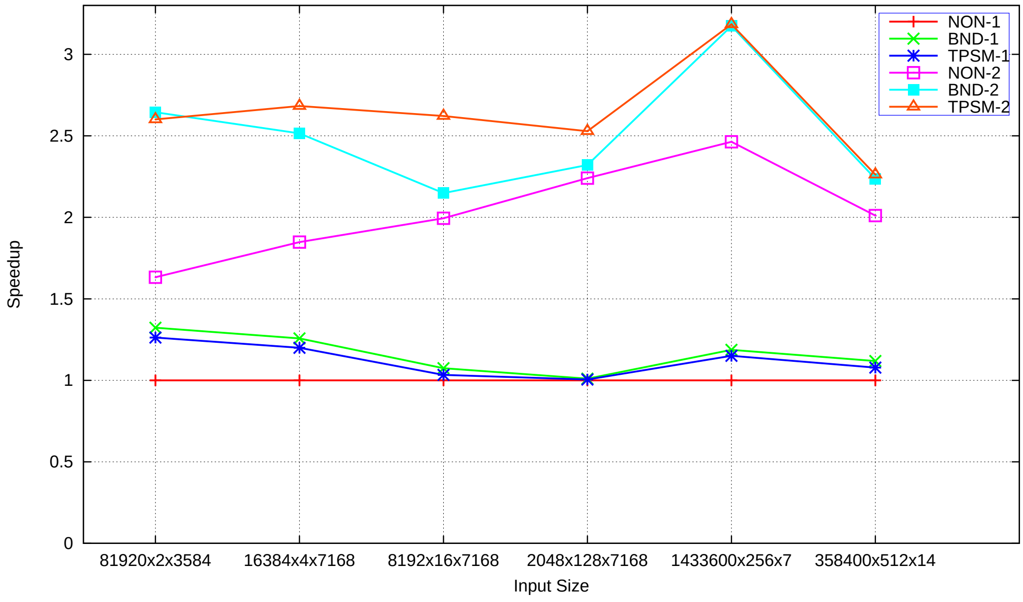


Figure 4.39: Matrix Multiplication Speed Results in Tesla C2070

In dual GPU case, manual codes can achieve 1.63 ∼ 2.46 that appears super speedup. This is many because the input data are divided and loaded in two GPUs which may cause the change of memory access pattern and may increase or decrease the performance of GPU local memory access. TPSM can achieve 2.26 ∼ 3.18 speedup while the lower bound speedup is 2.14 ∼ 3.17. The results of TPSM are very close to the results of the lower bound in Tesla C2070.

We also applied our work to Tesla C2075 platform. The time cost results are shown in Figure 4.40 and the speedup results are shown in Figure 4.41.

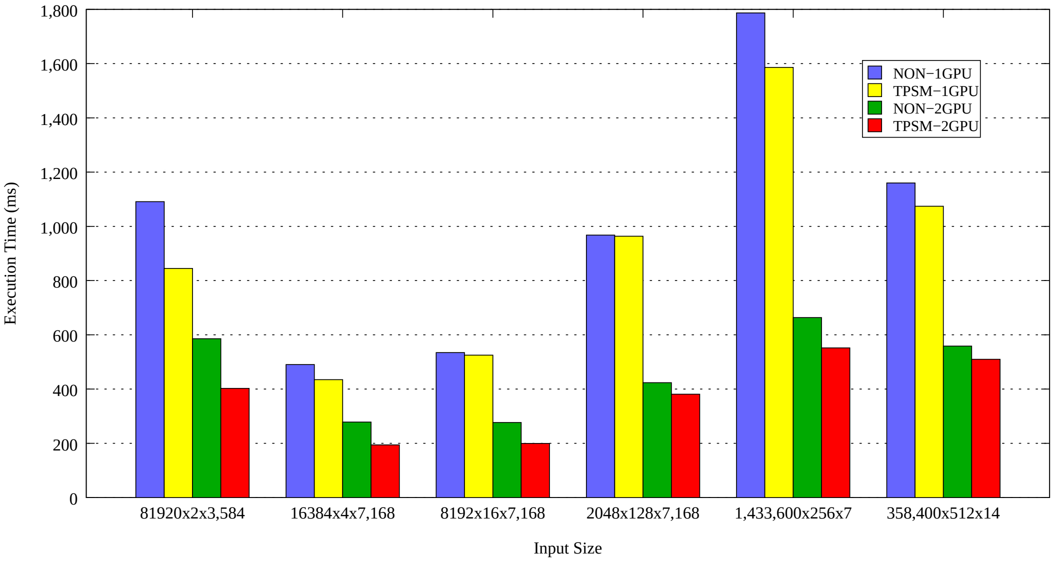


Figure 4.40: Matrix Multiplication Results in Tesla C2075

The data transfer part of matrix multiplication accounts for 1.04% ∼ 24.42% with different input size in Tesla C2075. In single GPU case, TPSM can achieve 1.025 ∼ 1.26 speedup while the lower bound speedup is 1.01 ∼ 1.32. In dual GPU case, manual codes can achieve 1.86 ∼ 2.69 speedup. TPSM can achieve 2.28 ∼ 3.04 speedup while the lower bound speedup is 2.22 ∼ 3.11. TPSM can well hide the communication latency in Tesla C2075 as well.

Then we applied our work to Tesla M2090 platform. The time cost results are shown in Figure 4.42 and the speedup results are shown in Figure 4.43. The data transfer part accounts for 9.67% ∼ 23.06% with different input size.

In single GPU case, the lower bound speedup is 1.107 ∼ 1.29 while TPSM can achieve 1.06 ∼ 1.35. The results of TPSM is very close to the results of the lower bound and even better for some input size. This is because TPSM partitions the application into smaller subtask which change the memory access pattern and may increase or decrease the performance.

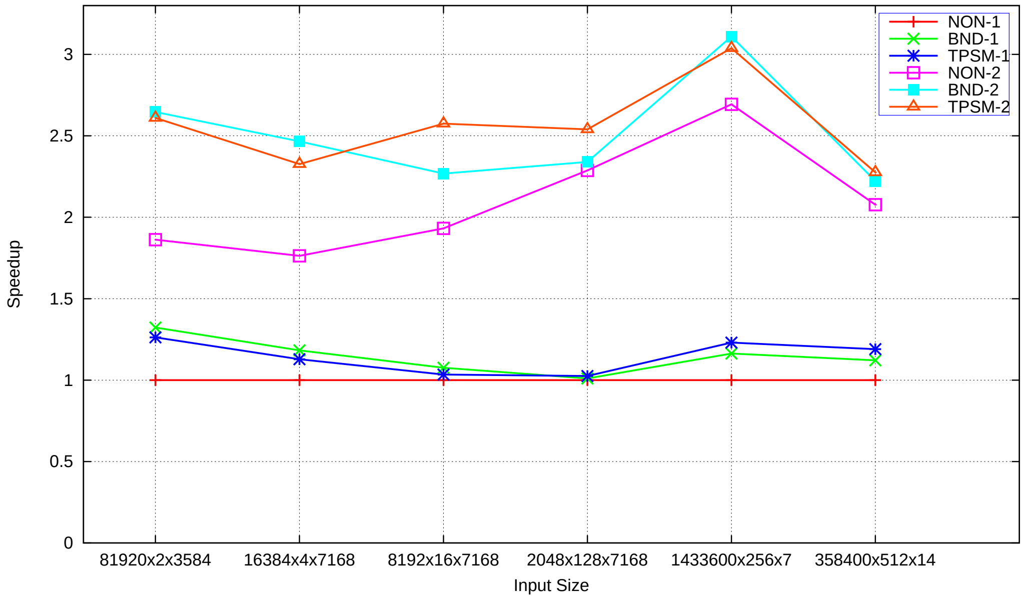


Figure 4.41: Matrix Multiplication Speed Results in Tesla C2075

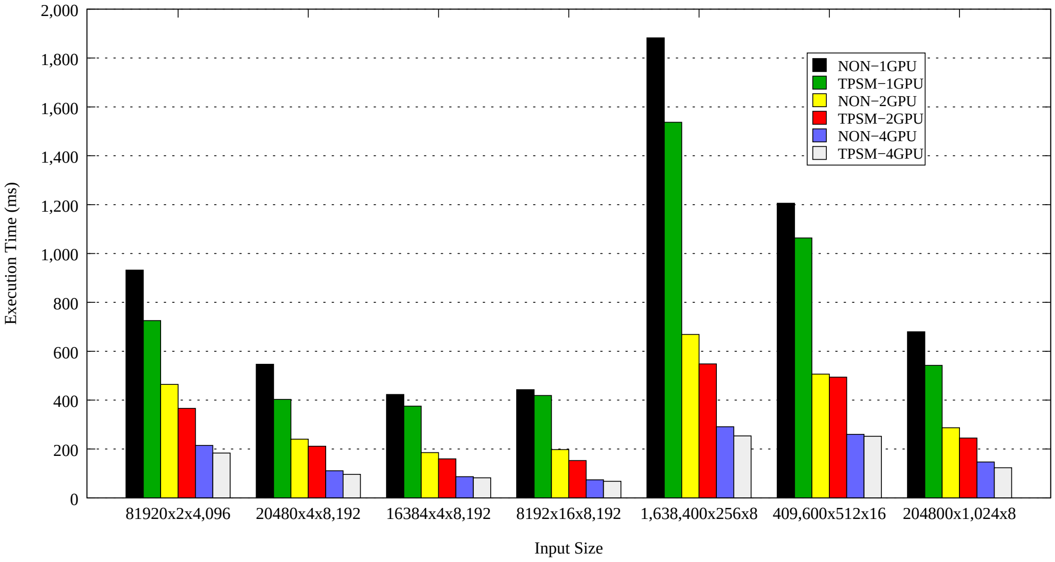


Figure 4.42: Matrix Multiplication Results in Tesla M2090

In dual GPU case, manual codes can achieve 2 ∼ 2.81 speedup which is even better than the ideal twice speedup. We find that the performance of matrix multiplication benchmark can be greatly affected by the memory access pattern. We can change the memory access pattern by designing the output computing order. However, the partitioning operation will greatly change the memory access pattern. Therefore, it is difficult to design a good and fare output computing order for all comparative scenes. Hence, we choose sequential output for all scenes which might cause performance fluctuation. For lower bound, the speedup is 2.46 ∼ 3.41 while TPSM can achieve 2.44 ∼ 3.43. TPSM almost achieve the same performance as the lower bound.

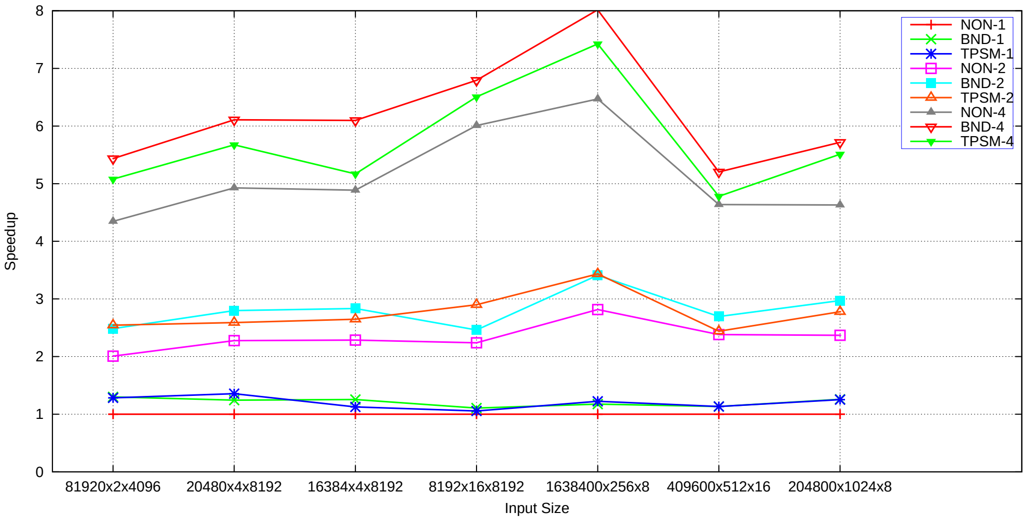


Figure 4.43: Matrix Multiplication Speedup Results in Tesla M2090

In four GPU case, manual codes can achieve 4.34 ∼ 6.46 speedup which is much better than the ideal four times speedup. The main reason is the change of memory access pattern. TPSM can achieve 4.78 ∼ 7.42 speedup while the lower bound speedup is 5.2 ∼ 8.02. TPSM can well hide the communication latency in Tesla M2090.

As there are some works to use equal size subtask for overlapping, we also compare our work to equal size subtask overlapping with black-scholes benchmark in Tesla C2075. We partition the application into 10 subtasks(10-T), 100 subtasks(100-T), 1000 subtasks(1000-T), 5000 subtasks(5000-T) and 10000 subtasks(10000-T) and compare the results with no partitioning(1-T) and our TPSM results. The time cost results are shown in Figure 4.44 and the speedup results are shown in Figure 4.45. We can find that our TPSM achieves better performance than all equal size scenes. This is because our method can well hide the communication latency than equal size overlapping method. Also notice that, the performance with 5000 subtasks and 10000 subtasks for some input sizes are even worse than the performance without overlapping. This is because too many subtasks will lead to heavy synchronization overhead such as thread launch. Although the performance for 100 subtasks is close to our method performance in some input sizes, the performance of 100 subtasks varies a lot along with the input size. Therefore, the simple equal size overlapping can hide parts of the communication latency but can not well hide the communication latency. Our method is asymptotic optimal method and can well hide the communication latency.

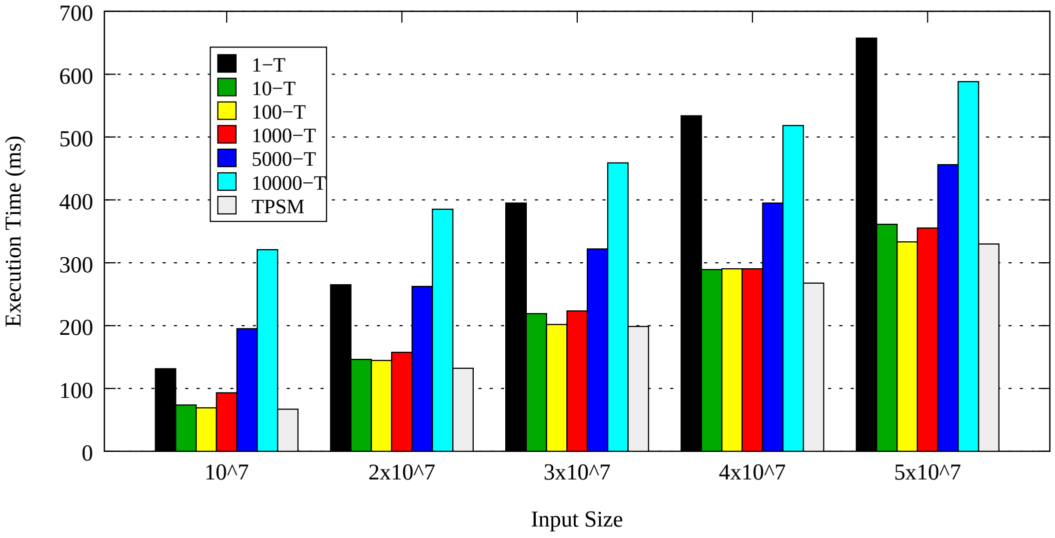


Figure 4.44: Black-Scholes Results with equal size subtasks in Tesla C2075

4.5.2 Non-symmetric Architecture

4.5.2.1 Configuration

For non-symmetric architecture, we install one Tesla C 2075 and Tesla K20c in AT38. The peak Tflops/s of single precision of Tesla C2075 is 1.03 while the peak Tflops/s of Tesla K20c is 3.5. Tesla K20c is much faster than Tesla C2075. We use four benchmark used in previous sections to test our work in the non-symmetric architecture. We repeat the experiment in four scenes and the abbreviations used in the Figures are as follow:

2075: the results of using single Tesla C2075 without TPSM;

$K20c$: the results of using single Tesla K20c without TPSM;

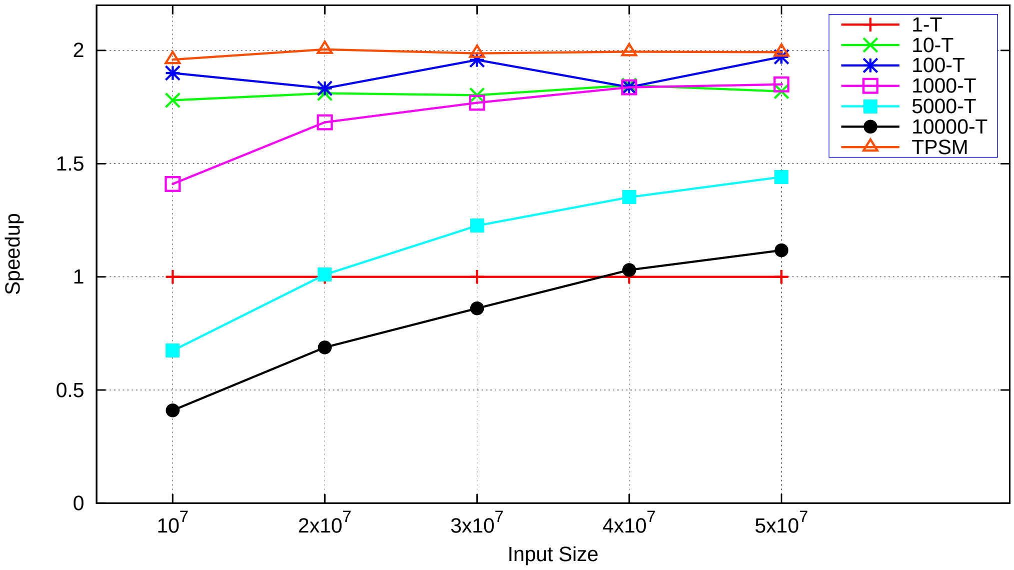


Figure 4.45: Black-Scholes Speedup Results with equal size subtasks in Tesla C2075

$2GPU$: the results of using both Tesla C2075 and K20c without TPSM (with load balance awareness);
$TPSM1$: the results of using both Tesla C2075 and K20c with TPSM and using performance analytical model in the first block (20 blocks);
$TPSM2$: the results of using both Tesla C2075 and K20c with only TPSM.

4.5.2.2 Results

First of all, we apply the linear filter benchmark in the non-symmetric architecture to test our work. We have three input size:8000^2 pixels, 4000^2 pixels and 2000^2 pixels. The time cost results are shown in Figure 4.46. We also compare the speedups to single Tesla C2075 case(the slower GPU) and the speedup results are shown in Figure 4.47.

With one GPU, Tesla K20c achieves 1.42 speedup for input size 8000^2, 1.33 speedup for input size 4000^2 and 1.26 speedup for input size 2000^2 comparing to Tesla C2075. We can find that the speedup decreases along with the decrease of input size. This is because the kernel time almost linearly decreases along with the decrease of the input size. However the overhead such as thread launching and data transfer preparation does not decrease linearly along with the decrease of the input size.

For two GPUs without TPSM, the speedup is from 2.1 to 2.45 comparing to single Tesla C2075. The speedup for all input size is more than twice speedup

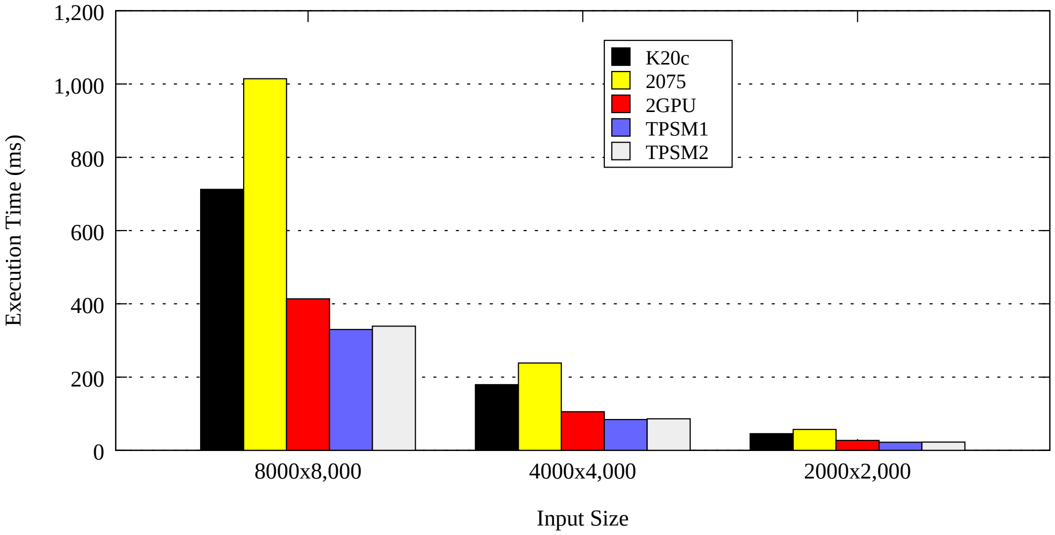


Figure 4.46: Linear Filter Results in Non-symmetric Architecture

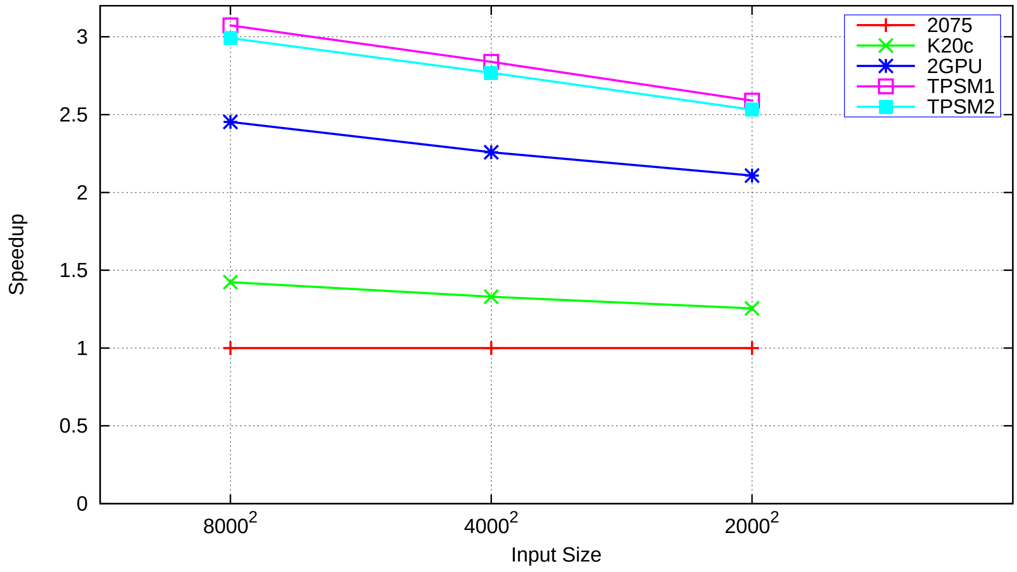


Figure 4.47: Linear Filter Speedup Results in Non-symmetric Architecture

comparing to single Tesla C2075 and less than twice speedup of Tesla K20c. This is because the load is evenly allocated between Tesla C2075 and Tesla K20c. Therefore, the performance should be better than the performance of dual Tesla C2075 and poorer than the performance of dual Tesla K20c.

With two GPUs, TPSM1 can achieve 2.59 $\sim$ 3.07 speedup which is even better than twice of the speedup of single Tesla K20c. The main reason is that we not only consider about load balance between two GPUs but also use TPSM to hide the communication latency between GPU and CPU. Therefore, we can achieve more than twice speedup of the fastest GPU in the non-symmetric architecture. TPSM2 can achieve a speedup which is a little slower than TPSM1.

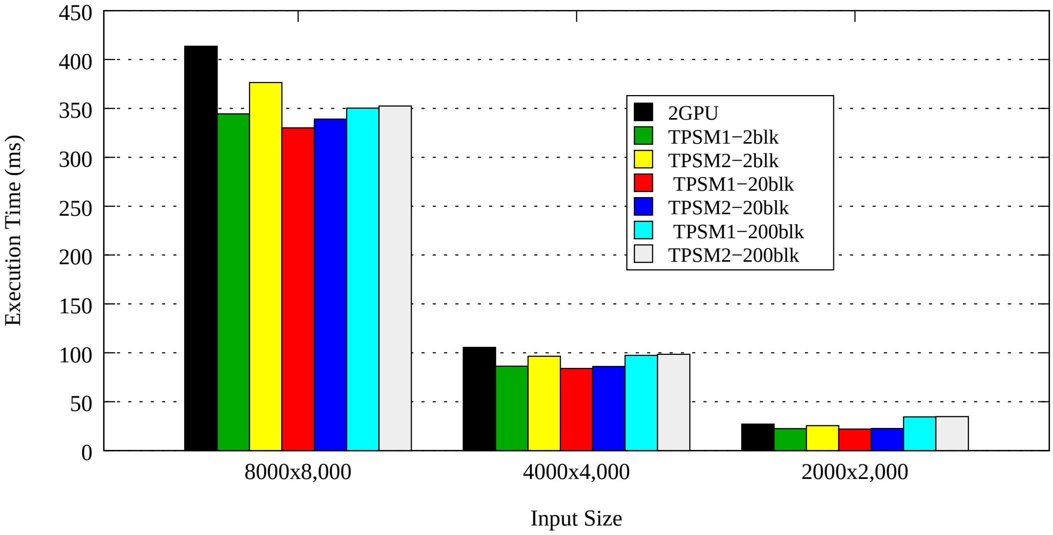


Figure 4.48: Linear Filter Results with Different Block Number in Non-symmetric Architecture

The block number used in the first layer partitioning can greatly affect the performance. To understand the affection, we change the number of block from 2 blocks to 200 blocks and compare the performance for TPSM1 and TPSM2. The time cost results are shown in Figure 4.48 and the speedup results are shown in Figure 4.49. With 2 blocks, TPSM1 can achieve 1.21 speedup while TPSM2 achieves 1.108 speedup. As the block number is small, the load allocation in the first block is very important and can greatly affect the total time cost. With 20 blocks, the distance between TPSM1 and TPSM2 is closer than the scene of 2 blocks. This is because the proportion of the first block in the total time cost becomes lower. With 200 blocks, the performance of TPSM1 and TPSM2 are very close. This is because the time cost of the first block only accounts for a tiny part of the total time cost.

The improvement by using the performance analytical model is very limited. Notice that the performance decrease rapidly along with the decrease of input size. This is because the kernel becomes small with small input and large block number will lead to smaller kernel for each GPU overlapping. As we have mentioned that our TPSM method is not suitable for the applications with small kernel execution. Therefore, the performance of our method can be very poor with too large block number for the first layer partitioning as the kernel for each GPU can be very small. Therefore, a suitable block size is also very important. We suggest to adjust the block size according to the time cost of applications. The time cost of the execution with one block is better to be 10 $\sim$ 50 millisecond level.

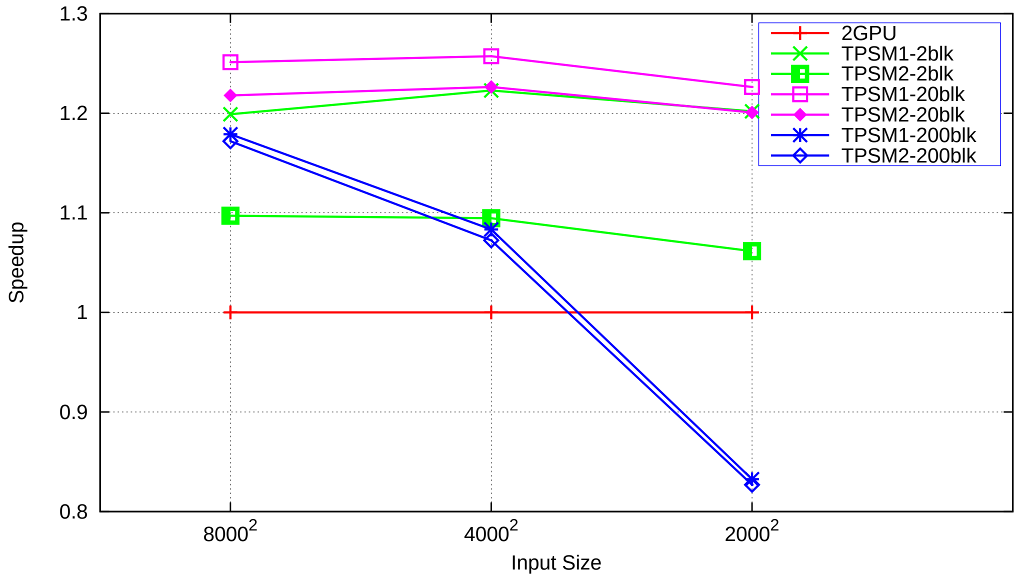


Figure 4.49: Linear Filter Speedup Results with Different Block Number in Non-symmetric Architecture

We also apply sepia filter benchmark to test our work. The execution time results are shown in Figure 4.50 and the speedup results are shown in Figure 4.51. The time cost of data transfer part accounts for about 31% in the total time cost in Tesla C2075 as discussed earlier.

For single GPU case, Tesla K20c achieves 1.37 speedup for input 8000^2, 1.33 speedup for input 4000^2 and 1.31 speedup for input 2000^2. The speedup decreases along with the decrease of input size. The reason is the same as in linear filter case.

For two GPU case without TPSM, the speedup is from 1.95 to 2.12 Notice that the speedup of input size 2000^2 is less than twice. This is because the data transfer part can hardly achieve twice speedup due to bandwidth limitation. Although the

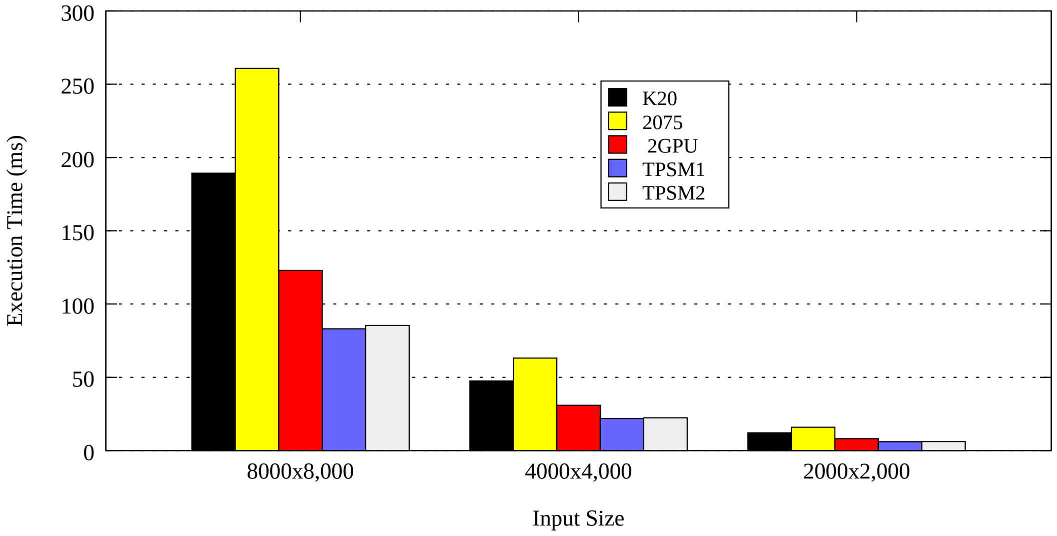


Figure 4.50: Sepia Filter Results in Non-symmetric Architecture

kernel part can achieve more than twice speedup with dual GPUs, high proportion of data transfer part in the total time cost will lead to total speedup low. Moreover, considering the affection of the fixed overhead for small input size, it is possible to have less than twice speedup even with one faster GPU in the dual GPU non-symmetric architecture.

For two GPU case in TPSM1, our work can achieve 2.61 $\sim$ 3.14 speedup which is much better than the speedup of two GPU without TPSM. The benefit mainly comes from well hiding the communication latency between GPU and CPU as well as suitable load allocation between GPUs. TPSM2 can also achieve a speedup which is a little slower than TPSM1.

We apply black-scholes to test our work as well. The execution time results are shown in Figure 4.52 and the speedup results are shown in Figure 4.53.

In single GPU case, Tesla K20c achieves speedup from 1.17 to 1.2. The speedup decreases a little along with the decrease of input size and the speedup is much smaller comparing to linear filter and sepia filter. This is because the data transfer part accounts for around half of the total time cost. Although Tesla K20c is much faster than Tesla C2075, the bandwidth of them are almost the same. Therefore, we can only achieve little speedup even with a much faster GPU device.

For two GPU case without TPSM, the speedup is 1.99 $\sim$ 2.01 which is around twice speedup comparing to single Tesla C2075. The performance of dual GPUs with one Tesla C2075 and one Tesla K20c should be faster than twice of single Tesla C2075 and slower than twice of single Tesla K20c. Here we only achieve

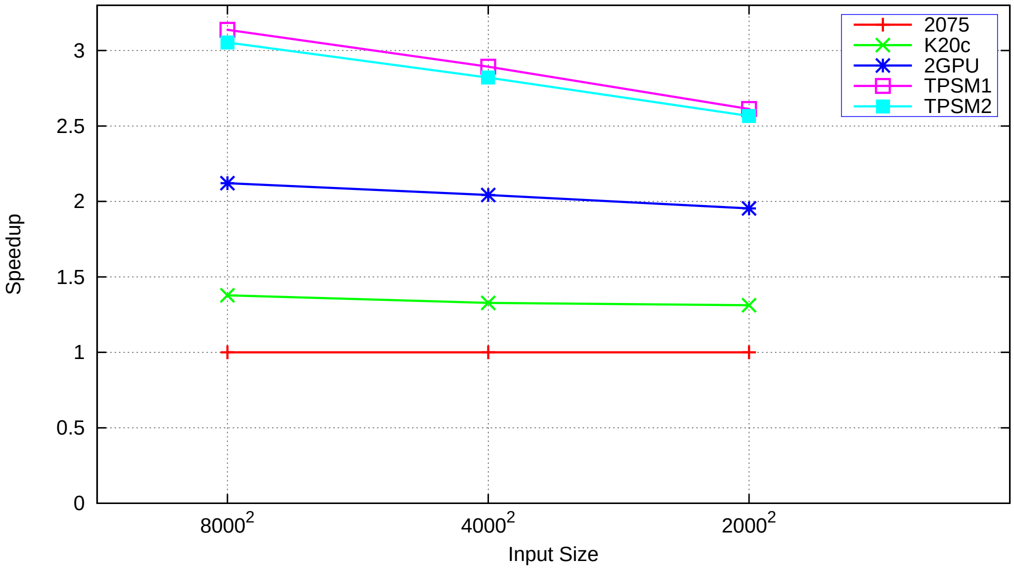


Figure 4.51: Sepia Filter Speedup Results in Non-symmetric Architecture

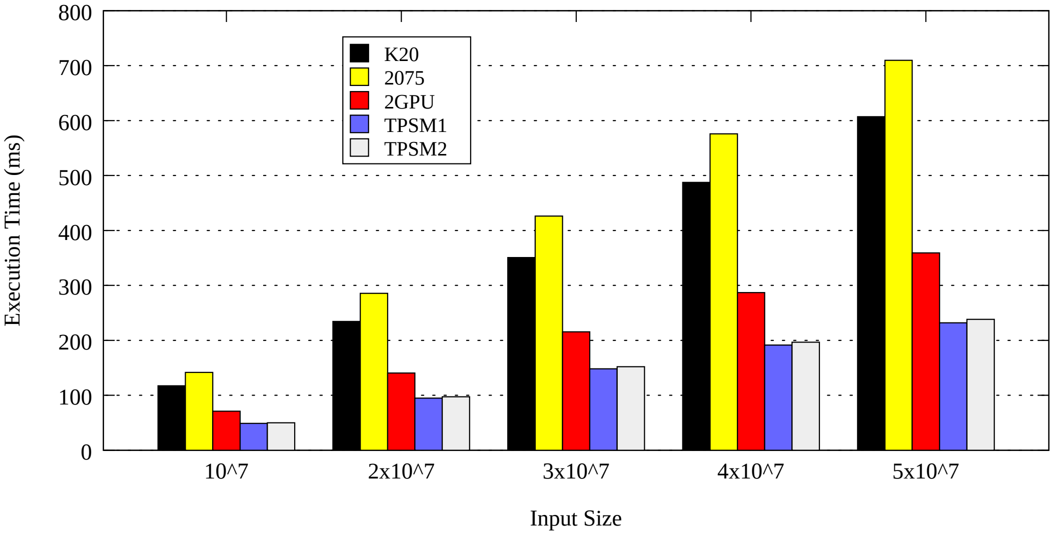


Figure 4.52: Black-scholes Results in Non-symmetric Architecture

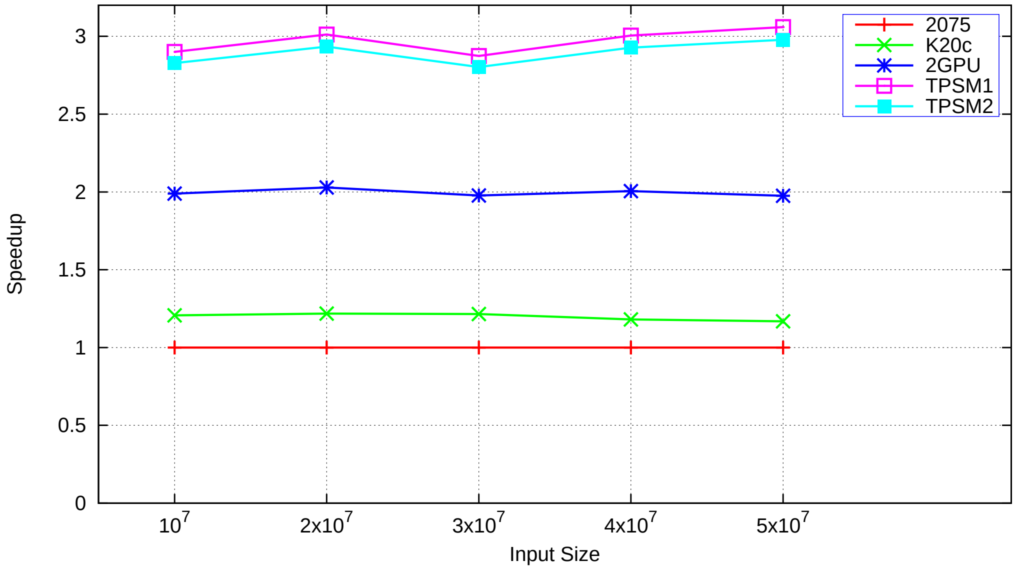


Figure 4.53: Black-scholes Speedup Results in Non-symmetric Architecture

twice speedup of single Tesla C2075 that is because of the bandwidth limitation as mentioned in above. For two GPU case in TPSM1, we can achieve 2.9 ∼ 3.06 speedup which is much better than the performance of dual GPU without TPSM. The main reason is that we well hide the communication latency in each GPU and well handle the load balance. TPSM2 can achieve a speedup which is a little slower than TPSM1.

Finally, we use matrix multiplication benchmark to test our work. The execution results are shown in Figure 4.54 and the speedup results are shown in Figure 4.55. For single GPU case, Tesla K20c achieves 1.08 ∼ 1.24 speedup comparing to single Tesla C2075. We can find that the speedup of Tesla K20c with input size $1433600 \times 256 \times 7$ is the lowest due to its highest data transfer part proportion. For two GPU case without TPSM, the speedup varies a lot from 1.85 to 2.71. The fluctuation is mainly because the proportion of data transfer part in the total execution varies a lot along with the input size. In the other aspect, the reallocation of the application will change the memory access pattern which may greatly affect the kernel execution performance.

For two GPU case in TPSM1, we can achieve 2.19 ∼ 3.55 speedup which is much better than the case without TPSM. The results show we can not only hide the communication latency between GPU and CPU but also well handle the load balance between GPUs. TPSM2 can achieve a speedup which is a little slower than TPSM1.

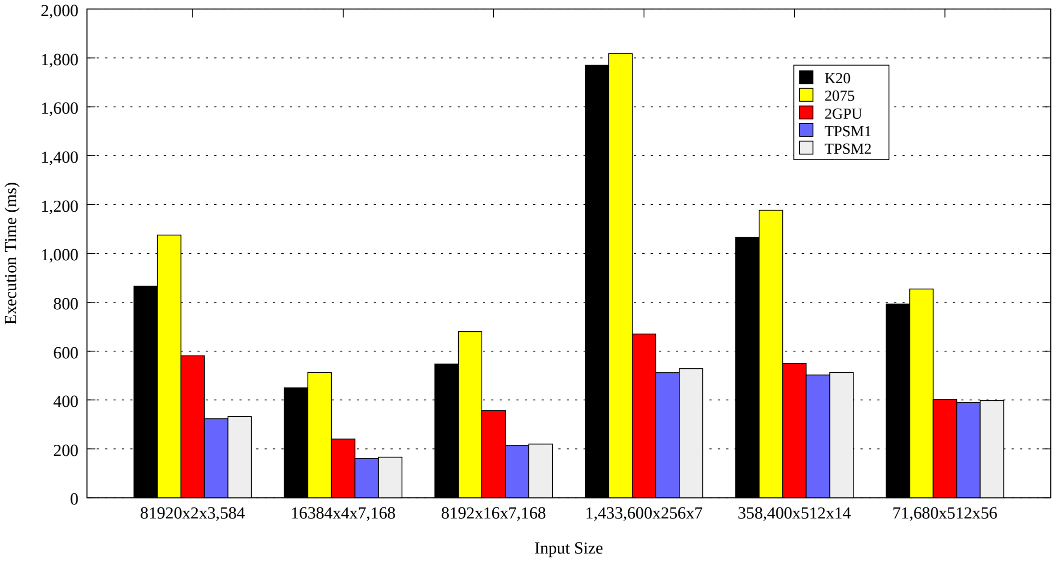


Figure 4.54: Matrix Multiplication Results in Non-symmetric Architecture

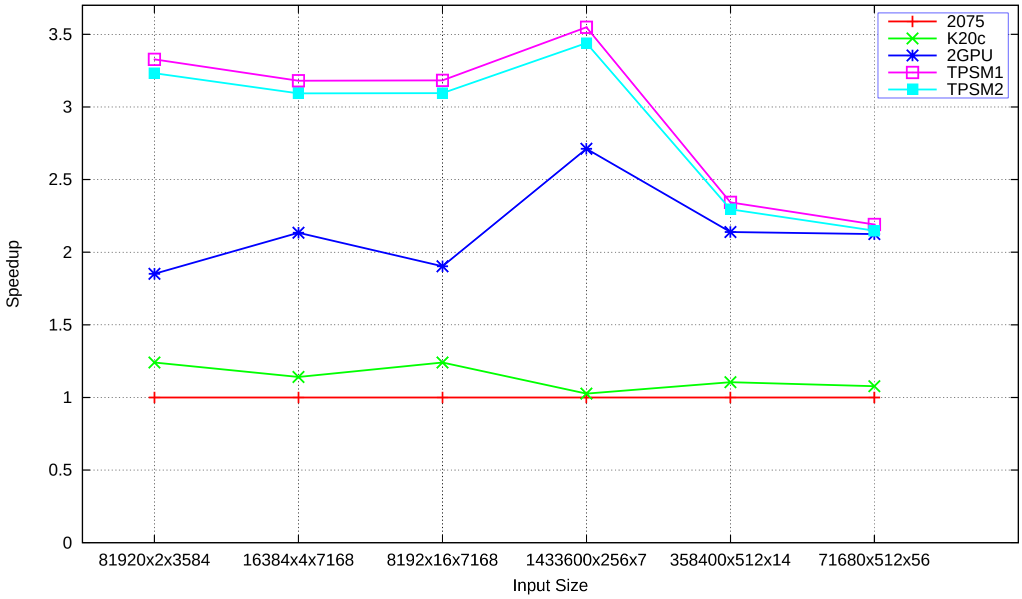


Figure 4.55: Matrix Multiplication Speedup Results in Non-symmetric Architecture

Chapter 5

Related Works

Contents

5.1 GPU Performance Prediction

In the past years, performance prediction has been widely studied in parallel and sequential systems [31] [82][4][117]. The parallel algorithms community has provided several models for design and analysis of parallel algorithms such as Log-P [38] and QRQW[49]. These models can help programmers to find out the problem in the parallelism. However, they are mostly architecture independent which makes them have little help for an insight into a specific architecture.

Since the emergence of GPUs, there are also some work about predicting performance for them [4] [6] [53] [54] [69].

The studies on GPU performance prediction can be classified in to two categories: analytical models and GPU simulators.

5.1.1 GPU Analytical and Performance Models

Ma et all[79] proposed a memory access model called *Threaded Many-core Memory* (TMM) model to analyze factors that affect performance. Their work main focus on how to hide the memory latency within the device memory. They work is only applicable for four algorithms for the problem of All Pairs Shortest Paths (APSP).

They also provided another performance model in [80]. They developed an analytical performance model for memory-limited kernel to help configure the tuning parameters. This work also focus on the optimization in the device memory.

Hong and Kim[53] proposed a model which can predict the execution time of a kernel on a GPU with a set of 23 parameters. Although we have a similar approach, there are still some differences between the two works. First of all, we provide an insight to the assembly codes of CUDA programs for the performance analysis. We analyze the PTX codes with instruction-level parallelism awareness to obtain a high accuracy results. Secondly, we take the parallelism execution of computation instructions into consideration.

Sim and Aniruddha[103] [104] proposed a performance analysis framework to help find out the performance bottlenecks in current code and estimate the potential performance benefits from removing the bottlenecks.

Zhang and Owens[133] proposed a benchmark-based performance model for Nvidia GeForce 200-series GPUs to identify the performance bottlenecks and predict potential benefits with analysis the instruction pipeline, shared memory access and global memory access. Their model has 5% $\sim$ 15% error rate. Our work can achieve better prediction than their work. Moreover, out work can be applicable for many kind of GPUs rather on only GeForce 200-series.

Kishore and Rishabh[69] developed a performance prediction model for CUDA program which combine many known models of parallel computation such as BSP, PRAM and QRQW. From their experiment results, the prediction accurate rate of their model is not as good as our work.

Jia and Zhang[62] proposed a model for guiding performance optimizations on GPUs named GPURoofline. Their model is an empirical model for guiding optimizations in GPUs to help identify bottlenecks and provide optimization methods.

Kerr and Diamos[64] proposed an emulation and translation infrastructure based on Ocelot to characterize GPU workload and predict relative performance in GPU or CPU. They did the empirical evaluation of 25 CUDA applications on four GPUs and three CPUs. The GPU results are no as good as our work.

5.1.2 GPU Simulators

Another method for GPU performance prediction is using simulators [34] [63] [6]. Most of the GPU simulators aim at the CUDA platform and perform simulation for PTX or native GPU code. *Parallel Thread Execution* (PTX) provided by Nvidia is a virtual instruction set architecture with clear data parallel semantics for CUDA architecture.

Collange [34] develop a GPU functional simulator for Nvidia Tesla GPUs named

Barra. Barra is implemented based on UNISM which is a modular simulation framework. The advantage of Barra is that it provides cycle accurate performance prediction and enable users to monitor all GPU activities. However, the emulation time cost can be very long. Moreover, it is not very flexible as the binary instruction set can be changed in the next generation.

Keer [63] provides another GPU simulator framework named Ocelot. Ocelot provides an emulation and compilation infrastructure which achieve the CUDA runtime API. Inside ocelot, there is a virtual machine which can emulate PTX instructions. With this features, Ocelot enables the emulation to different architectures. Besides GPU performance prediction, Ocelot can also calculate control and data dependencies by collecting instruction and memory traces.

Bakhoda[6] provides another complex GPU simulator named GPGPU-Sim. They aim to enable users to do experiments with different GPU architectures and easily find the design space. They emulate the PTX instruction set and closely follow the CUDA architecture by which achieve the goal. Moreover, GPGPU-Sim also enable cycle accurate performance predictions and enable the change of several architecture details.

The main difference between our model and other work is that our model take instruction-level and thread-level into consideration to achieve good prediction accuracy.

5.2 Communication Latency Hiding

5.2.1 Divisible Load Scheduling

As many parallel applications can be partitioned into smaller tasks[102], divisible load theory has been used into different fields such as large-data management, image and video processing, biology and network applications. In addition, the implementations of divisible load theory have ranged from homogeneous and heterogeneous clusters to cloud environment. Here, we present and overview some of the works.

Most of studies on divisible load theory are performed on homogeneous and heterogeneous. For example, Drozdowski and Wolniewicz[41] adopted four different test beds (pattern search, database join, file compression and graph coloring and genetic search) to verify the efficient of divisible load theory. The proved that divisible load model can accurately describe the reality for each test bed. However, they also pointed out that the predictions obtained from the model were not satisfied for some data-dependent loads such as genetic search. This was because the references to disk files or memory allocation procedures leads to great amount of uncertainty and dependence. In such cases, the assumption on linear dependence of communication time on the volume of data was not performed where the communication speed decreased along with the increase of data size.

The work from Kim[65] was also for homogeneous clusters. They improved the initial work from Cheng and Robertazzi[28]. Their proposal reduced communication times by saving only its specific load instead of having a duplicated record of the whole load in each processing unit.

Lin[74][76][75] studied real-time scheduling algorithms and the influences of design parameters. They combined divisible load theory with an approach that considered the earliest deadline to finish a task to enhance the service quality in cluster. They pointed out that the execution of partitioned subtasks in a homogeneous cluster where processors have different available times can lead to lower completion time comparing to the estimated.

Lin also identified three important design decisions which referred to workload partitioning, node assignment and task execution order regarding real-time divisible load scheduling in later work[77]. Therefore, they proposed a scheduling framework which enabled to configure different policies for each scheduling decision. Chuprat and Baruah devised efficient algorithms in their work[30] to determine the minimum number of processors to complete a job based on its deadline and determined the earliest completion time of a job on a specific number of processors.

Veeravalli and Ranganath[113] used the divisible load paradigm to schedule processing of an image onto homogeneous and heterogeneous processors. They aimed to minimize the total processing time of the whole image submitted to the bus network system. They took edge detection as an example of image processing applications. These applications qualified to use a divide-and-process strategy where initial load can be partitioned into smaller independent data chunks and hence was supported by divisible load theory.

Drozdowski and Wolniewicz[42][43] used scheduling divisible loads on a distributed computing system with physical restrictions such as limited available memory. They took communication latency and the system heterogeneity into consideration. Their study problem was to find a distribution of the load with minimum communication and computation time cost. They tested their method on star network systems and demonstrated that memory limitations did not restrict efficiency of parallel processing as much as computation and communication speed did in many cases.

Brest and Žumer[23] tried to improve the total execution time for applications developed in a Master-Worker pattern by using divisible load theory. Their method partitioned the program into computationally homogeneous subtasks which can be of different size based on the current load in each machine in the heterogeneous computing system. They evaluated their work by using continuous speech recognition problem and the asymmetric traveling salesman problem and had promising results on improving the total execution time of the applications. Besides, they also reported the influence of the initial data size partitions on the overall time cost.

Beaumont and Legrand[13][14] proposed an approach to schedule divisible workloads on heterogeneous systems. They proposed a multi-round method for resources selection based on the speed of the processors and communication links. Notice that the situations they tackled had not too high communication-to-computation ratio. Similarly, Yang and Casanova[123][122] studied the distribution of divisible load in heterogeneous distributed systems by using parallel applications designed in a Master-Worker pattern. They sent load to each worker with multi-round scheduling and sent several data chunks rather than single one. To solve this specific divisible load scheduling problem, they defined the worker subsets that should be used, the communication sequence to these workers and the sizes of each chunks.

It was Ko and Robertazzi[68] that firstly introduced an equal-size allocation scheme for divisible load. They considered equal allocation scheduling as a reasonable policy with prior knowledge of processor and link capacities missed. Meanwhile, they tried to prove the influence on the execution time comparing to the case with optimal scheduling policy.

In recent studies, more attentions are payed on translating divisible scheduling policy to distributed environments such as grid. Othman and Abdullah[90] designed to achieve an estimated performance level for larger jobs which was common in grid applications. They built an adaptive model which took both computation time and communication time into consideration to estimate the optimal distribution. They later proposed an enhanced model in [91]. The new model aimed to distribute loads over all grid sites to an optimal completion time for large scale jobs.

Furthermore, lots of data grid applications can be partitioned into multiple independent tasks for parallel execution and analysis which can be successfully exploited for scheduling divisible load on large scale data grids by using genetic algorithms. Abdullah and Othman[1] proposed an adaptive genetic algorithm to improve the representation of the data partition and the initial population to reduce the total execution time. They also designed a load distribution model in [2]. The proposed model took both the communication time and the computation time into consideration to minimize the total processing time cost by an optimal estimation of the completion time and the optimal distribution of the tasks among available processors in the grid.

Yu and Marinescu[127] introduced divisible load scheduling algorithms for data intensive applications. They identified divisible load scheduling to partition the input data and give out optimal mappings to collection of autonomous and heterogeneous systems.

However, there are very few work on optimization with DLT in GPU architecture.

5.2.2 GPU Performance Optimization

There are many studies on compiler techniques to optimize GPU memory references such as GPU optimizing compilers[55, 124, 125], polyhedral models[9], and performance tuning[98]. These works focus on static irregularities that are amenable for compiler analysis.

There are also some other studies on exploring the synergistic usage of CPU and GPU such as the execution strategies proposed by Huo[58] and the exploitation of OpenCL[66, 67]. Thread divergence (the threads in a warp follow different paths of a kernel) is a type of dynamic irregularity in GPU as well. Some hardware extensions are provided to remove thread divergences from kernel execution[44]. Carrillo and Siegel propose loop splitting and branch splitting to release register pressure[25].

Motokubota and Ino[86, 59] propose a parallelization scheme for parameter sweep (PS) applications with CUDA. They focus on PS applications with irregular access patterns. They try to exploit the similarity of data access between different parameters to resolve this irregularity to improve the performance.

Meng and Tarjan[83, 84] introduce an optimization technique named dynamic warp subdivision which enables threads to interleave the computations of paths along different branches to hide memory latency. Zheng and Jiang[128] propose runtime optimizations that can eliminate thread divergence with a CPU-GPU pipelining scheme and later enhance the performance of their work by eliminating dynamic irregularities in memory references and control flows [129, 130, 131, 132]. Che and Sheaffer [27] propose a simple application program interface that optimizes memory efficiency based on some hints about memory access patterns. Their work improve the performance by improve the performance of GPU local memory access while we focus on the communication latency between CPU and GPU.

Lee and Lo [72] investigate data streaming and data compression to reduce the communication cost and demonstrate the effectiveness of the two techniques via two case studies on GPU. Lee and Sung [73] propose a GPU thread-block scheduling method that can better utilize L2 cache and reduce the DRAM memory access to improve the performance. Wu and Zhao [118] develop two new data reorganization algorithms to overcome the limitation of previous methods to reduce non-coalesced memory access.

Some works implement automatic data management and communication optimization systems for GPUs. Jablinet al. develop a fully automatic CPU-GPU communication management system named CGCM[60]. DyManD[61] and GMAC[46] try to manage communication between the CPU and GPU automatically by using distributed shared memory techniques[3]. These works try to optimize the communication by automatic data management or data caching while we try to overlap the communication and computation to improve the performance.

5.2.3 Stream Overlapping

There are also some works on improving the performance of GPGPU applications with multiple streams on GPU architecture. Phillips[93] and Rodrigues[97] propose a stream programming model to overlap streams to implement and optimize their own specific applications rather than general applications. Adnan[92] proposes a pipelined parallel LZSS compression algorithm for GPUs to overlap the CPU code and GPU code in CPU-GPU architecture.

Hou and Zhou[56, 57] propose a programming language name bulk-synchronous GPU programming for stream scheduling on CUDA compatible GPUs. With their compiler, programmers can translate the sequential C language programs into kernels with host code. This frees programmers form the tedious work of the temporary stream management. Nakagawa and Ino[87] developed a compiler which enables an out of order execution for a batch of applications to increase the effects of multiple stream scheduling. Both of them work on application level while we focus on subtask level.

Suda and Aoki[109] also try to improve the performance of individual application by partitioning the application into subtasks and overlapping them to hide the communication latency. However, their work only support unidirectional data transmission and kernel execution overlapping in single GPU architecture.

The main difference between our TPSM and other work is that our TPSM can support overlapping of data sending, data receiving and kernel execution for individual application in GPU while some other works only support uni-directional data transfer or only for overlapping between applications. Besides overlapping, our work also aware how to use efficiently bandwidth between CPU and GPU.

Chapter 6

Conclusions and Future Work

Contents

6.1 Conclusions

This thesis proposed two performance optimization methods based on performance analytical modeling and communication latency hiding in GPU architecture: a performance analytical model and a task partitioning and scheduling method.

First of all, I proposed a performance analytical model for GPU architecture with instruction-level and memory-level awareness which can predict the kernel execution time cost of CUDA codes without running on GPUs. I used open source framework Ocelot to generate and analysis PTX codes from CUDA codes. I wrote a set of micro benchmark to test the time cost of each PTX instruction. With the time cost of each PTX instruction as input, I built a MPD submodel to dynamically calculate the maximum number of warps for concurrent memory access. Besides MPD submodel, I also proposed a CPD submodel to present the parallel execution of computation instructions between warps and within warps. With the two submodels, the performance model can predict the total time cost of the kernel execution. The evaluation showed that the performance model can achieve average 89.99% accuracy prediction with four benchmarks on four different type GPUs. I believe that the performance model can help programmers better understand the performance of their application on GPU and improve their application.

Secondly, we proposed a Task Partitioning and Scheduling Method(TPSM) which can partition GPU application into subtasks and overlap data send, kernel execution and data receive part of these subtasks. I classified GPU applications into six basic types based on the computation-to-communication ratio aspect and send-to-receive ratio aspect. Based on the classification, I proposed six TPSM submethod for each application type. To effectively utilize the bandwidth between the host and the device, I also provided a time optimal data transfer algorithm which can give a time optimal data transfer plan in multiple GPU architecture. TPSM can almost maximize the overlapping to improve the performance of individual application in single GPU architecture, multiple GPU symmetric architecture and multiple GPU non-symmetric architecture. Experiment results from four benchmarks on four type GPUs showed that TPSM can overlap the bidirectional data transfer with the kernel execution. Because the available performance improvement depends on the proportions of the three parts in the application. Therefore, TPSM is more suitable for the applications that there are significant communication latency between the host and the device.

6.2 Future Work

For performance model, there are two problems left. One is the cache simulation and another is the CPD model. As now most GPUs have cache inside, the prediction of our model for memory access intensive application can be greatly affected by the cache hits. With cache simulation in the future, we can have better prediction in all kind GPUs. For the CPD model, we find that there are parallel execution for computing instruction between warps and within warps from our black box test. However, we can hardly set an accurate model to simulate the parallel execution for computing instruction without GPU architecture details which is considered as commercial secrets. We are looking forward to the cooperation with manufactures in the future.

For task partitioning and scheduling method, we have already supported all kind of architectures. There are two potential work can be done in the future. First of all, we plan to focus on reduce the overhead from modeling and scheduling in TPSM. As we need to model and calculate the optimal scheduling plan, there are some computation on CPU which decreases the total performance a little. The computation is necessary while we would like to minimize the overhead as much as possible. Secondly, TPSM currently is an optimization method for programmers to hide the communication latency between GPU and CPU. Programmers need to implement some codes to use this method. In the future, we are considering to develop a software to automatically partition and schedule with input applications. Another idea is to combine TPSM into compiler to achieve automatically partition and schedule. In this way, we can release programmers from the burden of the implementation.

As the development of GPU devices is rapid, the architecture of future generation GPU devices can be very different from current. The bandwidth between the GPU and CPU may be increased a lot. However, notice that the applications become more complex and require more communication with GPU while the computing capacity of GPU and the bandwidth between the host and the device are greatly improved. For example, we might use GPU devices to accelerate big data applications in the future. Therefore, the bandwidth will still be an important performance bottleneck in the future generation GPU devices. Then our work can also be used in the future generation GPU architecture to hide the communication latency with very little modification.

Appendix A

Appendix

Table A.1: Time cost of PTX instructions in GTX C2050 (unit: GPU clock)

	int_const	int_reg	float_const	float_reg
add	22	64	22	64
sub	44	133	22	67
mul	44	136	22	65
div	225	941	811	767
neg	24	24	24	24
min	65	65	64	64
max	65	65	64	64
and	62	62	62	62
or	62	62	62	62
xor	65	65	65	65
not	24	22	24	22
shl	24	62	24	62
shr	24	62	24	62
mv	40	40	40	40
cvt	24	24	24	24
ld/st	70	70	70	70

Table A.2: Bandwidth of Dual Tesla C2070 under Different States

	G0-N	G0-S	G0-R	G0-S&R
G1-N	-	$S0 = 5.2$	$R0 = 4.5$	$S0 = 3.03$ $R0 = 3.16$
G1-S	$S1 = 5.2$	$S0 = 3.5$ $S1 = 3.5$	$R0 = 3.15$ $S1 = 3.08$	$S0 = 1.97$ $R0 = 2.36$ $S1 = 2.85$
G1-R	$R1 = 4.5$	$S0 = 3.08$ $R1 = 3.15$	$R0 = 2.28$ $R1 = 2.28$	$S0 = 1.69$ $R0 = 1.72$ $R1 = 2.03$
G1-S&R	$S1 = 3.03$ $R1 = 3.16$	$S0 = 2.85$ $S1 = 1.97$ $R1 = 2.36$	$R0 = 2.03$ $S1 = 1.69$ $R1 = 1.72$	$S0 = 1.56$ $R0 = 1.59$ $S1 = 1.56$ $R1 = 1.59$

Table A.3: Bandwidth of Dual Tesla M2090 under Different States

	G0-N	G0-S	G0-R	G0-S&R
G1-N	-	$S0 = 6.12$	$R0 = 5.8$	$S0 = 5.13$ $R0 = 5.06$
G1-S	$S1 = 6.12$	$S0 = 6.09$ $S1 = 6.09$	$R0 = 5.75$ $S1 = 5.92$	$S0 = 4.44$ $R0 = 4.33$ $S1 = 5.84$
G1-R	$R1 = 5.8$	$S0 = 5.92$ $R1 = 5.75$	$R0 = 5.71$ $R1 = 5.71$	$S0 = 4.42$ $R0 = 4.31$ $R1 = 5.68$
G1-S&R	$S1 = 5.13$ $R1 = 5.06$	$S0 = 5.84$ $S1 = 4.44$ $R1 = 4.33$	$R0 = 5.68$ $S1 = 4.42$ $R1 = 4.31$	$S0 = 4.12$ $R0 = 4.08$ $S1 = 4.12$ $R1 = 4.08$

Table A.4: Bandwidth of Four Tesla M2090 under Different States

Status	Bandwidth(GB/s)	Send	Receive
S/N/N/N	S0=6.14	6.14	0
R/N/N/N	R0=4.98	0	4.98
SR/N/N/N	S0=4.55, R0=3.86	4.55	3.86
S/S/N/N	S0=S1=6.11	12.22	0
S/R/N/N	S0=6.12, R1=4.96	6.12	4.96
R/R/N/N	R0=R1=4.84	0	9.68
SR/S/N/N	S0=4.61, R0=4.17, S1=6.05	10.66	4.17
SR/R/N/N	S0=4.39, R0=3.97, R1=4.8	4.39	8.77
SR/SR/N/N	S0=S1=4.4, R0=R1=3.83	8.8	7.66
S/S/S/N	S0=S1=S2=6.1	18.3	0
S/S/R/N	S0=S1=6.09, R2=5.22	12.18	5.22
S/R/R/N	S0=6.12, R1=R2=4.74	6.12	9.48
R/R/R/N	R0=R1=R2=4.77	0	14.31
SR/S/S/N	S0=4.66, R0=4.12, S1=S2=6.06	16.78	4.12
SR/S/R/N	S0=4.6, R0=4.19, S1=6.11, R2=4.86	0.71	9.05
SR/R/R/N	S0=4.56, R0=4.17, R1=R2=4.34	4.56	12.85
SR/SR/S/N	S0=S1=3.98, RO=R1=4.14, S2=6.1	14.06	8.28
SR/SR/R/N	S0=S1=4.42, R0=R1=4.2, R2=4.12	8.84	12.52
SR/SR/SR/N	SO=S1=S2=4.49, R0=R1=R2=3.89	13.47	11.67
S/S/S/S	S0=S1=S2=S3=6.08	24.32	0
S/S/S/R	S0=S1=S2=6.07, R3=4.76	18.21	4.76
S/S/R/R	S0=S1=6.04, R2=R3=4.85	12.08	9.7
S/R/R/R	S0=6.08, R1=R2=R3=4.54	6.08	13.62
R/R/R/R	R0=R1=R2=R3=4.42	0	17.68
SR/S/S/S	S0=4.51, RO=3.97, S1=S2=S3=5.99	16.49	3.97
SR/S/S/R	S0=4.15, R0=3.96, S1=S2=6.08, R3=4.39	16.31	8.35
SR/S/R/R	S0=4.34, R0=3.95, S1=6.1, R2=R3=4.31	10.44	12.57
SR/R/R/R	S0=4.34, R0=4.15, R1=R2=R3=4.39	4.34	17.23
SR/SR/S/S	S0=S1=4.08, R0=R1=4.03, S2=S3=6.03	20.22	8.06
SR/SR/S/R	S0=S1=4.15, R0=R1=4.28, S2=6.09, R3=4.85	14.39	13.41
SR/SR/R/R	S0=S1=4.18, R0=R1=4.08, R2=R3=4.22	8.36	16.6
SR/SR/SR/S	S0=S1=S2=4.04, R0=R1=R2=3.58, S3=5.79	17.91	10.74
SR/SR/SR/R	S0=S1=S2=3.92, R0=R1=R2=3.83, R3=4.03	11.76	15.52
SR/SR/SR/SR	S0=S1=S2=S3=3.9, R0=R1=R2=R3=3.81	15.6	14.24

Bibliography

[1] Monir Abdullah, Mohamed Othman, Hamidah Ibrahim and Shamala Subramaniam. *An Integrated Approach for Scheduling Divisible Load on Large Scale Data Grids.* In International Conference on Computational Science and Its Applications, ICCSA'07, Kuala Lumpur, Malaysia, August 26-29, 2007, volume 4705 of *Lecture Notes in Computer Science*, pages 748–757. Springer, 2007. 113

[2] Monir Abdullah, Mohamed Othman, Hamidah Ibrahim and Shamala Subramaniam. *Optimal workload allocation model for scheduling divisible data grid applications.* Future Generation Comp. Syst., vol. 26, no. 7, pages 971–978, 2010. 113

[3] Cristiana Amza, Alan L. Cox, Hya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu and Willy Zwaenepoel. *TreadMarks: Shared Memory Computing on Networks of Workstations.* IEEE Computer, vol. 29, pages 18–28, 1996. 115

[4] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp and Wen-mei W. Hwu. *An adaptive performance modeling tool for GPU architectures.* In Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10, pages 105–114, New York, NY, USA, 2010. ACM. 109

[5] Kenneth R. Baker. Introduction to sequencing and scheduling. J. Wiley and sons, New York, 1974. 11

[6] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong and Tor M. Aamodt. *Analyzing CUDA workloads using a detailed GPU simulator.* In IEEE International Symposium on Performance Analysis of Systems and Software,ISPASS2009, pages 163–174. IEEE, April 26-28, 2009, Boston, Massachusetts, USA. 109, 110, 111

[7] Evripidis Bampis, Jean-Claude König and Denis Trystram. *Optimal Parallel Execution of Complete Binary Trees and Grids into most Popular Interconnection Networks.* In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou and Sergios Theodoridis, editeurs, PARLE 94: Parallel Architectures and Languages Europe, 6th International PARLE Conference,

Athens, Greece, July 4-8, 1994, Proceedings, volume 817 of *Lecture Notes in Computer Science*, pages 122–133. Springer, 1994. 11

[8] Gerassimos D. Barlas. *Collection-Aware Optimum Sequencing of Operations and Closed-Form Solutions for the Distribution of a Divisible Load on Arbitrary Processor Trees.* IEEE Trans. Parallel Distrib. Syst., vol. 9, no. 5, pages 429–441, May 1998. 12

[9] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev and P. Sadayappan. *A compiler framework for optimization of affine loop nests for gpgpus.* In Proceedings of the 22nd annual international conference on Supercomputing, ICS '08, pages 225–234, New York, NY, USA, 2008. ACM. 114

[10] S. Bataineh and B. Al-Asir. *An efficient scheduling algorithm for divisible and indivisible tasks in loosely coupled multiprocessor systems.* Software Engineering Journal, vol. 9, no. 1, January 1994. 12

[11] S. Bataineh, Te-Yu Hsiung and T. G. Robertazzi. *Closed Form Solutions for Bus and Tree Networks of Processors Load Sharing a Divisible Job.* IEEE Trans. Comput., vol. 43, no. 10, pages 1184–1196, October 1994. 12

[12] Jacek Bazewicz and Et Al. Scheduling computer and manufacturing processes. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd édition, 2001. 11

[13] Olivier Beaumont, Arnaud Legrand and Yves Robert. *Scheduling divisible workloads on heterogeneous platforms.* Parallel Computing, vol. 29, no. 9, pages 1121–1152, 2003. 113

[14] Olivier Beaumont, Henri Casanova, Arnaud Legrand, Yves Robert and Yang Yang. *Scheduling Divisible Loads on Star and Tree Networks: Results and Open Problems.* IEEE Trans. Parallel Distrib. Syst., vol. 16, no. 3, pages 207–218, 2005. 113

[15] V. Bharadwaj, D. Ghose and V. Mani. *Optimal Sequencing and Arrangement in Distributed Single-Level Tree Networks with Communication Delays.* IEEE Trans. Parallel Distrib. Syst., vol. 5, no. 9, pages 968–976, September 1994. 12

[16] V. Bharadwaj, D. Ghose and V. Mani. *Multi-installment load distribution in tree networks with delays.* IEEE Transactions on Aerospace and Electronic Systems, vol. 31, no. 2, pages 555–567, April 1995. 12

[17] Veeravalli Bharadwaj, Thomas G. Robertazzi and Debasish Ghose. *Scheduling Divisible Loads in Parallel and Distributed Systems.* In IEEE Computer Society Press, Los Alamitos, CA, USA, 1996. 11

[18] V. Bharadwaj, Xiaolin Li 0001 and Chi Chung Ko. *Efficient partitioning and scheduling of computer vision and image processing data on bus networks using divisible load analysis.* Image Vision Comput., vol. 18, no. 11, pages 919–938, 2000. 12

[19] Veeravalli Bharadwaj, Debasish Ghose and Thomas G. Robertazzi. *Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems.* Cluster Computing, vol. 6, no. 1, pages 7–17, January 2003. 12

[20] Jacek Blazewicz and Maciej Drozdowski. *Distributed processing of divisible jobs with communication startup costs.* In Proceedings of the second international colloquium on Graphs and optimization, GO-II Meeting, pages 21–41, Amsterdam, The Netherlands, The Netherlands, 1997. Elsevier Science Publishers B. V. 11

[21] Jacek Blazewicz and Maciej Drozdowski. *Distributed processing of divisible jobs with communication startup costs.* Discrete Appl. Math., vol. 76, no. 1-3, pages 21–41, June 1997. 11

[22] Shekhar Borkar and Andrew A. Chien. *The future of microprocessors.* Commun. ACM, vol. 54, no. 5, pages 67–77, May 2011. 1

[23] Janez Brest and Viljem Žumer. *A Simple Method for Dynamic Scheduling in a Heterogeneous Computing System.* Journal of Computing and Information Technology, vol. 10, no. 2, pages 1330–1136, 2002. 112

[24] *Brook+ sc07 bof session.* `http://developer.amd.com/wordpress/media/2012/10/AMD-Brookplus.pdf`, 2007. [Online; accessed 24-Dec-2012]. 3

[25] Snaider Carrillo, Jakob Siegel and Xiaoming Li. *A control-structure splitting optimization for GPGPU.* In Proceedings of the 6th ACM conference on Computing frontiers, CF '09, pages 147–150, New York, NY, USA, 2009. ACM. 114

[26] Bernard Chazelle. *An Optimal Convex Hull Algorithm and New Results on Cuttings (Extended Abstract).* In FOCS, pages 29–38. IEEE Computer Society, 1991. 79

[27] Shuai Che, Jeremy W. Sheaffer and Kevin Skadron. *Dymaxion: optimizing memory access patterns for heterogeneous systems.* In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pages 13:1–13:11, New York, NY, USA, 2011. ACM. 114

[28] Y.C. Cheng and T.G. Robertazzi. *Distributed computation with communication delay (distributed intelligent sensor networks).* IEEE Transaction on Aerospace and Elecronic Systems, vol. 24, no. 6, pages 700–712, 1988. 11, 112

[29] Y.-C. Cheng and T. G. Robertazzi. *Distributed computation for a tree network with communication delays.* IEEE Trans. Aerosp. Electron. Syst., vol. 26, no. 3, May 1990. 12

[30] Suriayati Chuprat and Sanjoy K. Baruah. *Scheduling Divisible Real-Time Loads on Clusters with Varying Processor Start Times.* In The Fourteenth IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2008, Kaohisung, Taiwan, 25-27 August 2008, pages 15–24. IEEE Computer Society, 2008. 112

[31] M. J. Clement and M. J. Quinn. *Analytical performance prediction on multicomputers.* In Proceedings of the 1993 ACM/IEEE conference on Supercomputing, Supercomputing '93, pages 886–894, New York, NY, USA, 1993. ACM. 109

[32] Edward G. Coffman Jr. and Peter J. Denning. Operating systems theory. Prentice Hall Professional Technical Reference, 1973. 11

[33] Edward Grady Coffman and John L. Bruno, editeurs. Computer and job-shop scheduling theory. Wiley, New York, 1976. A Wiley-Interscience publication. 11

[34] Sylvain Collange, Marc Daumas, David Defour and David Parello. *Barra: A Parallel Functional Simulator for GPGPU.* In 18th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2010, pages 351–360. IEEE, Miami, Florida, USA, August 17-19, 2010. 110

[35] Nicholas Comino and V. Lakshmi Narasimhan. *A Novel Data Distribution Technique for Host-Client Type Parallel Applications.* IEEE Trans. Parallel Distrib. Syst., vol. 13, no. 2, pages 97–110, February 2002. 12

[36] NVIDIA Compute. *PTX: Parallel Thread Execution ISA Version 2.3.* 14, 17

[37] *CUDA Programming Guide.* `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, 2012. [Online; accessed 24-Dec-2012]. 3, 7

[38] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian and Thorsten von Eicken. *LogP: towards a realistic model of parallel computation.* In Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM. 109

[39] Gregory Frederick Diamos, Andrew Kerr, Sudhakar Yalamanchili and Nathan Clark. *Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems.* In 19th International Conference on Parallel Architecture and Compilation Techniques,PACT2010, pages 353–364. ACM, Vienna, Austria, September 11-15, 2010. 10

[40] *Divisible (partitionable) load scheduling research.* `http://www.ece.sunysb.edu/~tom/dlt.html`, 2012. [Online; accessed 24-Dec-2012]. 12

[41] Maciej Drozdowski and Pawel Wolniewicz. *Experiments with Scheduling Divisible Tasks in Clusters of Workstations.* In International Euro-Par Conference on Parallel Processing, Euro-Par'00, volume 1900 of *Lecture Notes in Computer Science*, pages 311–319. Springer, 2000. 111

[42] Maciej Drozdowski and Pawel Wolniewicz. *Divisible Load Scheduling in Systems with Limited Memory.* Cluster Computing, vol. 6, no. 1, pages 19–29, 2003. 112

[43] Maciej Drozdowski and Pawel Wolniewicz. *Optimum divisible load scheduling on heterogeneous stars with limited memory.* European Journal of Operational Research, vol. 172, no. 2, pages 545–559, 2006. 112

[44] Wilson W. L. Fung, Ivan Sham, George Yuan and Tor M. Aamodt. *Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow.* In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society. 114

[45] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang and Vasily Volkov.

Parallel Computing Experiences with CUDA. IEEE Micro, vol. 28, no. 4, pages 13–27, 2008. 2

[46] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro and Wen-mei W. Hwu. *An asymmetric distributed shared memory model for heterogeneous parallel systems.* In Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems, ASPLOS XV, pages 347–358, New York, NY, USA, 2010. ACM. 115

[47] D. Ghose and V. Mani. *Distributed computation with communication delays: asymptotic performance analysis.* J. Parallel Distrib. Comput., vol. 23, no. 3, pages 293–305, December 1994. 12

[48] Debasish Ghose and Hyoung Joong Kim. *Load partitioning and trade-off study for large matrix-vector computations in multicast bus networks with communication delays.* J. Parallel Distrib. Comput., vol. 55, no. 1, pages 32–59, November 1998. 12

[49] Phillip B. Gibbons, Yossi Matias and Vijaya Ramachandran. *The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms.* SIAM J. Comput., vol. 28, no. 2, pages 733–769, February 1999. 109

[50] E. Grochowski and M. Annavaram. *Energy per Instruction Trends in Intel Microprocessors.* Technology @ Intel Magazine, Mar.2006. 1

[51] John L. Hennessy and David A. Patterson. Computer architecture, fourth edition: A quantitative approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. 2

[52] G.Jee H.J.Kim and J.G.Lee. *Optimal load distribution for tree network processors.* IEEE Trans. Aerosp. Electron. Syst., vol. 32, no. 2, April 1996. 12

[53] Sunpyo Hong and Hyesoon Kim. *An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness.* SIGARCH Comput. Archit. News, vol. 37, no. 3, pages 152–163, June 2009. 109, 110

[54] 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. 109

[55] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge and Scott Mahlke. *Sponge: portable stream programming on graphics engines.* In Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, ASPLOS XVI, pages 381–392, New York, NY, USA, 2011. ACM. 114

[56] Qiming Hou, Kun Zhou and Baining Guo. *BSGP: bulk-synchronous GPU programming.* In ACM SIGGRAPH 2008 papers, SIGGRAPH '08, pages 19:1–19:12, New York, NY, USA, 2008. ACM. 115

[57] Qiming Hou, Kun Zhou and Baining Guo. *BSGP: bulk-synchronous GPU programming.* ACM Trans. Graph., vol. 27, no. 3, pages 19:1–19:12, August 2008. 115

[58] Xin Huo, Vignesh Ravi, Wenjing Ma and Gagan Agrawal. *An execution strategy and optimized runtime support for parallelizing irregular reductions on modern GPUs.* In Proceedings of the international conference on Supercomputing, ICS '11, pages 2–11, New York, NY, USA, 2011. ACM. 114

[59] Fumihiko Ino, Kentaro Shigeoka, Tomohiro Okuyama, Masaya Motokubota and Kenichi Hagihara. *A parallel scheme for accelerating parameter sweep applications on a GPU.* Concurrency and Computation: Practice and Experience, pages n/a–n/a, 2013. 114

[60] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard and David I. August. *Automatic CPU-GPU communication management and optimization.* In Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI '11, pages 142–151, New York, NY, USA, 2011. ACM. 115

[61] Thomas B. Jablin, James A. Jablin, Prakash Prabhu, Feng Liu and David I. August. *Dynamically managed data for CPU-GPU architectures.* In Proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO '12, pages 165–174, New York, NY, USA, 2012. ACM. 115

[62] Haipeng Jia, Yunquan Zhang, Guoping Long, Jianliang Xu, Shengen Yan and Yan Li. *GPURoofline: a model for guiding performance optimizations on GPUs.* In Proceedings of the 18th international conference on Parallel Processing, Euro-Par'12, pages 920–932, Berlin, Heidelberg, 2012. Springer-Verlag. 110

[63] Andrew Kerr, Gregory Diamos and Sudhakar Yalamanchili. *A characterization and analysis of PTX kernels.* In Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), IISWC '09, pages 3–12, Washington, DC, USA, 2009. IEEE Computer Society. 110, 111

[64] Andrew Kerr, Gregory Diamos and Sudhakar Yalamanchili. *Modeling GPU-CPU workloads and systems.* In Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU '10, pages 31–42, New York, NY, USA, 2010. ACM. 110

[65] Hyoung Joong Kim. *A Novel Optimal Load Distribution Algorithm for Divisible Loads.* Cluster Computing, vol. 6, no. 1, pages 41–46, 2003. 112

[66] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo and Jaejin Lee. *OpenCL as a unified programming model for heterogeneous CPU/GPU clusters.* SIGPLAN Not., vol. 47, no. 8, pages 299–300, February 2012. 114

[67] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo and Jaejin Lee. *OpenCL as a unified programming model for heterogeneous CPU/GPU clusters.* In Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPoPP '12, pages 299–300, New York, NY, USA, 2012. ACM. 114

[68] Kwangil Ko and Thomas G. Robertazzi. *Equal allocation scheduling for data intensive applications.* IEEE Transactions on Aerospace and Electronic Systems, vol. 40, no. 2, pages 695–705, April. 2004. 113

[69] Kishore Kothapalli, Rishabh Mukherjee, M. Suhail Rehman, Suryakant Patidar, P. J. Narayanan and Kannan Srinathan. *A performance prediction model for the CUDA GPGPU platform.* In 16th International Conference on High Performance Computing, HiPC 2009, pages 463–472. IEEE, December 16-19, 2009, Kochi, India. 109, 110

[70] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan and Dean M. Tullsen. *Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction.* In Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society. 1

[71] Cheol-Hoon Lee and Kang G. Shin. *Optimal Task Assignment in Homogeneous Networks.* IEEE Trans. Parallel Distrib. Syst., vol. 8, no. 2, pages 119–129, February 1997. 12

[72] Che-Rung Lee, Shih-Hsiang Lo, Nan-Hsi Chen, Yeh-Ching Chung and I-Hsin Chung. *GPU Performance Enhancement via Communication Cost Reduction: Case Studies of Radix Sort and WSN Relay Node Placement Problem.* In Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012), CCGRID '12, pages 132–139, Washington, DC, USA, 2012. IEEE Computer Society. 114

[73] Seungyeol Lee and Wonyong Sung. *DRAM Access Reduction in GPUs by Thread-Block Scheduling.* In 2013 IEEE International Symposium on Circuits and Systems, ISCAS'13, pages 901–904, Beijing, China, 2013. IEEE Computer Society. 114

[74] *Real-Time Divisible Load Scheduling for Cluster Computing.* In IEEE Real-Time and Embedded Technology and Applications Symposium, pages 303–314. IEEE Computer Society, 2007. 112

[75] *Real-Time Divisible Load Scheduling with Different Processor Available Times.* In International Conference on Parallel Processing, ICPP'07, page 20. IEEE Computer Society, 2007. 112

[76] Xuan Lin, Ying Lu, Jitender S. Deogun and Steve Goddard. *Enhanced Real-Time Divisible Load Scheduling with Different Processor Available Times.* In High Performance Computing, HiPC'07, volume 4873 of *Lecture Notes in Computer Science*, pages 308–319. Springer, 2007. 112

[77] Xuan Lin, Anwar Mamat, Ying Lu, Jitender S. Deogun and Steve Goddard. *Real-time scheduling of divisible loads in cluster computing environments.* J. Parallel Distrib. Comput., vol. 70, no. 3, pages 296–308, 2010. 112

[78] Michael D. Linderman, Jamison D. Collins, Hong Wang and Teresa H. Y. Meng. *Merge: a programming model for heterogeneous multi-core systems.* In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008, pages 287–296. ACM, 2008. 22, 82, 83

[79] Lin Ma, Kunal Agrawal and Roger D. Chamberlain. *A Memory Access Model for Highly-threaded Many-core Architectures.* In IEEE 18th International Conference on Parallel and Distributed Systems, ICPADS 2012, pages 339–347. IEEE Computer Society, Singapore, December 17-19, 2012. 109

[80] Lin Ma and Roger D. Chamberlain. *A Performance Model for Memory Bandwidth Constrained Applications on Graphics Engines.* In 23rd IEEE

International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2012, pages 24–31. IEEE Computer Society, Delft, The Netherlands, July 9-11, 2012. 110

[81] V. Mani and D. Ghose. *Distributed computation in linear networks: Closed-form solutions.* IEEE Trans. Aerosp. Electron. Syst., vol. 30, no. 2, April 1994. 12

[82] Gabriel Marin and John Mellor-Crummey. *Cross-architecture performance predictions for scientific applications using parameterized models.* In Proceedings of the joint international conference on Measurement and modeling of computer systems, SIGMETRICS '04/Performance '04, pages 2–13, New York, NY, USA, 2004. ACM. 109

[83] Jiayuan Meng, David Tarjan and Kevin Skadron. *Dynamic warp subdivision for integrated branch and memory divergence tolerance.* In Proceedings of the 37th annual international symposium on Computer architecture, ISCA '10, pages 235–246, New York, NY, USA, 2010. ACM. 114

[84] Jiayuan Meng, David Tarjan and Kevin Skadron. *Dynamic warp subdivision for integrated branch and memory divergence tolerance.* SIGARCH Comput. Archit. News, vol. 38, no. 3, pages 235–246, June 2010. 114

[85] Gordon E. Moore. *Readings in computer architecture.* chapitre Cramming more components onto integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. 1

[86] Masaya Motokubota, Fumihiko Ino and Kenichi Hagihara. *Accelerating Parameter Sweep Applications Using CUDA.* In Yiannis Cotronis, Marco Danelutto and George Angelos Papadopoulos, editeurs, PDP, pages 111–118. IEEE Computer Society, 2011. 114

[87] Shinta Nakagawa, Fumihiko Ino and Kenichi Hagihara. *A middleware for efficient stream processing in CUDA.* Computer Science - R&D, vol. 25, no. 1-2, pages 41–49, 2010. 115

[88] *Ocelot:.* `https://code.google.com/p/gpuocelot/`, 2012. [Online; accessed 24-Dec-2012]. 5, 10, 17

[89] *OpenCL - The open standard for parallel programming of heterogeneous systems.* `http://www.khronos.org/opencl/`, 2012. [Online; accessed 24-Dec-2012]. 3

[90] Mohamed Othman, Monir Abdullah, Hamidah Ibrahim and Shamala Subramaniam. *Adaptive Divisible Load Model for Scheduling Data-Intensive Grid Applications.* In International Conference on Computational Science, ICCS'07, Beijing, China, May 27-30, 2007, volume 4487 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2007. 113

[91] Mohamed Othman, Monir Abdullah, Hamidah Ibrahim and Shamala Subramaniam. *A2DLT: Divisible Load Balancing Model for Scheduling Communication-Intensive Grid Applications.* In International Conference on Computational Science, ICCS'08, Kraków, Poland, June 23-25, 2008, volume 5101 of *Lecture Notes in Computer Science*, pages 246–253. Springer, 2008. 113

[92] Adnan Ozsoy, D. Martin Swany and Arun Chauhan. *Pipelined Parallel LZSS for Streaming Data Compression on GPGPUs.* In IEEE 18th International Conference on Parallel and Distributed Systems, ICPADS 2012, pages 37–44. IEEE Computer Society, Singapore, December 17-19, 2012. 115

[93] James C. Phillips, John E. Stone and Klaus Schulten. *Adapting a message-driven parallel application to GPU-accelerated clusters.* In Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08, pages 8:1–8:9, Piscataway, NJ, USA, 2008. IEEE Press. 115

[94] G. N. Srinivasa Prasanna and Bruce R. Musicus. *Generalized multiprocessor scheduling for directed acylic graphs.* In Proceedings of the 1994 ACM/IEEE conference on Supercomputing, Supercomputing '94, pages 237–246, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. 11

[95] Thomas G. Robertazzi. *Ten Reasons to Use Divisible Load Theory.* Computer, vol. 36, no. 5, pages 63–68, May 2003. 12

[96] T.G. Robertazzi. *Networks and Grids: Technology and Theory.* Springer Publishing Company, Inc., 1st edition, 2007. 12

[97] Christopher I. Rodrigues, David J. Hardy, John E. Stone, Klaus Schulten and Wen-Mei W. Hwu. *GPU acceleration of cutoff pair potentials for molecular modeling applications.* In Proceedings of the 5th conference on Computing frontiers, CF '08, pages 273–282, New York, NY, USA, 2008. ACM. 115

[98] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk and Wen-mei W. Hwu. *Optimization principles and application performance evaluation of a multithreaded GPU using CUDA.* In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice

of parallel programming, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM. 114

[99] Karsten Schwan and Hongyi Zhou. *Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads*. IEEE Trans. Softw. Eng., vol. 18, no. 8, pages 736–748, August 1992. 11

[100] R Seidel. *Constructing higher-dimensional convex hulls at logarithmic cost per face*. In Proceedings of the eighteenth annual ACM symposium on Theory of computing, STOC '86, pages 404–413, New York, NY, USA, 1986. ACM. 79

[101] Terry Shepard and J. A. Martin Gagné. *A Pre-Run-Time Scheduling Algorithm for Hard Real-Time Systems*. IEEE Trans. Softw. Eng., vol. 17, no. 7, pages 669–677, July 1991. 11

[102] Amin Shokripour and Mohamed Othman. *Survey on divisible load theory and its applications*. In International Conference on Information Management and Engineering, ICIME'09, pages 300–304, 2009. 11, 111

[103] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim and Richard Vuduc. *A performance analysis framework for identifying potential benefits in GPGPU applications*. SIGPLAN Not., vol. 47, no. 8, pages 11–22, February 2012. 110

[104] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim and Richard Vuduc. *A performance analysis framework for identifying potential benefits in GPGPU applications*. In Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPoPP '12, pages 11–22, New York, NY, USA, 2012. ACM. 110

[105] Alan Jay Smith. *Cache Memories*. ACM Comput. Surv., vol. 14, no. 3, pages 473–530, September 1982. 2

[106] J. Sohn and T. G. Robertazzi. *Optimal divisible job load sharing for bus networks*. IEEE Trans. Aerosp. Electron. Syst., vol. 32, no. 1, pages 34–40, June 1996. 12

[107] Jeeho Sohn, Thomas G. Robertazzi and Serge Luryi. *Optimizing Computing Costs Using Divisible Load Analysis*. IEEE Trans. Parallel Distrib. Syst., vol. 9, no. 3, pages 225–234, March 1998. 12

[108] John A. StankovicThis work was done while the, Marco Spuri, Marco Di Natale and Giorgio C. Buttazzo. *Implications of Classical Scheduling Results*

for Real-Time Systems. Computer, vol. 28, no. 6, pages 16–25, June 1995. 11

[109] Reiji Suda, Takayuki Aoki, Shoichi Hirasawa, Akira Nukada, Hiroki Honda and Satoshi Matsuoka. *Aspects of GPU for general purpose high performance computing.* In Proceedings of the 2009 Asia and South Pacific Design Automation Conference, ASP-DAC '09, pages 216–223, Piscataway, NJ, USA, 2009. IEEE Press. 115

[110] Andrew S Tanenbaum and Albert S Woodhull. Operating systems design and implementation (3rd edition). Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005. 11

[111] T.G.Robertazzi. *Processor equivalence for daisy chain load sharing processors.* IEEE Trans. Aerosp. Electron. Syst., vol. 29, no. 4, pages 1216–1221, October 1993. 12

[112] Bharadwaj Veeravalli, Xiaolin Li and Chi Chung Ko. *On the Influence of Start-Up Costs in Scheduling Divisible Loads on Bus Networks.* IEEE Trans. Parallel Distrib. Syst., vol. 11, no. 12, pages 1288–1305, December 2000. 12

[113] Bharadwaj Veeravalli and Surendra Ranganath. *Theoretical and experimental study on large size image processing applications using divisible load paradigm on distributed bus networks.* Image Vision Comput., vol. 20, no. 13-14, pages 917–935, 2002. 112

[114] Bharadwaj Veeravalli, Debasish Ghose and Thomas G. Robertazzi. *Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems.* Cluster Computing, vol. 6, no. 1, pages 7–17, 2003. 12

[115] David W. Wall. *Limits of instruction-level parallelism.* SIGARCH Comput. Archit. News, vol. 19, no. 2, pages 176–188, April 1991. 1

[116] David W. Wall. *Limits of instruction-level parallelism.* In Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, ASPLOS IV, pages 176–188, New York, NY, USA, 1991. ACM. 1

[117] R. Clint Whaley and David B. Whalley. *Tuning High Performance Kernels through Empirical Compilation.* In 34th International Conference on Parallel Processing (ICPP 2005), pages 89–98. IEEE Computer Society, 14-17 June 2005, Oslo, Norway. 109

[118] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang and Xipeng Shen. *Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU.* In Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '13, pages 57–68, New York, NY, USA, 2013. ACM. 114

[119] V.Bharadwaj X.Li and C.C.Ko. *Divisible load scheduling on single-level tree networks with buffer constraints.* IEEE Trans. Aerosp. Electron. Syst., vol. 36, no. 4, October 2000. 12

[120] J. Xu. *Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations.* IEEE Trans. Softw. Eng., vol. 19, no. 2, pages 139–154, February 1993. 12

[121] J. Xu and D. L. Parnas. *On Satisfying Timing Constraints in Hard-Real-Time Systems.* IEEE Trans. Softw. Eng., vol. 19, no. 1, pages 70–84, January 1993. 11

[122] Yang Yang and Henri Casanova. *RUMR: Robust Scheduling for Divisible Workloads.* In 12th International Symposium on High-Performance Distributed Computing (HPDC-12), 22-24 June 2003, Seattle, WA, USA, pages 114–125. IEEE Computer Society, 2003. 113

[123] Yang Yang and Henri Casanova. *UMR: A Multi-Round Algorithm for Scheduling Divisible Workloads.* In 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, page 24. IEEE Computer Society, 2003. 113

[124] Yi Yang, Ping Xiang, Jingfei Kong and Huiyang Zhou. *A GPGPU compiler for memory optimization and parallelism management.* In Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM. 114

[125] Yi Yang, Ping Xiang, Jingfei Kong and Huiyang Zhou. *A GPGPU compiler for memory optimization and parallelism management.* SIGPLAN Not., vol. 45, no. 6, pages 86–97, June 2010. 114

[126] Dantong Yu, , Dantong Yu and Thomas G. Robertazzi. *Divisible Load Scheduling for Grid Computing.* In in PDCS '2003, 15th Int 'l Conf. Parallel and Distributed Computing and Systems. IASTED. Press, 2003. 12

[127] Chen Yu and Dan C. Marinescu. *Algorithms for Divisible Load Scheduling of Data-intensive Applications.* J. Grid Comput., vol. 8, no. 1, pages 133–155, 2010. 113

[128] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo and Xipeng Shen. *Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping.* In Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10, pages 115–126, New York, NY, USA, 2010. ACM. 114

[129] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian and Xipeng Shen. *On-the-fly elimination of dynamic irregularities for GPU computing.* SIGPLAN Not., vol. 47, no. 4, pages 369–380, March 2011. 114

[130] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian and Xipeng Shen. *On-the-fly elimination of dynamic irregularities for GPU computing.* In Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, ASPLOS XVI, pages 369–380, New York, NY, USA, 2011. ACM. 114

[131] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian and Xipeng Shen. *On-the-fly elimination of dynamic irregularities for GPU computing.* SIGPLAN Not., vol. 46, no. 3, pages 369–380, March 2011. 114

[132] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian and Xipeng Shen. *On-the-fly elimination of dynamic irregularities for GPU computing.* SIGARCH Comput. Archit. News, vol. 39, no. 1, pages 369–380, March 2011. 114

[133] Yao Zhang and John D. Owens. *A quantitative performance analysis model for GPU architectures.* In Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture, HPCA '11, pages 382–393, Washington, DC, USA, 2011. IEEE Computer Society. 110