

**PARALLEL COMPUTING FRAMEWORK
AND GPU PERFORMANCE MODELING**

by

Wenjing Xu, B.S., M.S.

A Dissertation Presented in Partial Fulfillment
of the Requirements of the Degree
Doctor of Philosophy

COLLEGE OF ENGINEERING AND SCIENCE
LOUISIANA TECH UNIVERSITY

February 2022

LOUISIANA TECH UNIVERSITY

GRADUATE SCHOOL

April 7, 2020

Date of dissertation defense

We hereby recommend that the dissertation prepared by

Wenjing Xu, B.S., M.S.

entitled **Parallel Computing Framework and GPU Performance Modeling**

be accepted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computational Analysis & Modeling



Chokchai (Box) Leangsuksun
Supervisor of Dissertation Research

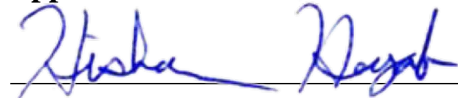


Weizhong Dai
Head of Computational Analysis & Modeling

Doctoral Committee Members:

Weizhong Dai
Pradeep Chowriappa
Songming Hou
Manki Min

Approved:



Hisham Hegab
Dean of Engineering & Science

Approved:



Ramu Ramachandran
Dean of the Graduate School

ABSTRACT

During the past decades, High-Performance Computing (HPC) has been widely used in various industries. In particular, the exponential growth of GPU (graphics processing unit) is a key technology that has helped promoting the development of artificial intelligence in real-world use cases. When we use GPU to accelerate parallel applications, its programmability, resource management, and scheduling are non-trivial jobs to obtain optimized performance. Therefore, how to effectively exploit GPU resources and improve program performance has been a hot research topic recently.

Benchmark does not always provide a good picture of the performance and details of the parallel applications. The various kinds of hardware devices and the constantly updated parallel programs make the performance analysis and modeling even more difficult.

In this dissertation, there are four main contributions. First, we conduct a study on the GPU analytical performance model, which aims to estimate the suitable number of threads per block for performance improvement.

Second, a novel method to elevate the limitation of GPU is proposed. This method offers a new way for optimization on GPU performance at the block schedule level.

Third, we propose two parallel computing abstract models, namely, the computational and programming models that represent various computing paradigms

based on Flynn's taxonomy and simplify the workload distribution characteristics. This framework provides a general way to create an analytical performance model.

Finally, we validate our proposed abstract models and demonstrate their usefulness with real-world applications in AI (Artificial Intelligence) on a distributed GPU system. The analytical performance model for CNN (Convolutional Neural Network) application analyzes performance characteristics on multiple GPUs, enabling users to evaluate their techniques before running applications on targeted machines.

APPROVAL FOR SCHOLARLY DISSEMINATION

The author grants to the Prescott Memorial Library of Louisiana Tech University the right to reproduce, by appropriate methods, upon request, any or all portions of this Dissertation. It is understood that “proper request” consists of the agreement, on the part of the requesting party, that said reproduction is for his personal use and that subsequent reproduction will not occur without written approval of the author of this Dissertation. Further, any portions of the Dissertation used in books, papers, and other works must be appropriately referenced to this Dissertation.

Finally, the author of this Dissertation reserves the right to publish freely, in the literature, at any time, any or all portions of this Dissertation.

Author _____

Date _____

DEDICATION

I dedicate this work to my wife and my parents for loving me and supporting me all the time.

TABLE OF CONTENTS

ABSTRACT.....	iii
APPROVAL FOR SCHOLARLY DISSEMINATION	v
DEDICATION	vi
LIST OF FIGURES	xii
LIST OF TABLES.....	xiv
ACKNOWLEDGMENTS	xv
CHAPTER 1 INTRODUCTION	1
1.1 Overview of GPU	1
1.2 GPU Performance Model.....	2
1.3 GPU Optimization	3
1.4 Analytical Model for Parallel Applications on a Distributed System	3
1.5 Dissertation Contributions	4
1.5.1 Block Size Estimation.....	4
1.5.2 GPU Dynamic Partitioning.....	4
1.5.3 General Parallel Computing Framework for Performance Analytical Model	4
1.5.4 Performance Model for CNN on a Distributed System.....	5
1.6 Outline of Dissertation.....	5
CHAPTER 2 BACKGROUND	6
2.1 GPU Architecture	6
2.1.1 Machine Model	6

2.1.2	Execution Model.....	7
	Single-Instruction-Multiple-Threads (SIMT).....	7
	Thread Hierarchy Mapping.....	8
2.1.3	Programming Model.....	8
2.2	Parallel Computing and Distributed System for Deep Learning.....	9
2.2.1	Deep Learning.....	9
2.2.2	Parallel Computing and Distributed System.....	9
2.2.3	Convolutional Neural Networks.....	10
CHAPTER 3 ANALYTICAL GPU PERFORMANCE MODELS WITH BLOCK SIZE ESTIMATION		12
3.1	Introduction.....	12
3.2	Related Work.....	16
3.3	Background.....	18
3.3.1	CUDA Programming Model.....	18
3.3.2	Threads and Blocks Scheduling.....	19
3.4	Analytical GPU Performance Modeling.....	20
3.4.1	Modeling GPU Hardware Specification.....	21
	Threads Hierarchy.....	21
	Relevant Characteristics and Limitation of Threads Hierarchy.....	23
	Memory Hierarchy.....	25
	Relevant Characteristics and Limitation of Memory Hierarchy.....	26
3.4.2	Modeling GPU Memory and Instructions Requirement.....	27
	Memory Requirement & Number of Instructions.....	27
3.4.3	Modeling Execution Performance.....	28
	Enough Warps to Cover Memory Latency.....	28
	Not Enough Warps to Cover Memory Latency.....	30

Compute-Intensive & Memory-Intensive Applications.....	31
Memory Accessing Analysis	32
3.5 Experimental and Result.....	33
3.5.1 Experimental Setup.....	33
3.5.2 Benchmarks.....	34
3.5.3 Results.....	36
3.6 Conclusion	38
CHAPTER 4 DYNAMIC PARTITION GPU MECHANISM FOR CUDA PROGRAM PERFORMANCE ACCELERATION	40
4.1 Introduction.....	40
4.2 Background.....	42
4.2.1 GPU Tasks Scheduling	42
4.2.2 Dynamic Partition Mechanism Workflow	43
4.3 Related Works.....	44
4.4 Partition Streaming Multiprocessors on NVidia GPU.....	46
4.4.1 Subset SMs on GPU.....	46
4.4.2 Mapping Control.....	47
4.5 Dynamic Partition SM Subset.....	49
4.5.1 Information Collection.....	49
4.5.2 Executing Time Estimation.....	50
4.5.3 Dynamic Partition	52
4.6 Evaluation	54
4.6.1 Methodology	54
4.6.2 Machine Environment.....	55
4.6.3 Experiment Result.....	55
4.6.4 Experiment Conclusion.....	60

4.7	Conclusion	60
CHAPTER 5 A PARALLEL COMPUTING FRAMEWORK FOR PERFORMANCE ANALYTICAL MODELS		
5.1	Introduction.....	62
5.2	Background and Related Work.....	64
5.3	Framework for Parallel Application Analytical Modeling.....	67
5.3.1	Parallel Application Abstract Model	68
5.3.2	Parallel System Abstract Model.....	73
	<i>TLD</i> Loading Data	74
	<i>TI</i> Instructions Passing	75
	<i>TC</i> Execution Time.....	76
5.3.3	Extended Flynn’s Taxonomy	77
	Single Instruction Stream, Single Data Stream (SISD)	78
	Single Instruction Stream, Multiple Data Streams (SIMD).....	78
	Multiple Instruction Streams, Single Data Stream (MISD).....	79
	Multiple Instruction Streams, Multiple Data Streams (MIMD)	80
	Heterogeneous Computing.....	81
5.4	Conclusion	82
CHAPTER 6 PERFORMANCE MODEL FOR CNN ON DISTRIBUTED GPU SYSTEM.....		
6.1	Introduction.....	83
6.2	Background.....	86
6.2.1	Convolutional Neural Network (CNN).....	86
6.2.2	Graphic Processing Unit (GPU) Architectures	87
6.2.3	CNN Training Process	88
6.2.4	Programming CNN to GPU	90

6.3	Parallel Computing Performance Model	91
6.3.1	Time of One Training Iteration.....	92
6.3.2	GPU Instruction Queue Model	94
6.4	Experiments and Evaluation	97
6.4.1	Layer Time Evaluation	98
6.4.2	Transmission Time Evaluation	100
6.5	Conclusion	101
CHAPTER 7 CONCLUSIONS AND FUTURE WORK.....		102
7.1	Conclusions.....	102
7.2	Future Work.....	104
BIBLIOGRAPHY.....		105

LIST OF FIGURES

Figure 2-1: GPU thread hierarchy mapping to hardware architecture.....	8
Figure 2-2: Simple GPU kernel.	9
Figure 2-3: LeNet-5 architectures.....	11
Figure 3-1: Overview of our analytical GPU performance model workflow.....	14
Figure 3-2: CUDA sample code.	19
Figure 3-3: Streaming multiprocessor working rule.....	20
Figure 3-4: Enough warps to cover memory latency.....	29
Figure 3-5: Not Enough warps to cover memory latency.....	31
Figure 3-6: The execution of each benchmark on Tesla M2050.	37
Figure 3-7: The execution of each benchmark on NVidia GTX650.	37
Figure 3-8: The execution of each benchmark on NVidia GTX970.	38
Figure 4-1: workflow of performance model analysis.....	44
Figure 4-2: Subset SMs in NVidia Tesla C2050.	49
Figure 4-3: Speedup of two same kind kernels using a dynamic partition on GTX970..	57
Figure 4-4: Speedup of two same kind kernels using a dynamic partition on Tesla M2050.	57
Figure 4-5: Speedup of two same kind kernels using a dynamic partition on GTX650..	58
Figure 4-6: Speedup of two different kernels with the same computational throughput using a dynamic partition on GTX 970.	58
Figure 4-7: Speedup of two different kernels with the same computational throughput using a dynamic partition on Tesla M2050.	59

Figure 4-8: Speedup of two different kernels with different computational throughput before using dynamic partition and after on GTX970.	59
Figure 4-9: Speedup of two different kernels with different computational throughput before using dynamic partition and after on Tesla M2050.	60
Figure 5-1: Overview of the parallel computing performance modeling framework.....	68
Figure 5-2: Examples of the four types of tasks.	69
Figure 5-3: Application logic example.	70
Figure 5-4: Overview of the extended Flynn’s Taxonomy.....	76
Figure 5-5: Single instruction multiple data streams.	78
Figure 5-6: Multiple instruction single data stream.....	79
Figure 5-7: Multiple instruction single data streams.	80
Figure 5-8: Multiple instructions multiple data streams.	81
Figure 5-9: Heterogeneous system.	82
Figure 6-1: The architecture of LeNet-5.....	87
Figure 6-2: Parameter server (PS).	89
Figure 6-3: The NVIDIA Collective Communication Library (NCCL).....	90
Figure 6-4: Memory intensive queue model. There are I Global Load instructions, M Shared Load instructions, L CP instructions, and K Global Store instructions in each block iteration.	95
Figure 6-5: Computation intensive queue model. There are I Global Load instructions, M Shared Load instructions, L CP instructions, and K Global Store instructions in each block iteration.	96
Figure 6-6: Comparison of runtime prediction for each layer in Alexnet (batch size 256).	99
Figure 6-7: Comparison of runtime prediction for each layer in Resnet-50 (batch size 256).	99

LIST OF TABLES

Table 3-1: Parameters of GPU thread hierarchy (obtained by <i>cudaGetDeviceProperties</i> (<i>struct cudaDeviceProp * prop, int device</i>) function in CUDA SDK).....	23
Table 3-2: Parameters of GPU memory hierarchy (we can get parameters by <i>cudaGetDeviceProperties</i> (<i>struct cudaDeviceProp * prop, int device</i>) function in CUDA SDK).....	25
Table 3-3: Specification of target GPU.....	34
Table 4-1: GPU hardware characteristic information.....	48
Table 4-2: Description of the benchmark.	55
Table 5-1: Parameters for the Parallel application and system abstract model.	71
Table 6-1: Computation notations.....	92
Table 6-2: Inputs parameters.....	98
Table 6-3: Resnet-50 data parallel comparison between actual runtime and model prediction in the PS mode.	100
Table 6-4: Resnet-50 data parallel comparison between actual runtime and model prediction in the NCCL mode.....	100

ACKNOWLEDGMENTS

I truly would like to thank my advisor, Dr. Chokchai (Box) Leangsuksun, who inspired me to start this extraordinary journey. He has always given me strong support. He has given me guidance on the experiment, research, and career planning. He has guided me to explore new topics and given me critical feedbacks in time. He has provided me opportunities to participate in both research and teaching. When I was confused on this journey, he always gave me confidence. Without him, I would not be able to complete this dissertation.

I would like to acknowledge Dr. Pradeep Chowriappa for his guidance in data science. I am grateful to Dr. Songming Hou and Dr. Manki Min for discussing ideas. Dr. Weizhong Dai for advising me through the CAM program.

In Ruston, my fellow students have made my life more rewarding. I would like to thank all my colleagues and friends: Dr. Supada Laosooksathit, Ajay Pasagada, Andrew Dale Touchet. Dr. Shi He.

Finally, I would like to greatly thank my wife and my parents for their unconditional support in my life. I also would like to thank my son Ayden, he is another reason for me to finish this dissertation.

CHAPTER 1

INTRODUCTION

High Performance Computing (HPC) has become increasingly indispensable and a main driving force for technology advancements. HPC has played significant roles in many scientific discoveries and engineering product design and development. Recently, the HPC gives a significant boost to Artificial Intelligence (AI) with the explosion of parallel computing performance such as GPU. Most of the supercomputers in the top500.org (TOP500, 2020) are equipped with GPUs as a co-processors. Thus, to obtain maximum efficiency from the computing systems equipped with GPU, a parallel application such as AI, performance analysis in the heterogeneous system is explored in this work.

1.1 Overview of GPU

GPUs have been employed for parallel computing for a decade, known as General-Purpose computation on GPU (GPGPU). For some applications, GPU can process hundreds of times faster than CPU counterpart. Researchers have attempted to harness the massive data parallelism with GPGPU to accelerate grand challenges in both scientific discovery and product advancements.

Typically, GPU is considered as a co-processor of the CPU in heterogeneous computing. The data which must be prepared on the host (CPU) is transferred and

executed on the computing device such as GPU and the results are sent back to the host side. A round trip of data between CPU-GPU is the main concern to obtain an overall good performance. In the larger scale HPC, applications are scaled out to multiple nodes in a distributed fashion. Not only the data round trip is a major concern but also the communication overhead among the nodes is important. Thus, the communication among nodes in the distributed system is influential as data transfer between host and device. As we know, the GPU execution model is Single Instruction Multi-Threading or SIMT. How programmers assign parallel tasks to GPU hardware is another crucial factor worth studying.

1.2 GPU Performance Model

In recent years, many researchers and developers have exploited the advancement of GPU for their applications and computational tasks. However, parallel computing is not a non-trivial job to obtain well-tuned performance in a short period for those who have little experience and a deep understanding of GPU and its optimization techniques. CUDA is NVIDIA runtime and tool for their GPU product. It provides a relatively user-friendly and flexible environment for programmers to develop their GPU applications. It is important to understand the GPU architecture, the CUDA programming paradigm, thread hierarchy, memory architecture, and various optimization mechanisms to obtain good performance. That is a challenge for most programmers and scientists who do not have too much GPU background and understanding.

A GPU performance model can help programmers and developers gain a deeper understanding of their applications on the targeted machine. Therefore, performance modeling becomes a vital foundation for further optimization.

1.3 GPU Optimization

In the previous section, we discussed performance factors in the CPU-GPU environment. Well-matching task assignment, communication, and memory access patterns with underlying architecture need to be optimized and fully utilized. A GPU scheduler usually controls tasks and groups of threads scheduling. On the CPU side, system APIs control the thread scheduling. Thousands of GPU threads need to be scheduled for their execution. However, the current GPU runtime does not allow programmers to have direct control for the thread block scheduling. This limitation hampers the way to optimize GPU programs, especially on the block level.

1.4 Analytical Model for Parallel Applications on a Distributed System

With performance enhancement on recent hardware, especially the GPUs, machine learning and deep learning applications have made revolutionary progress. Performance prediction becomes a burning desire of the parallel computing users to fine-tune their applications and to achieve more efficiency. Fortunately, analytical models are widely studied and employed to describe application performance characteristics. More and more analytical performance models have been recently developed for complex parallel applications such as a deep neural network (DNN). with multi-layers processed on a heterogeneous system, the parallel computing analytical performance model can be quite complex. Moreover, with the complex parallel systems, various hardware, and software components, it is challenging to develop an accurate analytical performance model for general hardware architecture and software logic. Furthermore, parallel computing architecture and programming paradigms continue to evolve. A minor change

in the architecture, interconnection network, or parallel algorithm may require extensive work to adapt to the change.

The successful analytical performance model must endure and adapt to these conditions. Therefore, a robust framework is a vital requirement as an enabling parallel computing tool and must be flexible to model users' logic on ever-changing targeted hardware while predicting accurate performance.

A framework for building the parallel computing abstraction models and analytical performance models are good guidance amid myriad variations.

1.5 Dissertation Contributions

The follows are the main contributions of this dissertation:

1.5.1 Block Size Estimation

We propose the GPU analytical performance model, which firstly considers the number of threads per block and estimates the suitable number of threads per block for performance improvement. The technique can be extended for other multithreaded parallel computing systems.

1.5.2 GPU Dynamic Partitioning

We present a novel method to elevate the limitation of GPU, which only allows one kernel to be executed in the device simultaneously. Our work offers a new way for optimization on GPU performance at the block schedule level.

1.5.3 General Parallel Computing Framework for Performance Analytical Model

we present two parallel computing abstract models. The models represent various computing paradigms based on Flynn's taxonomy and simplify the workload distribution

characteristics. An extension to Flynn's taxonomy is proposed to support heterogeneous systems and consider communication overhead.

1.5.4 Performance Model for CNN on a Distributed System

We present a comprehensive performance analysis model and demonstrate real-world applications that can predict performance and understand bottlenecks for CNN on GPU. Meanwhile, we analyze performance aspects for CNN on multiple GPUs, which will help users evaluate their techniques before running on targeted machines/architecture.

1.6 Outline of Dissertation

Chapter 1 gives an overview, motivation and current issues of this research work. Chapter 2 provides an overall background of the GPU and related topics. Chapter 3 presents our study on the performance model for GPU. Chapter 4 discusses our dynamic partition GPU optimization method. Chapter 5 presents a framework for building parallel computing abstraction models and an analytical model. Chapter 6 illustrates a demonstration of our proposed modeling techniques on a distributed GPU system. Chapter 7 concludes our research and recommends some future works.

CHAPTER 2

BACKGROUND

In this chapter, background narratives will provide some ideas. We present some background knowledge on modern GPU architecture and parallel computing concepts, as well as deep learning and related performance issues. We describe the GPU in three aspects: the GPU machine model (hardware architecture), the GPU execution model (thread hierarchy and mapping tasks to hardware), and the GPU programming model (kernel configuration and threads scheduling). We also introduce the parallel computing and distributed system for Deep Learning Convolutional Neural Network.

2.1 GPU Architecture

2.1.1 Machine Model

Typical GPU hardware consists of multiple Streaming Multiprocessors (SMs) (Nvidia, 2019) that share the L2 cache and DRAM controller through a cross the network on chip (NoC). The SMs are the core part of the GPU architecture, and they execute all vertex/geometry/pixel fragment shader programs and GPU programs.

The SM has multiple *scalar processor cores* (SPs) and two other functional units - the *double precision unit* (DPU) for *double precision* (DP) floating-point computation and the *special function-function unit* (SFU) for handling a priori functions and texture acquisition interpolation. Other components such as *register files* (RF), *load storage units*

(LSU), *scratchpad memory* (i.e., shared memory), and various caches (e.g., instruction cache, constant cache, and texture/read-only cache, L1 cache) on-chip data.

2.1.2 Execution Model

The GPU execution model is in data parallelism. We introduce the Single-Instruction-Multiple-Threads (SIMT) execution model and the thread hierarchy mapping of GPUs in this subsection.

Single-Instruction-Multiple-Threads (SIMT)

The GPU execution model is Single Instruction Multi-Threading or SIMT that is evolved from Single Instruction Multiple Data (SIMD) and is from the classical Flynn's taxonomy (Flynn, 1972). A kernel is a function that runs on the GPU side of heterogeneous computing (CPU+GPU) and contains thousands of concurrent lightweight GPU threads that are mainly divided into multiple thread blocks or collaborative thread arrays (CTAs). When the kernel is started, its CTAs are assigned to the SM. Depending on the available SM on-chip resources (e.g., registers and shared memory), it is possible to schedule multiple CTAs to the same SM. These resources are equally distributed among the concurrent CTAs of the SM.

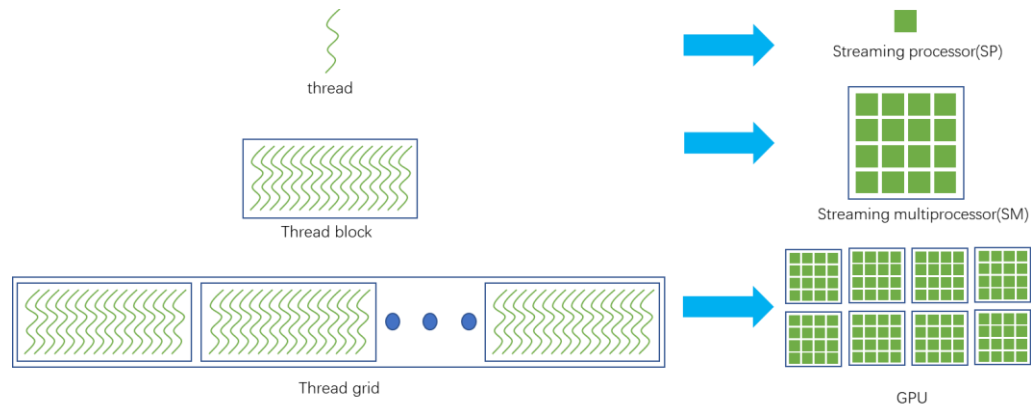


Figure 2-1: GPU thread hierarchy mapping to hardware architecture.

Thread Hierarchy Mapping

Figure 2-1 shows the mapping from the CUDA thread hierarchy to the GPU hardware architecture. It shows that (1) thread instructions are mapped to SP or SFU or DPU (in warp), (2) thread blocks or CTAs are mapped to SM, and (3) thread grids are mapped to GPU devices. Global memory, constant memory, and texture memory are shared among all threads in the grid while accessing shared memory is only available for threads in the same CTA. Register files and local memory are only available for the thread.

2.1.3 Programming Model

CUDA (Nvidia, 2019) is a language extension to C/C++ that allows programmers to define GPU subroutine or kernel functions. As discussed previously, the kernel is the function that runs on the GPU with massive parallel GPU threads. The way to specify the number of threads to execute the kernel is via the `<<<...>>>` configuration notation. As shown in **Figure 2-2**, which is a simple multiplication for 2D matrices, `<<<Grid_Dim, CTA_Dim >>>` implies that a kernel has a grid configuration defined by `Grid_Dim` and a `CTA_Dim`.

```

__global__ void MM(A,B,C){
int x=blockIdx.x * blockDim.x+threadIdx.x;
int y=blockIdx.y * blockDim.y+threadIdx.y;
C[x][y]+=A[x][y] * B[x][y];
}
MM <<<Grid_Dim,CTA_Dim>>>(A,B,C)

```

Figure 2-2: Simple GPU kernel.

2.2 Parallel Computing and Distributed System for Deep Learning

2.2.1 Deep Learning

Deep neural networks (LeCun, et al., 2015) are machine learning techniques that are good at discovering relevant structures in data in an unsupervised manner. Therefore, it is widely used in computer vision, speech analysis, and natural language processing.

The data structure information is stored in a distributed fashion. The model's information is distributed in different layers of the neural network, and the model information (weights) is distributed in different neurons. There are many ways to combine information into a layer distributed over different neurons and there are many ways to combine information across layers to minimize the loss function.

2.2.2 Parallel Computing and Distributed System

In classical neural networks, there are millions of parameters involved in defining the model and a large amount of data is required to train these parameters. These are computational-intensive processing that requires fast computing and networking capability. In the past, it typically takes a long time to train a deep neural network. Sometimes the dataset is too large to be stored on a single machine. Therefore, parallel computing and the distributed system are suitable solutions to improve training efficiency in recent years.

Parallel computing has made a tremendous impact on many areas during the past decades. With the recent development and advancement of GPU hardware, parallel computing becomes the most important tool for accelerating computational performance from simulations for scientific and engineering to artificial intelligence. Deep learning algorithms like CNN (LeCun, et al., 1989) get an enormous benefit from GPU parallel computing. Because the distributed system has a deeper neural network and bigger data set it has proved to be beneficial to processing grand, challenging tasks.

2.2.3 Convolutional Neural Networks

The training process of Convolutional Neural Networks (CNN) (LeCun, et al., 1989) is a typical feed-forward neural network. The basic structure of CNN consists of an input layer, a convolution layer, a pooling layer, a fully connected layer, and an output layer. Generally speaking, the convolutional layer and the pooling layer will be set alternately. The convolutional layer is the central part of CNN. In the convolutional layer, each neuron of the same feature map applies the same weight to the input data. The result of convolution is organized into a set of two-dimensional feature maps. All neurons in the feature map also use the same weight, which is called shared weight. The neurons in each layer are connected to the previous layer portion of the area. The purpose of using the pooling layer after the convolutional layer is to reduce the spatial size of the feature map while controlling the overfitting problem.

Let us take Lenet-5 (LeCun, et al., 1998) as a typical example to illustrate the architecture of CNNs. As shown in **Figure 2-3**, Lenet-5 is stacked by convolutional layer, pooling layer, and two fully connected layers. The input images are sent to the input layer

and then go through all convolutional and pooling layers. Finally, get to the full connection layer.

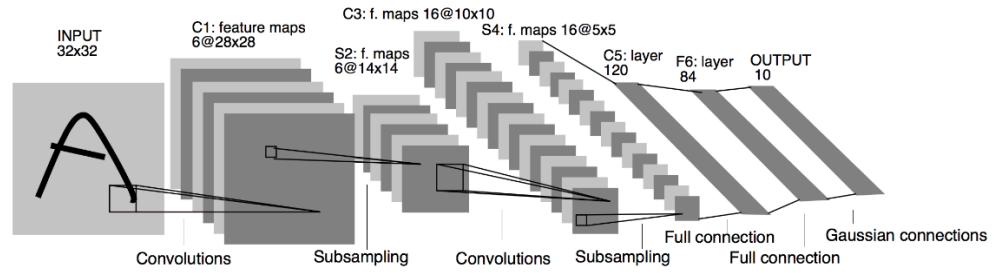


Figure 2-3: LeNet-5 architectures.

CHAPTER 3

ANALYTICAL GPU PERFORMANCE MODELS WITH BLOCK SIZE ESTIMATION

3.1 Introduction

During the past decades, HPC has played significant roles in many scientific discoveries and engineering product design and development. Its applicability ranges from mathematics, high-energy physics, biology, financial oil exploration, and recently, by the advantage of GPU computing in Deep Learning and AI applications. All these fields have one thing in common: massive parallel computation requirements. In recent years, GPUs have become one of the most successful parallel computing devices. Meanwhile, many programmers and developers are chasing the advancement of GPU for their applications and computational tasks. Especially, parallel computing powered by Nvidia GPU (Nvidia, 2019) is not a trivia job to obtain well-tuned performance in a short period for those who do not have too much experience and a deep understanding of CUDA GPU and its optimization techniques. CUDA relatively gives a more user-friendly and flexible environment for programmers than other parallel computing models. To squeeze the last drop of GPU performance, programmers and researchers have left no stone unturned to obtain improved GPU program performance. However, their expectation is always too optimistic. To understand the architecture and the behavior of the GPU, they need to get into the GPU genuinely, such as the CUDA programming paradigm, thread hierarchy, memory architecture, and various optimization mechanisms.

That is a big challenge for most programmers and scientists who do not have too much GPU background.

Before sending kernel functions to GPU, programmers need to figure out the number of threads per block and how many blocks to join the execution. The different number of threads per block can vary application performance. The main reason is related to GPU hardware resource management and the number of physical GPU cores. How to pick the right number of threads per block in a given application is a common issue for GPU users. To solve this problem and estimate the GPU execution time, we propose an analytical GPU performance model. It considers the GPU hardware specification, memory & instruction requirement, and the number of threads per block. The model can identify the application bottleneck and provide suggestions for optimization options like the number of threads per block for programmers to improve their applications without changing the code.

Our GPU performance model consists of three stages. First is a GPU hardware model showing all hardware specifications and limitations. Second is a memory and instruction requirements model, which analyzes the program code to get memory and instruction requirements. Third is a kernel execution time model to show the total execution time of GPU computing.

Our performance model has considered the most important factors in GPU computing. **Figure 3-1** shows an overview of our performance model workflow. First, we apply the parameters collector to get the parameters we need in the performance model. In the program code it includes thread and grid dimension, memory usage in the program, loops and branches, shared memory references, a data structure, memory requirement,

and algorithm branch divergence. In the targeted hardware, it includes device characteristics, such as the number of shared memory registers and global memory, memory bandwidth, and the number of bank; this information can be obtained from the device specification sheet or system function in CUDA. Our performance model can estimate the quantified performance and the right number of block sizes. These factors have represented most behavior of GPU computing. With the quantified result, programmers and developers can determine which factor affects performance most and figure out which parts of the program have potential improvement. With the right number of threads per block, programmers can improve their program performance without other changes.

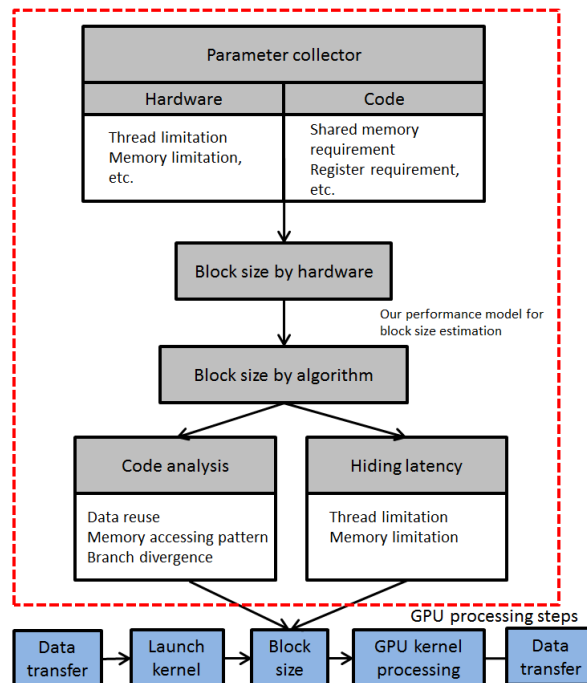


Figure 3-1: Overview of our analytical GPU performance model workflow.

In this chapter, we demonstrate the process of GPU performance quantitative analysis which helps programmers and developers understand the behavior of GPU

computing. By analyzing most essential factors of GPU computing, our models enable programmers and developers to figure out the bottleneck of their program and possibly improve the program with minimal code change.

The following are our contributions.

1. We propose the analytical GPU performance model, which firstly considers the number of threads per block. This can be easily extended to other parallel computing systems.

2. The analytical GPU performance model estimates the right number of threads per block for best performance. Programmers and developers can use such information to improve their application performance without any other changes.

3. Our model reveals the GPU computational behavior by analyzing hardware device characteristics, memory allocating, thread block organization, memory latency hiding, memory characteristic of memory hierarchies, coalesced memory, data reuse rate, and memory accessing pattern.

4. The analytical GPU performance model can potentially identify the program performance bottleneck without running the actual program on GPU.

This chapter is organized as follows. In Section 3-2, we discuss the related work. Section 3-3 introduces the GPU architecture and CUDA programming background. In 3-4, we describe the analytical GPU performance model. Section 3-5 verifies our model with the most representative experiments in the real world. Finally, we conclude in section 3-6.

3.2 Related Work

In recent years, there are some research and development in an analytical performance model for GPU kernel execution time estimation. For instance, Hong and Kim (Hong & Kim, 2009, June) introduced Memory Warp Parallelism (MWP) and Computation Warp Parallelism (CWP), and also showed the memory latency hiding mechanism. By finding the maximum number of memory warps to estimate the kernel execution time, Zhang and Owens (Zhang & Owens, 2011, February) have created a microbenchmark based performance model that considers performance from instruction pipeline, shared memory access, and global memory access. Bagsorkhi (Bagsorkhi, et al., 2010, January) analyzed each GPU kernel and discovered the bottlenecks by multiple benchmarks. These models help programmers estimate the execution time of the applications by analyzing the GPU kernel. Though those models can potentially point out the bottlenecks, programmers still need suitable optimization methods to improve the application performance. Moreover, the model requires many parameter inputs, and some of the parameters can only be obtained during the runtime. That imposes a big challenge for the programmers to evaluate their program without implementing it.

In an early GPU optimization work, researchers used numerous threads and set many threads to be executed at the same time to make GPU core busy by hiding the memory latency. Activating many threads at the same time is the most recommended way to improve the CUDA application performance. Many researchers presented threads scheduling, warp scheduling, even block scheduler techniques toward performance improvement. Kayiran (Kayiran, et al., 2013, September) employed the increased number of GPU cores which are on busy by allocating an optimal number of cooperative thread

arrays, the number of core cycles during which the pipeline is not stalled, and the number of core cycles during which all the warps are waiting for their data. All the optimization methods for the thread are making GPU core busy. All these methods must collect the application runtime information, like the number of idle GPU cores, on-chip memory usage, etc. This requires a plugin application or monitoring tool to get such information. The collector itself needs computing resources and it is not easy to be operated by users.

In GPU computing, there can be many data transfers between GPU and CPU by the limitation of the memory bandwidth and the data does not always fit into GPU memory (Bauer, et al., 2011, November), the data transfer time contributes significantly when compared to the entire execution time (Hong & Kim, 2009, June). Many researchers presented methods about using the on-chip memory to reduce bandwidth usage and increase data reuse rate. For example, Baghsorkhi (Baghsorkhi, et al., 2010, January) by the tightly coupled specialized DMA warps to improve the bandwidth usage and reduce the data transfer time. Some designed several data mapping/memory management algorithms to improve memory access efficiency (Jang, et al., 2010). These memory optimization techniques focused on using the on-chip memory to increase the data transfer speed. They just provided a way to optimize GPU applications, and it is still too difficult for programmers without much GPU background to improve their applications.

Most early works did not consider the size of the block and the size of the grid. They only used some factors which had huge effects on performance. Our research has studied how the block size affects application performance, also exposes the relationship between block size and other relevant factors.

We introduce a GPU performance model with block size estimation, which has considered the most important GPU behaviors, especially the number of threads per block. It discloses that the number of threads per block plays a critical role in resource distribution. Our model suggests programmers and developers the right number of threads per block to improve their application based on the given device configuration and application information.

3.3 Background

GPU architecture and CUDA programming model have been released in a few generations in the past decade. In this work, we focus on the three different generations of GPU devices and conduct experiments with CUDA 7.5. Studying benchmarks on various devices on the same platform can help us gain insight and also prove that our performance model can be effective and applied to other NVidia GPU devices.

3.3.1 CUDA Programming Model

NVIDIA introduced the CUDA, which is a general-purpose parallel computing platform for GPU in 2006 (Nvidia, 2019). CUDA programming model extends ANSI-C with a few keywords and constructs. It allows programmers and researchers to use high-level languages to build parallel programs, such as C or C++. It provides a user-friendly platform to take advantage of GPU with familiar programming languages. To construct a GPU kernel, a developer decomposes a parallel for loop into a grid of coordinated thread blocks. A block consists of coordinated scalar threads, and the threads with adjacent coordinates are implicitly grouped into a warp. During GPU execution, a thread block is mapped to an SM, and one SM can execute multiple thread blocks. A scheduler can synchronize threads in the same thread block with low overhead.

In heterogeneous computing, we call the CPU the host side. It generates the data and instructions. We call the GPU a coprocessor side where GPU takes the instructions and data from the CPU. We call kernel as a group of GPU instructions. During claiming kernel, programmers have to program functions and the block size. That means they should set how many threads are in a block and how many blocks be executed simultaneously. In this study, we have considered several factors that affect GPU performance. **Figure 3-2** shows a CUDA sample code that includes two values being set by programmers, i.e., the number of blocks per grid and the number of threads per block.

```

Int main ()
{
    .....
    cudaMalloc(...); /*allocate space*/
    cudaMemcpy(a_host, a_device, size,
    cudaMemcpyHostToDevice); /*copy values from host to
    device*/
    Kernel<<<gridDim, blockDim>>>( a_device); /*start
    executing on GPU*/
    cudaMemcpy(a_host, a_device, size,

```

Figure 3-2: CUDA sample code.

3.3.2 Threads and Blocks Scheduling

In the CUDA environment, threads in the same block can communicate and share data. A kernel consists of a grid executed by SMs and a streaming multiprocessor (SM) executes the block.

A GPU is comprised of groups of processors called SMs. Each SM can execute multiple blocks concurrently (Xiao & Feng, 2010, April). As illustrated in **Figure 3-3**, the scheduler assigns the blocks to the streaming multiprocessor. Still, there is a restricted number of resident blocks per streaming multiprocessor, which means some of the blocks need to wait until there are enough resources for new blocks. Programmers should ensure

that the kernel uses an appropriate number of threads per block to get better performance. Using CUDA cores and GPU on-chip resources effectively is one method to improve application performance. Hiding memory access latency is another way to reduce computing resource overhead. More details will be introduced in section 3.4. SM can execute multiple blocks at the same time. The standby blocks will get into the SM when there are enough resources available for them. As shown in **Figure 3-3**, there is no specific order for standby blocks.

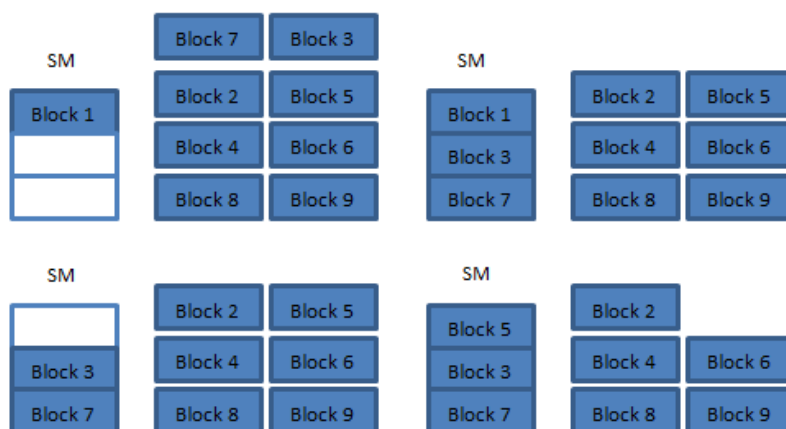


Figure 3-3: Streaming multiprocessor working rule.

When the block is assigned to SM, SM starts to execute a group of threads in the same block. A warp is the basic unit of the NVIDIA GPU scheduling. It is also the smallest executable unit in the CUDA code. Each warp is executed in single instruction multiple data (SIMD) or data parallelism, which means all threads in the same warp must be implemented with the same instruction simultaneously.

3.4 Analytical GPU Performance Modeling

This section presents three analytical models: GPU hardware model, GPU memory and instruction requirement model, and performance prediction model. These

three models are a pillar of our GPU analytical performance model. In the next section, we describe the model notations, limitations and show how to derive them.

3.4.1 Modeling GPU Hardware Specification

The CUDA general-purpose parallel computing platform provides three crucial abstractions: threads hierarchy, memory hierarchy, and barrier synchronization mechanism. These enable developers to understand performance factors and to improve hardware resource usage. Those abstractions may affect GPU computing performance. This section introduces the hardware model, which includes GPU hardware architecture such as threads hierarchy and memory hierarchy. Furthermore, we will put all the parameters from the aforementioned areas together to analyze the relationship between the number of threads per block and the program's performance.

There is a special structure composed of multiple groups of threads and multiple types of memory in the GPU. Each task is executed by a thread, and the task data stored in GPU memory, which is passed from CPU memory. The performance depends on the utilization of the hardware, like the number of parallel execution and the memory throughput. As such, thread hierarchy and memory hierarchy will be introduced and analyzed to obtain hardware performance characteristics.

Threads Hierarchy

GPU stream processor executes the kernel with multi-threads which are grouped by warp (Nvidia, 2019). In GPU computing, the warp is the basic unit in GPU scheduling. To maximize parallelism and increase the number of threads executed in the GPU processor, the programmers need to understand the thread hierarchy of each target GPU device with respect to their application. There are three layers for the threads. A

group of threads constitutes a block. The threads can be identified by three-dimensional within a block. For convenience, The CUDA programming uses *threadIdx* (Nvidia, 2019) to locate each thread position. Groups of blocks assemble into a grid. The thread hierarchy of GPU exhibits the three layers of the GPU threads. CUDA programs are then compiled and run on the GPU. GPU tasks are mapped to threads, and the scheduler will decide how to execute those threads in SM. That is how the warp threads and blocks are assigned to the stream processor and scheduled for execution in the processing cores. To get the best possible performance in a given CUDA program, the programmers need to understand GPU hardware features to configure the right size of a thread block.

In GPU computing, the execution configuration allows programmers to set the thread numbers and hierarchy for the kernel launch. This means how many threads are in each block and how many blocks are in the grid. There are three layers of threads in the GPU. The threads and blocks can be one, two, or three-dimensional. A user program can access thread position by the four built-in parameters: *threadIdx*, *blockIdx*, *blockDim*, and *gridDim*. The programmers need to understand the limitations of the target hardware to configure threads and their hierarchy for their applications. Next, we will discuss the limitations of a number of threads in a block and grid.

Table 3-1 shows parameters in the thread hierarchy. In the GPU thread architecture, each level has a limitation of the thread size, block size, and warp size. It depends on the capability of the GPU device for thread management, warps, blocks, and grids. However, each generation of NIVIDA GPU has different technical specifications. Before we compile the program on the targeted GPU, we need to figure out the feature of the GPU and understand the technical specifications.

Table 3-1: Parameters of GPU thread hierarchy (obtained by *cudaGetDeviceProperties* (*struct cudaDeviceProp * prop, int device*) function in CUDA SDK).

parameter	Description	Obtained
N_{TB}	Number of threads per block	Program
N_{BS}	Number of blocks per streaming multiprocessor	Hardware
N_{AW}	Number of Warps active in the SM	Hardware
N_W	Warp size	Hardware
N_{WB}	Number of warps per block	Hardware
N_{MTS}	Maximum number of resident threads per streaming multiprocessor	Hardware
N_{MTB}	Maximum number of threads per block	Hardware
N_{MBS}	Maximum number of resident blocks per streaming multiprocessor	Hardware
N_{MWS}	Maximum number of resident warps per streaming multiprocessor	Hardware
N_{SM}	Number of SM in GPU device	Hardware

Relevant Characteristics and Limitation of Threads Hierarchy

In the GPU thread hierarchy model, each level has several constraints concerning the number of threads. N_{TB} is the number of threads per block. N_{TB} will be specified in the CUDA function by the programmer, as shown in figure 3-1. After CUDA 2.0 version, the maximum number of threads per block is 1024. However, it was only 512 before CUDA 2.0. If a programmer specifies the number of threads per block larger than the maximum number of threads per block, the GPU runtime will report a warning and reduce the number of threads per block to satisfy the aforementioned limitation. N_{MTB} is the maximum number of threads per block. The number of threads per block that we set for the program must be equal to or less than the maximum number of threads per block as shown in **Eq. 3-1**.

$$N_{TB} \leq N_{MTB} \quad \text{Eq. 3-1}$$

In the block level, N_{MBS} is the maximum number of resident blocks per streaming multiprocessor and N_{MTS} is the maximum number of resident threads per streaming multiprocessor. By the same token, the total number of resident threads in streaming multiprocessor must also be equal to or less than the maximum number of threads per

streaming multiprocessor, as shown in **Eq 3-2**. In addition, the number of blocks per streaming multiprocessor N_{BS} must be equal to or smaller than the maximum number of resident blocks per streaming multiprocessor as shown in **Eq 3-3**.

$$N_{MTS} \geq N_{TB} \times N_{BS} \quad \text{Eq. 3-2}$$

and

$$N_{MBS} \geq N_{BS} \quad \text{Eq. 3-3}$$

A block of threads is executed in a streaming multiprocessor and organized into groups of parallel threads. Each group has 32 threads, and the group is called a “warp” (Nvidia, 2019). For example, in one block, there are 128 threads. The threads are grouped in 4 warps. With the CUDA rules, only a limited number of warps can be assigned for the streaming multiprocessor simultaneously, and we call them active warps. N_{MWS} is the maximum number of resident warps per streaming multiprocessor, and N_W is the size of the warp, means how many threads are in each warp. To get full utilization of the hardware, programmers typically specify multiples of 32 to be the number of threads per block. The maximum number of threads per streaming multiprocessor is equal to the maximum number of resident warps per streaming multiprocessor multiplied by warp size, as shown in **Eq. 3-4**.

$$N_{MTS} = N_W \times N_{MWS} \quad \text{Eq. 3-4}$$

According to the memory latency hiding mechanism, when one or more warps are waiting for the memory request, the warp scheduler will assign the other warps to a streaming multiprocessor processor. Once the waiting warps are ready, the warp scheduler will put those warps in the queue. Here, N_{WB} is the number of warps per block. The number of active threads in the block N_{AT} can be derived by **Eq. 3-5**.

$$N_{AT} = N_W \times N_{WB} \quad \text{Eq. 3-5}$$

Memory Hierarchy

There are multiple memory types in the GPU memory model, such as shared memory, register, global memory, constant memory, and texture memory. These memory properties are described in CUDA documentation (Nvidia, 2019). **Table 3-2** lists the parameters of the memory hierarchy. GPU performance depends on how threads in the block access memory. Different GPU devices and releases have different memory sizes, and perhaps each memory type may also have different memory latency. Programmers must understand the GPU memory model for further optimization.

Table 3-2: Parameters of GPU memory hierarchy (we can get parameters by `cudaGetDeviceProperties (struct cudaDeviceProp * prop, int device)` function in CUDA SDK).

Symbol	Parameter	Obtained
M_{MSM}	The maximum amount of shared memory per streaming multiprocessor	Hardware
M_{MSB}	The maximum amount of shared memory per thread block	Hardware
M_{TS}	Amount of shared memory required by each thread	Program
M_{BS}	Amount of shared memory required by each block	Program
R_{MRM}	Maximum number of 32-bit registers per streaming multiprocessor	Hardware
R_{MRB}	Maximum number of 32-bit registers per thread block	Hardware
R_{MRT}	Maximum number of 32-bit registers per thread	Hardware
R_{RT}	Number of 32-bit registers required by each thread	Hardware
R_{RB}	Number of 32-bit registers required by each block	Hardware

Global memory is a kind of memory that threads on different blocks can exchange data. It involves the DRAM and L1 L2, which impose high latency on accessing. Global memory is only used to store automatic variables and the compiler will use the global memory when there is no more on-chip space to store the variable. Usually, the large structures or arrays are placed in global memory. GPU devices typically have a large global memory size.

When all threads in the same block share data, the shared memory is available to all threads in the same block. The global memory access operation will reduce GPU efficiency. To alleviate this potential issue, setting the right number of threads per block according to the memory size will help improve memory access efficiency.

Relevant Characteristics and Limitation of Memory Hierarchy

M_{MSM} is the maximum amount of shared memory per streaming multiprocessor, and M_{MSB} is the maximum amount of shared memory per thread block. From code analysis, we can find out the amount of shared memory required by each thread. Here, we define M_{TS} be the amount of shared memory required by each thread. When each thread resource requirement exceeds the GPU device's limitation, the CUDA compiler will automatically reduce the number of active blocks in the streaming multiprocessor. In the following equations, we can find that the number of threads per block depends on the amount of shared memory required by each thread and the limitation of shared memory usage.

$$M_{BS} \geq M_{TS} \quad \text{Eq. 3-6}$$

$$M_{MSB} \geq M_{BS} \quad \text{Eq. 3-7}$$

$$M_{MSM} \geq M_{BS} \quad \text{Eq. 3-8}$$

$$M_{MSM} \geq N_{BS} \times M_{BS} \quad \text{Eq. 3-9}$$

$$N_{TB} \leq \frac{M_{MSB}}{M_{TS}} \quad \text{Eq. 3-10}$$

Typically, the GPU register has the same latency as shared memory and occasionally is lower than the shared memory. In the GPU memory hierarchy, we can also get the register information from CUDA *GetDeviceProperties* found in the CUDA SDK (Nvidia, 2019). Here R_{MRM} is the number of 32-bit registers per streaming

multiprocessor, R_{MRT} and R_{MRB} are the maximum number of 32-bit registers per thread and the maximum number of 32-bit registers thread blocks, respectively. R_{RT} is the number of 32-bit registers required by each thread. The number of threads per block can also be estimated by the register information, as shown in the following equations.

$$R_{RT} \leq R_{RB} \quad \text{Eq. 3-11}$$

$$R_{RB} \leq R_{MRB} \quad \text{Eq. 3-12}$$

$$R_{RB} = N_{TB} \times R_{RT} \quad \text{Eq. 3-13}$$

$$R_{MRM} \geq N_{BS} \times R_{RB} \quad \text{Eq. 3-14}$$

$$N_{TB} \leq \frac{R_{MRB}}{R_{RT}} \quad \text{Eq. 3-15}$$

3.4.2 Modeling GPU Memory and Instructions Requirement

In each CUDA program, the memory and the number of instructions requirements depend on the algorithm. The hardware resources involve the memory space and processor units. For compute-intensive applications, the programs require much more computing resources. In a compute-intensive program, maximizing the utility of streaming multiprocessor processors will help increase GPU performance. On the other hand, in memory-intensive applications, there are a lot of memory transfer operations. Reducing memory latency is the most effective optimization method to increase program performance. In this part, we focus on the program algorithm and memory latency hiding method.

Memory Requirement & Number of Instructions

During the CUDA compiler compiles the CUDA code, the compiler generates intermediate assembler level instruction, the NVidia PTX (Nvidia, 2019) translates the

instruction one by one with the binary microinstructions later on. In this part, we use the number of PTX instructions to count the number of instructions in the CUDA program.

The memory requirement includes the memory transfer operations such as read, write, also the size of needed space. For each thread, the instruction shows the memory size and operation requirement. We collect those parameters information from code analysis.

The programmers specify the number of threads per block in the CUDA kernel, so the size memory requirement is proportional to the number of instructions. With the right size of thread per block for each program, GPU hardware resources are efficiently used by programmers. Because the number of instructions per thread is related to how much data are transferred among memories.

3.4.3 Modeling Execution Performance

In GPU computing, the execution time includes two parts. One is the kernel execution time, and another is memory accessing time. In most conditions, memory access takes up half of the whole execution time (Gregg & Hazelwood, 2011, April). The memory accessing model is also as crucial as the kernel execution model, which could not be ignored. In this part, we present the analytical model for GPU performance time, including hardware resources and different memory access patterns.

Enough Warps to Cover Memory Latency

When there are enough active warps to be executed, the latency of accessing memory can be overlapped, as shown in **Figure 3-4**. We assume that all the warps have the same computing time T_M and memory accessing time T_C . When the product of the number of warps times and the warp execution time is larger than warp memory

accessing time as shown by **Eq. 3-16**, the GPU streaming processing core could keep running all the time. The active number of warps can be obtained by **Eq. 3-17**. The number of wraps is equal to the quotient of the total active threads and warp size.

$$N_W \times T_C \geq T_M \quad \text{Eq. 3-16}$$

and

$$N_w = \frac{N_{TB} \times N_{BS}}{w} \quad \text{Eq. 3-17}$$

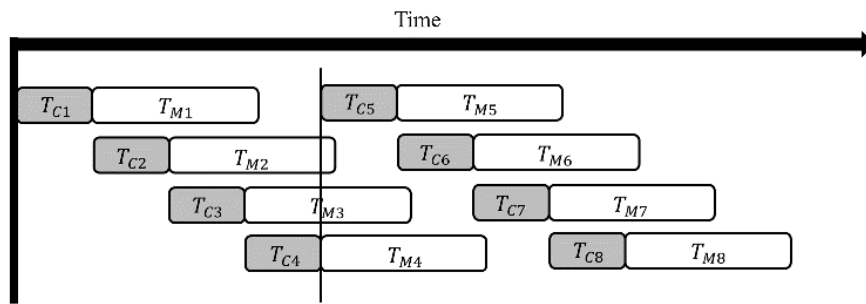


Figure 3-4: Enough warps to cover memory latency.

The total kernel execution time T_{Exe} consisted of total memory accessing time T_{Mem} and total kernel computation time T_{Com} , T_{Exe} can be represented by **Eq. 3-18**

$$T_{Exe} = T_{Mem} + T_{Com} \quad \text{Eq. 3-18}$$

The total memory accessing time is the sum of all memory accessing time. Here, we assume that all memory accessing times are the same. The whole computational time also depends on the GPU device feature and the computational clock cycles. Therefore, the T_{Mem} and T_{Com} are shown as **Eq. 3-19** and **Eq. 3-20**.

$$T_{Mem} = \sum_{i=1}^n T_{Mem_i} \quad \text{Eq. 3-19}$$

and

$$T_{Com} = \sum_{i=1}^n T_{Com_i} \quad \text{Eq. 3-20}$$

With enough warps in the SM, the latency time can be covered, as shown in **Figure 3-4**. The total kernel execution time is the last uncovered memory accessing time and the kernel execution time. Therefore, the total execution time is the product of the number of groups of the block assigned to GPU and the total computational time and last memory access time. The number of groups of blocks assigned to GPU is related the N_{CB} , the total number of blocks that need to be executed in the kernel. N_{SM} is the number of SM in the GPU, we can get N_{SM} from GPU hardware feature. And the N_{AB} is active blocks in each SM. N_{GOB} is the number of block groups assigned to GPU, it can be obtained by the following **Eq. 3-21**.

$$N_{GOB} = \frac{N_{CB}}{N_{AB}N_{SM}} \quad \text{Eq. 3-21}$$

Therefore, when the memory accessing time is covered by enough warps, the total execution time T_{Exe} can be obtained by the following **Eq. 3-22**.

$$T_{Exe} = (T_{Mem_n} + \sum_{i=1}^n T_{Com_i}) \times \frac{N_{CB}}{N_{AB}N_{SM}} \quad \text{Eq. 3-22}$$

Not Enough Warps to Cover Memory Latency

When there are not enough warps to be executed by the SM in GPU computing, the memory accessing latency cannot be covered. We assume that all the warps have the same computing time T_C and memory accessing time T_M . When the number of warps is smaller than the quotient of the memory accessing time and the warp execution time, the memory accessing time cannot be covered, as shown in **Figure 3-5**.

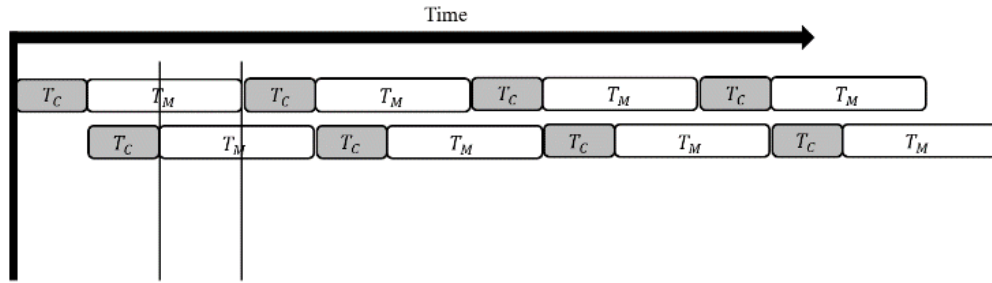


Figure 3-5: Not Enough warps to cover memory latency.

In this condition, the total kernel execution time includes the memory accessing time between two instructions, warp execution time, and the last memory accessing time.

The total execution time can be obtained as the following **Eq. 3-23**.

$$T_{Exe} = [(T_{C1} + ((\sum_{i=1}^n T_{Mem_i} + \sum_{i=1}^n T_{Com_i}) / \frac{N_{TB} \times N_{AB}}{N_{Sw} \times N_{W}}))] \times$$

Eq. 3-23

$$\frac{N_{CB}}{N_{MBS} N_{SM}}$$

Compute-Intensive & Memory-Intensive Applications

In the CUDA program, programmers like using more threads to cover the memory latency. Although programmers gain performance on some applications, the results are always not as good as expected. We propose compute-intensive & memory-intensive applications for performance model building. Using numerous threads to increase parallelization, the SM processor gets busy, and the CUDA application's performance gets better. We call this kind of application compute intensive. Oppositely, in memory-intensive applications, Using numerous threads to increase parallelization may reduce the GPU performance because multiple threads access the memory at the same time. To identify the type of applications: compute-intensive, memory-intensive or other, we present the processor active time rate. It represents the utilization of an SM processor. In our model, when the processor active time rate is higher than 60%, the

application will be categorized as a compute-intensive application. When the processor's active time rate is less than 20%, the application will be categorized as a memory-intensive application. For the compute-intensive application, the number of parallel threads should be increased. For the memory-intensive application, the data reuse rate should be increased, and the number of parallel threads should be reduced.

Memory Accessing Analysis

The access latency of the GPU global memory is very high compared to shared memory latency. The global memory latency can be as high as 400-800 clock cycles (Mei & Chu, 2016) per access. The CUDA program's memory usage will help dissect the memory requirement. Other factors also affect memory access efficiency. For example, in global memory accessing, memory coalescing is a significant influencing factor. For global memory accessing of a half-warp, if certain conditions are satisfied, the memory transactions can coalesce into one or two transactions (Jia, et al., 2012, June). The required conditions depend on the GPU hardware and computing capabilities of CUDA. If threads of one half-warp access adjacent memory elements, that is the memory coalescing. However, If the coalesced conditions cannot be satisfied, more memory transactions are needed, and performance will be reduced due to more memory accessing time.

By the code analysis, we can figure out the memory coalescing rate. We count the number of warp requests and recognize each request by hand. We set N_{Unco} to be the number of un-coalescing memory requests and set N_{Co} to be the number of coalescing memory requests. Each request rate is shown as following equations.

$$R_{Unco} = \frac{N_{Unco}}{N_{Co} + N_{Unco}} \quad \text{Eq. 3-24}$$

and

$$R_{Co} = \frac{N_{Co}}{N_{Co} + N_{Unco}} \quad \text{Eq. 3-25}$$

With the memory accessing rate, we can get the memory accessing time more accurately. Therefore, the memory accessing time with the memory type is shown as **Eq. 3-26**.

$$T_{Mem} = \sum_1^n T_{Munc_n} \times \frac{S_{Sha}}{S_{Buf}} \times \frac{N_{Unco}}{N_{Co} + N_{Unco}} \sum_1^m T_{Munc_m} \quad \text{Eq. 3-26}$$

$$\times \frac{S_{Sha}}{S_{Buf}} \times \frac{N_{Co}}{N_{Co} + N_{Unco}}$$

3.5 Experimental and Result

In this section, we introduce the hardware for the experiment and the benchmark used in this experiment.

3.5.1 Experimental Setup

We evaluate our performance model on three generations of GPU with five representative real-world GPU micro-benchmarks. The three GPUs are Tesla C2050, GTX650, and GTX970. Each specification of the GPUs is shown in table 3-3.

Table 3-3: Specification of target GPU.

Model	C2050	GTX650	GTX970
Streaming Multiprocessors	14	2	13
Processor Cores	448	384	1664
Processor Clock	1147MHz	1110.5MHz	1050MHz
Memory size	2G	2GB	4GB
Computing version	2.0	3.0	5.2

With a suitable number of threads per block, the program will get better performance compared with the real kernel execution time which is measured by using *cudaEventRecord* to record the data transfer start and kernel end. The total execution time is the sum of the kernel execution time and the data transfer time which are gained from *cudaEventRecord*. We run our five benchmarks on a three-generation GPU, respectively. We run all benchmarks twenty times for the different number of threads per block on each GPU. The final real-time of GPU processing represents the arithmetic means of twenty times execution.

3.5.2 Benchmarks

To verify our performance model can predict a suitable number of threads per block in GPU computing. We use six representative benchmarks in the real world to verify that our performance model can predict a suitable number of threads per block in GPU computing. The shared memory and register requirement by each thread is obtained manually. The program algorithm gets the number of instructions in each thread. The memory accessing pattern has coalesced and un-coalesced. We estimate the rate of each accessing type by code analysis. The rest of the hardware and program features can be gained from the information collector.

1. Matrix multiplication is a known benchmark for parallel computing. Here we use naïve Matrix multiplication and optimization with tiled Matrix multiplication as a test program to verify our model. The naïve matrix multiplication is taken from the NVidia CUDA sample without any optimization. Another matrix multiplication uses tile to increase the utilization of the shared memory. This algorithm will reduce the time to access global memory because of the data loaded from global memory. The change in shared memory requirement will affect the number of threads. Our experiment will run the benchmarks with a various number of threads per block.

2. Tridiagonal solver is tridiagonal linear systems (Zhang, et al., 2010) which are crucial systems to solve many problems in numerical analysis and computational fluid dynamics. The cyclic reduction is a popular parallel algorithm that can take advantage of GPU to solve the tridiagonal linear system. Tridiagonal Solver for Linear equations is critical for many scientific and engineering problems and real-time or interactive applications in graphics processing, video games, and 3D films. The applications of tridiagonal solvers include alternating direction implicit (ADI) methods, spectral Poisson solvers, cubic spline approximations, numerical ocean models, semi-coarsening for multi-grid solvers, and preconditioners for iterative linear solvers.

3. List ranking is one of the fundamental operations with applications to several problems. List ranking does not work well in sequential computing. The difficulty of using list ranking in parallel computing is recognized early by Ranade (Ranade, 1998). Using various techniques, several algorithms to solve this difficulty are proposed later on (Anderson & Miller, 1990). In this case, we focus on the local ranking aspect by Hellman- JáJá algorithm (Helman & JáJá, 1999, January).

4. LU decomposition, here we use LUD briefly. In numerical analysis and linear algebra, the lower upper factorization is used to solve a square system of linear problems. In the LUD, there is a loop in the kernel and the loop will access shared memory frequently. The shared memory utilization and the limitation will affect the execution result. Because most memory accessing is from shared memory, the memory latency will be covered.

5. Hotspot (Che, et al., 2009, October), it is a widely used differential equations algorithm for simulating processor temperature. The average temperature values of the microarchitecture's mapping area are represented by the output cell in the grid. In a 3×3 neighbor grid element, one can find the center element's temperature value.

3.5.3 Results

Figure 3-6, 7, 8 show the estimated execution time of our performance model and the measured execution time on the three different NVidia GPU cards. For each benchmark, we use 64 to 1024 threads per block to execute some programs. From the results, we have found that compared to others, the performance is better in some special block sizes such as 256, 512, and 128.

From the results in the following figures, we prove that using the default setting to run a CUDA program on the NVidia GPU does not always get the best performance, even though naïve matrix multiplication using default block size 256 can get the best performance on both generations GPU. After we run the matrix multiplication with optimization, the results from GTX 650 and GTX 970 show that the performance with block size 512 is better than the performance with block size 256. This situation also happens in List ranking. When we use GTX 970 with block size 320, block 512 can get

the same performance as block 256. In the GTX 650 when block size 128, the performance is better than others.

Even though when we run benchmarks on different NVidia GPU cards, the results have some disparity. Our performance model still works well for estimating the best block size for each program.

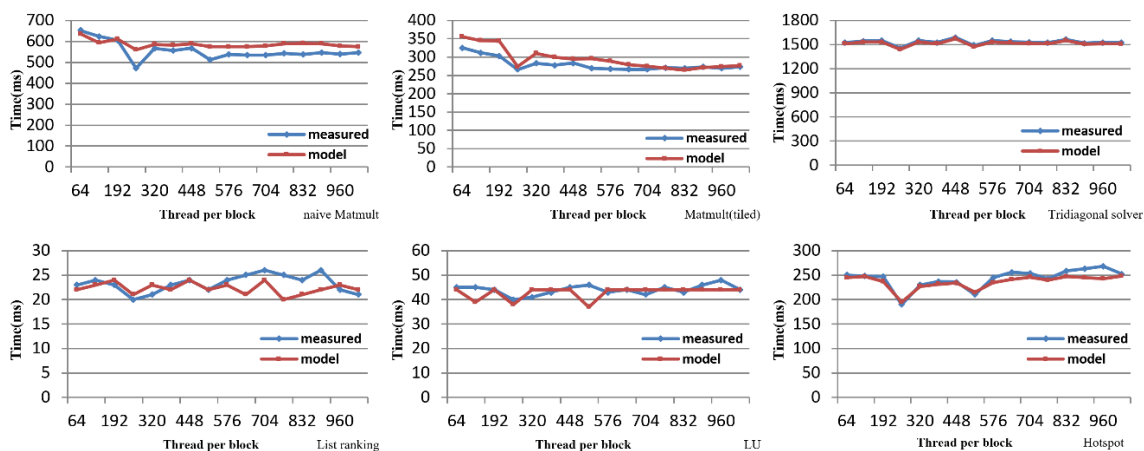


Figure 3-6: The execution of each benchmark on Tesla M2050.

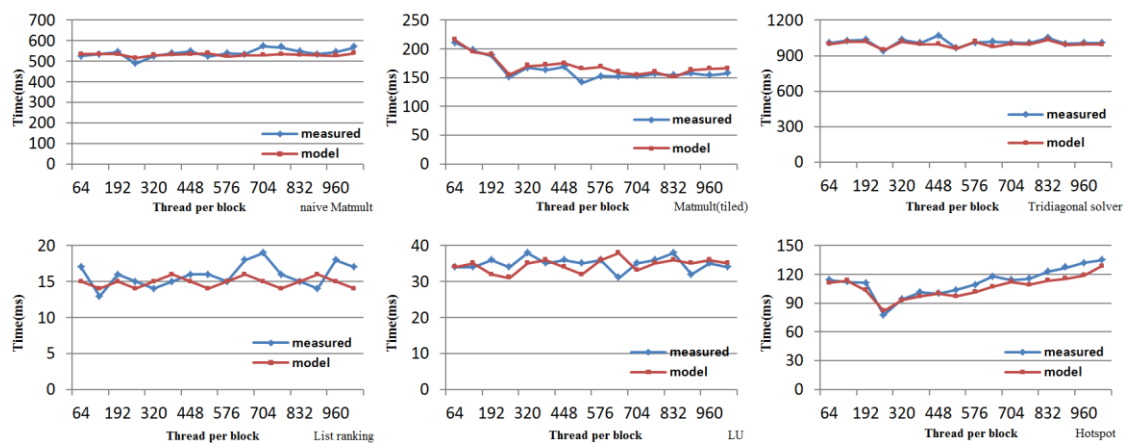


Figure 3-7: The execution of each benchmark on NVidia GTX650.

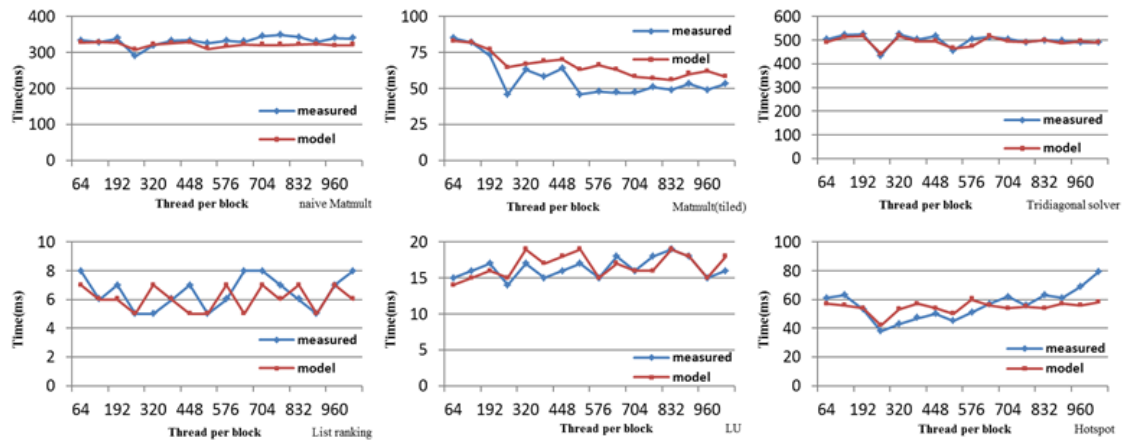


Figure 3-8: The execution of each benchmark on NVidia GTX970.

3.6 Conclusion

This chapter proposed a GPU performance model with the block size estimation, which analyzed the most important factors in GPU computing and considered the most critical cases. Our model revealed GPU computational behavior by analyzing hardware device characteristics, memory allocation, thread block organization, memory latency hiding, memory characteristics, memory hierarchy, coalesced memory, data reuse rate. Our model clarified the relationship between the number of threads per block and other factors. We validated our GPU performance model with six representative real-world GPU programs. The results showed that our model yielded good accuracy in performance estimation and verified that the right block size setting can help improve the execution efficiency of the application. Moreover, choosing the right block size was a new way to reduce the execution time of an application without editing any code. When we used our model to analyze GPU programs, our model can predict time-consuming parts and bottlenecks of the program and potentially optimize parts of the program. Last, using our

model to analyze applications and estimate the correct number of threads per block will help programmers and developers quickly enhance their programs.

CHAPTER 4

DYNAMIC PARTITION GPU MECHANISM FOR CUDA PROGRAM PERFORMANCE ACCELERATION

4.1 Introduction

In recent years, we have witnessed the increasing popularity of Graphics Processing Units (GPUs) for general-purpose computing, thanks to the large number of parallelisms provided by GPUs and their cost-effectiveness. With hundreds of processing cores equipped, the GPU can render thousands of threads for parallel applications. Numerous parallelisms produce not only huge potential throughput but also imposes grand challenges for thread management or scheduling.

It is essential for well-matching communication and memory access patterns with underlying architecture to use the parallelism fully. Task scheduling is usually controlled by a scheduler in the GPU. On the CPU, the thread scheduling is controlled by system APIs. However, on the GPU side, there is no such API; the scheduling on GPU has been implemented through hardware and some proprietary mechanism. a large amount of threads need to be scheduled in a short time. However, the current GPU does not allow programmers to schedule the thread blocks. This limitation hampers the way to optimize GPU programs, especially on the block level.

Applications on GPUs and GPU/CPU heterogeneous systems are typically written in a combination of data & task-parallelism manners that allow the runtime system to

handle the device and task scheduling. Traditionally, the programmer provides the largest possible task to load to the GPU, and generally speaking, the larger the tasks are, the better it will perform on the GPU. While large tasks usually provide a better performance, there are two situations where this may not be the case. First, the tasks could not fully use the GPU computing resources, which means some of the cores or on-chip memory are idle. Such tasks have limited scalability on the GPU. In this case, if a task could be scheduled on a smaller set of GPU SMs rather than the whole GPU, and let another task be executed on the rest of GPU SMs at the same time, the GPU utilization and efficiency would be higher. Second, applications can only issue or process a large task, which means some precedence rules limit the task-level parallelism, like processing multiple small tasks at the same time.

In our work, we overcome both thread scheduling and task scheduling by enabling more than one task executed on the GPU simultaneously. We achieve this by partitioning the GPU processing unit, called SMs into multiple segments and forcing each task to be performed on the specific SMs subset. With this approach, we can control the tasks on the block level and task level to allow multiple tasks to run on the GPU at the same time. By using the GPU performance model, we can estimate and dynamically set the GPU SMs subset depending on the application kernel execution time and the scalability of each task.

We evaluate our approach with real-world benchmarks. For comparison, we have implemented three scenarios in our experiment. All three cases are run on three different GPU cards. As a result, the demonstration shows that a potential performance in the GPU

application benefits from performing the high parallelism tasks and increasing GPU efficiency.

The following are the main contributions of our works.

1. We present a novel method to eliminate the limitation of GPU, which only allows one kernel to be executed in the device simultaneously. Our work offers a new way for optimization of GPU performance at the block schedule level.

2. We employ GPU performance modeling to estimate kernel computational throughput.

3. We dynamically partition the SMs for each kernel based on the kernel computational throughput.

4. A demonstration shows GPU's application performance by our technique up improved to 10% without any change in the algorithm.

The rest of this chapter is organized as follows. Section 4-2 introduces the background of GPU characteristics and block scheduling. Section 4-3 reviews some related works. Sections 4-4 and 4-5 present the dynamic partition method for GPU. Section 4-6 details a demonstrative experiment to verify our model with multiple benchmarks on three different GPU devices. Section 4-7 is the conclusion of this work.

4.2 Background

4.2.1 GPU Tasks Scheduling

In the current situation, CUDA API does not provide a mechanism to interrupt a kernel that has already started execution. Lacking the manual scheduling mechanism limits the traditional resource scheduling model to access GPUs, because GPU resources

cannot be accessed in the same way as the CPU. GPU runtime keeps kernels ready in a release queue and processes them on a first-come, first-served basis.

When CPU processes try to start a kernel, if a kernel is already running on the GPU, other kernels in the queue will be blocked until the first kernel finishes. The wait time for a kernel call is also affected by how many kernels are in the queue. The longer the queue, the greater the wait time for the CPU thread waiting for the result.

Concurrent execution of kernels on single GPUs was first supported by NVIDIA Fermi GPUs. The left-over policy is used on Fermi GPUs, which allows scheduling concurrent kernels only if the required number of computing units is available. NVIDIA's Kepler GPUs achieve concurrent execution of kernels using Hyper-Q technology (Nvidia, 2019), which employs multiple hardware queues to avoid false dependencies between computations. In this work, we partition the GPU processing units as multiple tasks to be executed simultaneously.

4.2.2 Dynamic Partition Mechanism Workflow

Our dynamic partition mechanism separates SMs in GPU into multiple parts by choosing the SMs to execute a kernel. For the rest of SMs, it allows another kernel to run on them. Our workflow is shown in **Figure 4-1**. First, we will collect most of the parameters that consist of hardware and application characteristics. It includes the limitation of the thread, block, memory, register in GPU device, the data structure, loop function, data transformation, number, and kernel algorithm. We apply these parameters to our GPU performance model to estimate kernel computational complexity. With each kernel's complexity, we calculate the current GPU's execution time that aims to reduce the possibility of SMs on idle. We set the number of SM in each part of the SM group.

Then we insert the partition mapping information in front of each kernel and allow more than one kernel to be launched on the GPU at the same time.

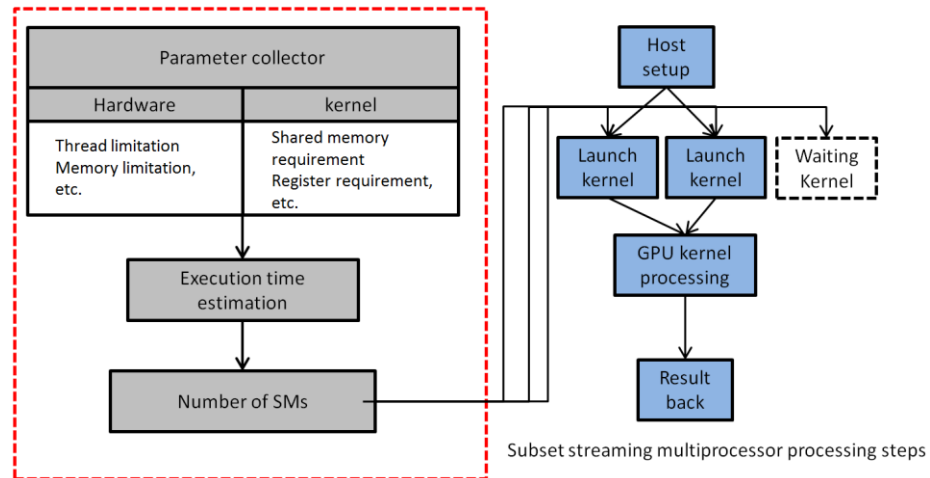


Figure 4-1: workflow of performance model analysis.

4.3 Related Works

In recent years, there is numerous research dedicated to GPU utilization and its performance. Previous works presented many solutions based on two areas, namely, hardware and software aspects. On the hardware level, it uses benchmark or simulation to estimate kernel execution time and then makes the GPU kernel scheduler smarter based on the processing performance information. On the other hand, the software level uses kernel transforming algorithm to improve low parallelism rate kernel to high parallelism kernel. In addition, it uses merging technology to combine multiple lightweight kernels with being a big one. These works for either grouping two or more kernels before launching them to the GPU or adjusting the kernel structure to fit the targeted GPU device.

Awatramani (Awatramani, et al., 2013, October) found out the impact on the throughput of mixed and partitioned execution pairs of kernels through simulation. Their

results showed that mixing bandwidth and compute-bound blocks was often beneficial. To predict performance, they must know the exact kernel pairing. Our work differs from this in that we partition the execution of two or more kernels and evaluate a complete complexity of application executing on a heterogeneous commodity platform, rather than just a single kernel in the simulation.

Gurevara (Guevara, et al., 2009, September) proposed a software solution to achieve concurrent kernel execution by merging pairs of kernels. They found that small kernels do not occupy the entire GPU. Therefore, GPU applications can get benefit from the merging. However, the number of resources used by the merged kernels may reduce the SM occupancy and limit the gains. In addition, merging cores with unequal execution times can cause cores with shorter runtimes to be bogged down and take up more of their resources. Our work leverages existing hardware to achieve true independent kernel co-execution and is not limited to pairs of kernels.

Pai (Pai, et al., 2013) proposed extended iterative packing to reduce the kernels' parallelism to allow pairs of kernels to execute together on the same SM. They statically reserved resources for a second kernel in the SM, but they cannot control the degree of concurrency due to the differences in resource utilization of the kernels, and the use of an even allocation strategy. Their evaluation used the CUDA API rather than a full heterogeneous application. Our work is similar, but we do the partitioning from the hardware level even before we launch the kernel. By doing so, we largely avoid sharing computational resources and thus unpredictable co-execution performance while giving us full control over the degree of co-execution and the ability to support more pairs of applications.

Wu (Wu, et al., 2015, June) proposed a similar software solution using partitioning. Their approach was to fetch each SM fetch block from a centralized queue at runtime rather than partition the hardware before launching the kernels to GPU as we do. While centralized block fetching provides more flexible scheduling, the global load required to do so hinders compiler optimization due to the introduction of new dependencies. Our technique is to evaluate the common execution of the kernel on a complete application and provide accurate results.

4.4 Partition Streaming Multiprocessors on NVidia GPU

With the requirement of executing multiple kernels on the GPU simultaneously, there must have been multiple independent stream multiprocessors and independent memory resources for each kernel. The GPU device only allows one kernel to be executed by streaming multiprocessors at a time. By this limitation, we need to find a method to enable us to run more than one kernel on the GPU simultaneously. This section will introduce our approach that separates the SMs into multiple parts for the individual kernel.

4.4.1 Subset SMs on GPU

To allow multiple kernels to run on the GPU concurrently, we present a subset SM mechanism to isolate kernels to particular groups of SM, which can effectively partition the GPU. With this mechanism, kernels are executed on different groups of SM, which can avoid sharing compute resources between each other. In this mechanism, each group of SM only executes the blocks from the same kernel. This feature helps us avoid much of the un-foreseeability in the parallel kernel execution. With this mechanism, we can specify the number of SMs in each group for executing the kernel's blocks. This

allows us to apply the GPU performance model to identify possible bottlenecks and to improve performance by using dynamic partitioning SMs.

The key to our co-execution approach is that if we restrict a kernel to execute on only a subset of SMs, then there will always be remaining SMs available. Therefore, the NVIDIA scheduler will execute the subsequently launched kernels on these remaining SMs simultaneously. In this way, we can force the GPU to co-execute multiple kernels and control the resources of GPU SMs for each kernel.

Commonly, by executing the kernel in the GPU, it can create many threads, which are often organized into three hierarchies. The execution of the GPU kernel can be understood as many tasks that are handled by multiple workers in the GPU SMs. For example, the workers mean a group of GPU threads, also called blocks, and the tasks mean the operation conducted by the blocks. Since the job enters the GPU, a unique ID will be created for each job to identify itself.

The key to our subset SM mechanism is that when we force the kernel to be executed in the subset SM in the GPU, there will be some idle SMs in the GPU, which means this resource is being used to perform the new kernel without sharing the resource concurrently executing kernel in the GPU. When the scheduler detects some idle isolated SMs in the GPU, it will arrange a new kernel into the device to be executed. At this moment, we are truly running more than two kernels concurrently on the GPU and controlling which group of SM to execute them.

4.4.2 Mapping Control

Before launching the kernel, on the host side, we insert a control code at the start of each kernel that reads the ID number of the SM, it will help the blocks in the kernel go

to the right SM. GPU characteristic information can be obtained by *devicequery.cu*, which is in the CUDA sample code. In the characteristic information report, we get the number of SMs in the device and other related info. For example, if we want to separate the SMs in NVidia Tesla C2050 GPU into two parts, by the information from the *devicequery.cu* report shown in **Table 4-1**, it contains 14 SMs in the GPU; if we want to set 6 SMs to one part, the other has 8 SMs, while we want the first kernel to run on the group with 6 SMs.

After this short setup, the GPU can execute more than one kernel concurrently. With the different settings in the front of the kernel, we group more kernels in the GPU and execute them simultaneously. For example, subset SMs in NVidia Tesla C2050 is shown in **Figure 4-2**.

Table 4-1: GPU hardware characteristic information.

Model	C2050	GTX650	GTX970
Streaming Multiprocessors	14	2	13
Processor Cores	448	384	1664
Processor Clock	1147MHz	1110.5MHz	1050MHz
Memory size	2G	2GB	4GB
Computing version	2.0	3.0	5.2

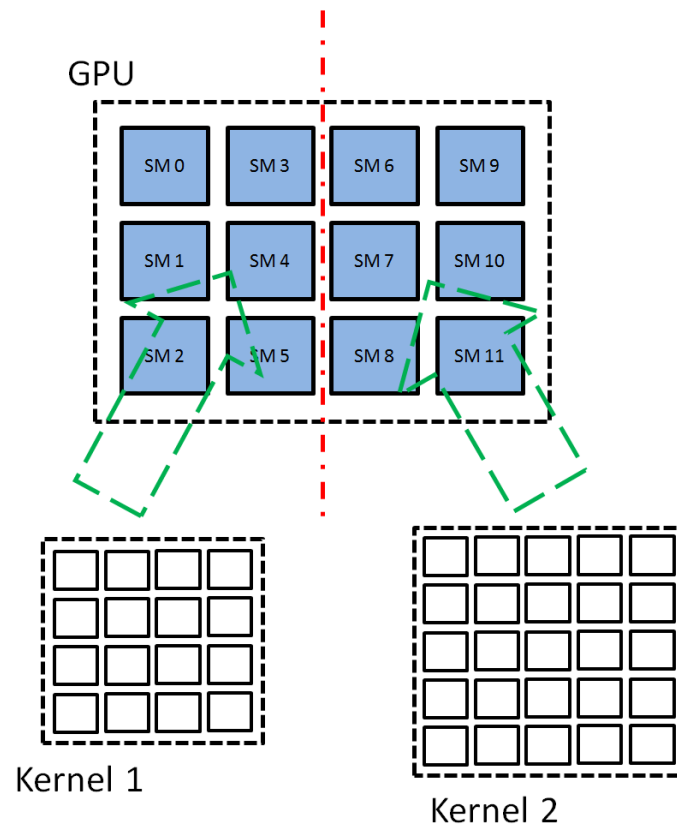


Figure 4-2: Subset SMs in NVidia Tesla C2050.

4.5 Dynamic Partition SM Subset

Subset SM mechanism aims to eliminate the limitation of GPU schedule, which allows only one kernel on the GPU when the application consists of multiple kernels. The idea is how to distribute kernels into an independent group of SMs effectively. In this work, we will introduce GPU performance modeling to dynamically partition the SMs into multiple groups.

4.5.1 Information Collection

To eliminate the sequential kernel execution on a GPU, we need to collect the kernel information, including a data structure, memory requirement, and the block size setting from the application. On the other hand, we also need to figure out the number of

kernels and computational throughput of the kernel for dynamically assigning kernels to the specific group of SMs.

Based on the current GPU toolset, we can find the related memory and other resource information that can be applied to the kernel execution estimation. Each GPU device has different hardware information. Whenever the new device is deployed, we need to figure out the GPU information with *devicequery.cu*, then use a counter to get the total number of kernels N_k in each program, and the kinds of kernels.

4.5.2 Executing Time Estimation

In this process, we assume that all the kernels are independently executed with full resources. With the executing time, we can figure out the probable computational throughput rate for each kernel.

In GPU computing processing, in addition to the execution time, memory access time is also very important. Total kernel execution time T_{Exe} consists of total memory access time T_{Mem} and total kernel execution time T_{Com} . This can be represented by **Eq. 4-1**.

$$T_{Exe} = T_{Mem} + T_{Com} \quad \text{Eq. 4-1}$$

The total memory access time depends on the total number of memory clock cycles needed and the memory frequency. We suppose that all memory accessing frequencies are the same. We analyze the memory access time based on the memory types. The total computational time depends on the GPU device's feature and the clock cycles. So, the T_{Mem} and T_{Com} can be obtained by **Eq. 4-2** and **Eq. 4-3**.

$$T_{Mem} = \sum_{i=1}^n T_{M_i} \quad \text{Eq. 4-2}$$

and

$$T_{Com} = \sum_{i=1}^n T_{M_i} \quad \text{Eq. 4-3}$$

There is a wait time for the memory to be read, during that time the SP units cannot process because it needs to wait for the returned data, we call the time of SP without working as idle time. Two common situations may occur during GPU execution. One is that idle time can be covered in GPU computing, and another is that idle time cannot be covered in GPU computing.

First, the idle time is covered in GPU computing. This means the SP in GPU keeps running from the beginning to the end. There is no idle time during kernel executing due to there being enough warps for SP units to execute. The total execution time T_{Exe_cov} is the sum of the total computational time and the last memory access time as shown in **Eq. 4-4**. In section 3.4.3, there are more details about the execution time estimation.

$$T_{Exe_cov} = (T_{M_n} + T_{Com} \times \frac{N_{TB} \times N_{MBS}}{w}) \times \frac{N_{CB}}{N_{MBS}N_{SM}} \quad \text{Eq. 4-4}$$

The N_{CB} is the total number of blocks that are executed in the kernel. We can get N_{CB} from code analysis.

In the second case, if there are not enough warps to cover the idle time. We have to consider the idle time for each warp. Therefore, the total execution time can be represented by **Eq. 4-5**.

$$T_{Exe_uncov} = [(T_M + T_{Com} + T_{C_n} \times (\frac{N_{TB} \times N_{MBS}}{w} - 1))] \times \frac{N_{CB}}{N_{MBS}N_{SM}} \quad \text{Eq. 4-5}$$

4.5.3 Dynamic Partition

In the previous section, we proposed how to estimate each kernel's throughput with the GPU performance model. Next, we will introduce a method that can dynamically partition SMs on the GPU.

Since the launch kernel with SM-ID number is at the head of each kernel. The execution location depends on the SM-ID number. The original idea of forcing part of SMs to process each kernel roughly separates the SMs into two equal parts. Under the same condition, this kind of partition can help GPU get higher performance without any other changes. However, when the two kernels have huge differences in the computational requirement, the defect of using two equal parts of SMs comes out.

For the previous issue, the original method cannot handle the kernels with different capacity requirements. The solution to this issue will be introduced in this section. As we know, the program execution time depends on the computational throughput. If a kernel has a large computational requirement, more computing resources should be signed to the kernel to reduce processing time.

In our experiment, we find that when there are more than two SM subdivisions, the performance drops sharply. Thus, we assume that all programs only have up to two kernels and up to two kinds of kernels in this method. We have used GPU performance model to estimate the kernel's execution time running. With the information collection, we make the partition plan smarter.

We define K_1 and K_2 as two kernel's computational throughput, and SM_1 and SM_2 as the number of each group of SMs. From the *devicequery.cu* GPU information, we can

get the total number of SMs in the current GPU environment. We also define SM_{Total} to be the total number of the SMs in the device. The sum of SM_1 and SM_2 is equal to the SM , as shown the **Eq. 4-6**.

$$SM_{Total} = SM_1 + SM_2 \quad \text{Eq. 4-6}$$

To estimate two kernels' completion time close to each other, the number of SMs for each kernel is based on computational throughput. The relationship between the number of SMs and kernel computational throughput is shown in **Eq. 4-7**.

$$K_1:K_2 = SM_1:SM_2 \quad \text{Eq. 4-7}$$

From **Eq. 4-6** and **Eq. 4-7**, we can derive K_1 and SM_1 a relationship that can be represented in **Eq. 4-8**.

$$\frac{K_1}{K_1 + K_2} = \frac{SM_1}{SM_{Total}} \quad \text{Eq. 4-8}$$

With the two kernel's computational throughput and the number of SM in the current GPU, our algorithm can dynamically choose the number of SM for two SMs groups. The number of SM groups for kernel 1 and kernel 2 are shown in **Eq. 4-9** and **Eq. 4-10**.

$$SM_1 = \frac{K_1 \times SM}{K_1 + K_2} \quad \text{Eq. 4-9}$$

and

$$SM_2 = \frac{K_2 \times SM}{K_1 + K_2} \quad \text{Eq. 4-10}$$

Last, we can estimate the number of SM in the SM group before the kernel is executed in GPU. The dynamic partition mechanism decides the number of SM by the kernel's computational throughput.

4.6 Evaluation

For comparison, we have implemented three GPU computing cases in our experiment. First, two same benchmark kernels are executed with the partition mechanism. Second, two different benchmark kernels have the same computational throughput with dynamic partition. Third, two different benchmark kernels have different computational throughput running with dynamic partition. All three cases are run on three different GPU cards.

4.6.1 Methodology

Our use cases focus on enhancing memory performance and processing performance. We need a set of memory-intensive and computational-intensive programs for the experiment. Meanwhile, for a comprehensive assessment of our technique applicability, the benchmark set should consist of programs of a broad range of domains and have good coverage of both regular and irregular programs. For these reasons, we select nine benchmarks to form our test set. As **Table 4-2** shows, these programs come from four benchmark suites, cover a broad set of domains, and include a similar number of regular and irregular programs.

In the following, we give a simple description of these benchmarks for our experiments. IRREG and NBF are rewritten in CUDA for benchmarks by Han and Tseng (Han & Tseng, 2006). MD and SPMV are both developed by Oak Ridge National Laboratory (Danalis, et al., 2010, March). CFD from Rodinia benchmark suite (Che, et al., 2009, October) simulates fluid dynamics; matrix multiplication and REDUCE are from the CUDA SDK samples (Nvidia, 2019). These applications represent compute-intensive applications that are used widely in parallel applications.

Table 4-2: Description of the benchmark.

Benchmark	Description	Source
irreg	partial diff. solver	Maryland
nbf	force field	Maryland
md	Molecular dynamics	SHOC
spmv	Sparse matrix vector multi	SHOC
cfv	Finite volume solver	Rodinia
nn	Nearest neighbor	Rodinia
pf	Dynamic programming	Rodinia
mm	Dense matrix multiplication	CUDA SDK
reduce	reduction	CUDA SDK

As current GPUs cannot support the two different contexts at the same time yet, we force the two kernels running in the GPU simultaneously in our work. For the evaluation, we designed three experiments each run on three different GPUs.

4.6.2 Machine Environment

We run all the benchmarks on the NVIDIA GTX 650, GTX 970, TESLA M2050 with CUDA 7.5. The host machine is an Intel Xeon E3-1230 CPU and 16GB of memory. The benchmark is run on 64-bit windows 7 ultimate, and each record time is an average of 20 repeated measurements; it includes the overhead time.

4.6.3 Experiment Result

Figure 4-3, 4-4, 4-5 is the speedup of two same kind kernels using a dynamic partition on three GPU. In the benchmark program, there are two kernels in the same function. They have different computational requirements. In **Figure 4-5**, the benchmark is run on the GTX 650. All kinds of the benchmark get a very bad performance. There are two reasons. First, the GTX 650, only has two SMs. There is only one partition plan for executing two kernels at the same time. Each kernel has only one SM for processing. In this case, the two kernels have different computational requirements. The SM for the

kernel with less work must be idle at the end. Many computing resources are wasted when SM is idle. Second, in the GTX 650, there is not enough shared memory for computing. In the co-run condition, there is more than one kernel in the GPU at the same time. When the kernel is memory sensitive, the processing performance will be worse. The GTX 970 and Tesla M2050 have 14 and 13 SMs, respectively. The computational sensitive kernel has a potential speedup shown in **Figures 4-3** and **Figures 4-5** like partial diff and force field, due to this two GPUs having powerful processing capability. For those memory-sensitive benchmarks like matrix multiplication, the nearest neighbor also has a few speedups faster than the two kernels running on serial. Both the GTX 970 and Tesla M2050 do not perform better on the molecular dynamics than on serial. The reason that Tesla M2050 does not get better performance on matrix multiplication is M2050 does not have many processing cores in each SMs. when the kernel is executed in the SM, the computing capability touches the limit of the GPU.

The reason we only run benchmarks on the GTX 970 and Tesla M2050 is that we find GTX 650 only has two SMs, which means it only has one option of partition. So, it is not a suitable device for partition SMs. The results show in **Figures 4-6, 4-7, 4-8,** and **4-9**. In these two cases, we show the speedup of benchmarks before dynamic partition applying and after. The results show our dynamic partition mechanism improves the processing performance. Also, we found that those computational sensitive kernels mixed together can get better performance. Those memory-sensitive benchmarks mix or mixed with computational-sensitive do not get better performance than execution in serial.

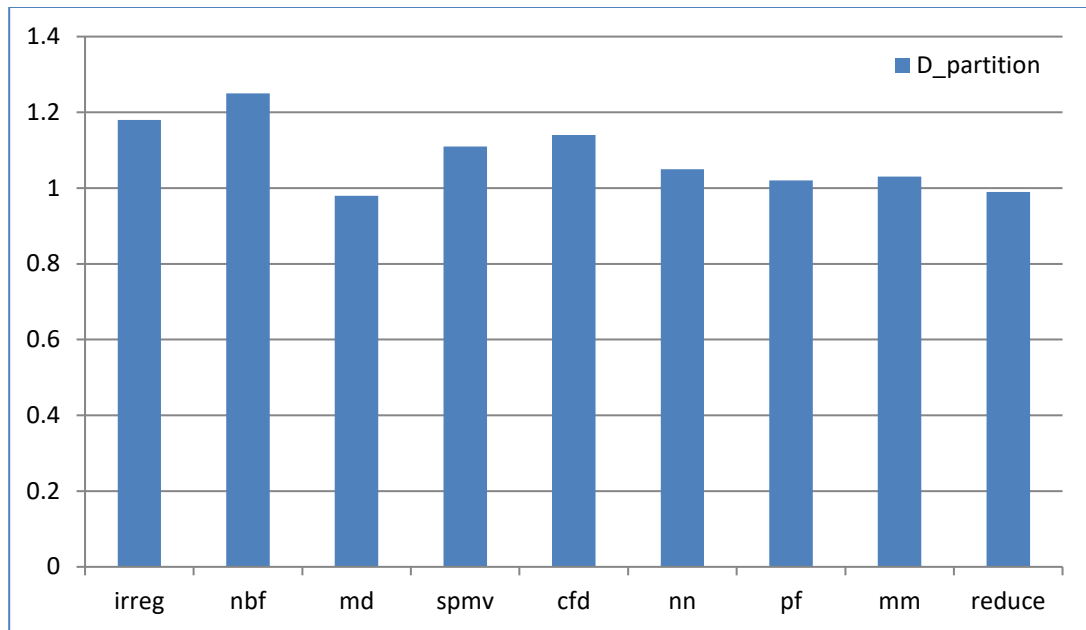


Figure 4-3: Speedup of two same kind kernels using a dynamic partition on GTX970.

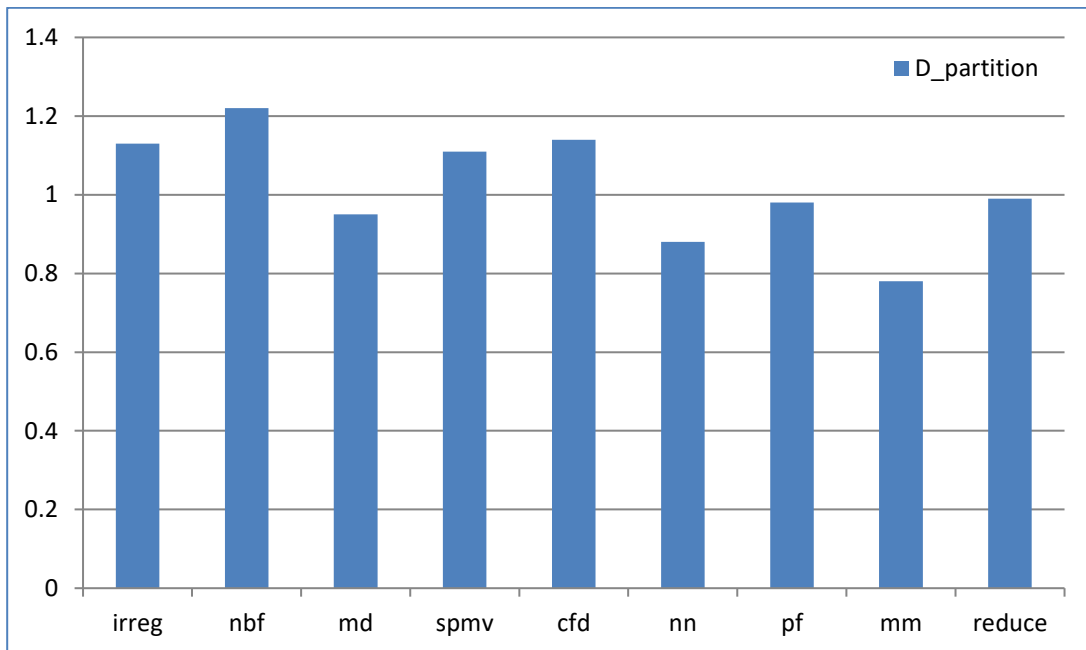


Figure 4-4: Speedup of two same kind kernels using a dynamic partition on Tesla M2050.

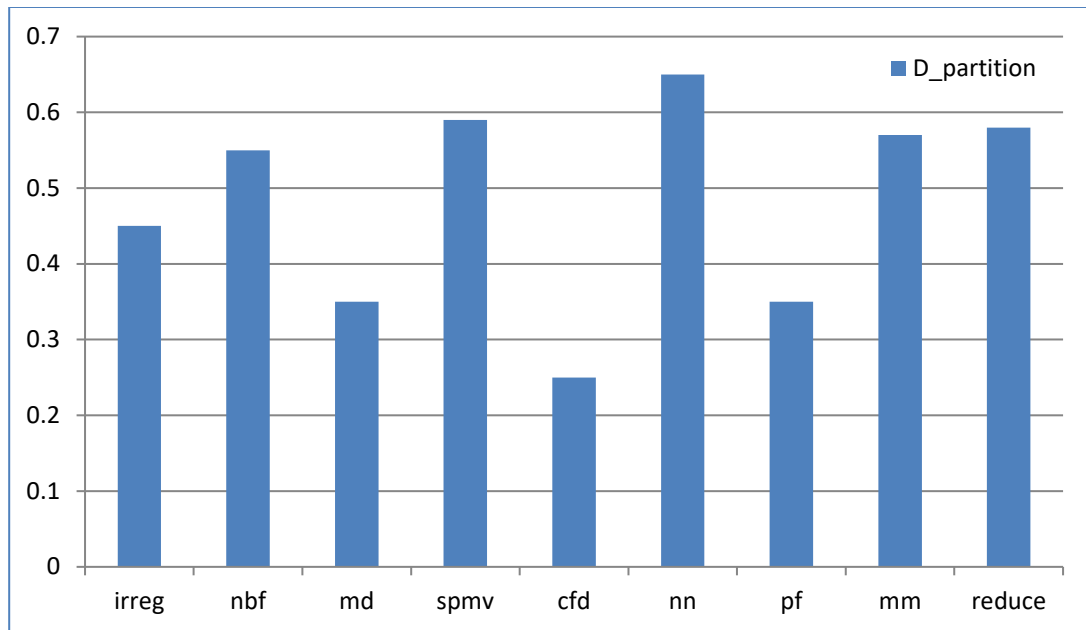


Figure 4-5: Speedup of two same kind kernels using a dynamic partition on GTX650.

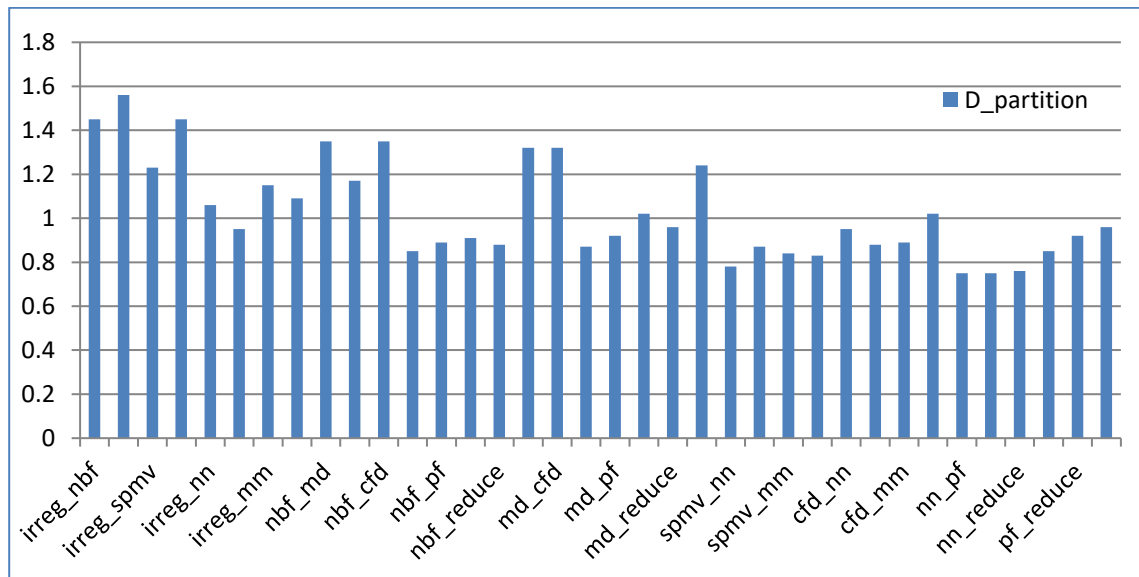


Figure 4-6: Speedup of two different kernels with the same computational throughput using a dynamic partition on GTX 970.

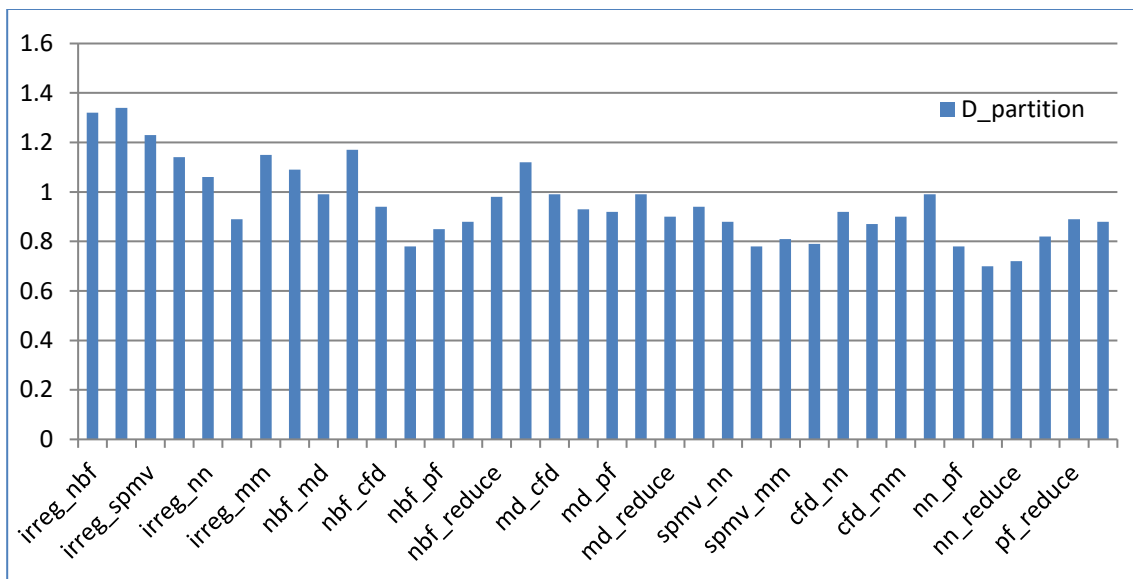


Figure 4-7: Speedup of two different kernels with the same computational throughput using a dynamic partition on Tesla M2050.

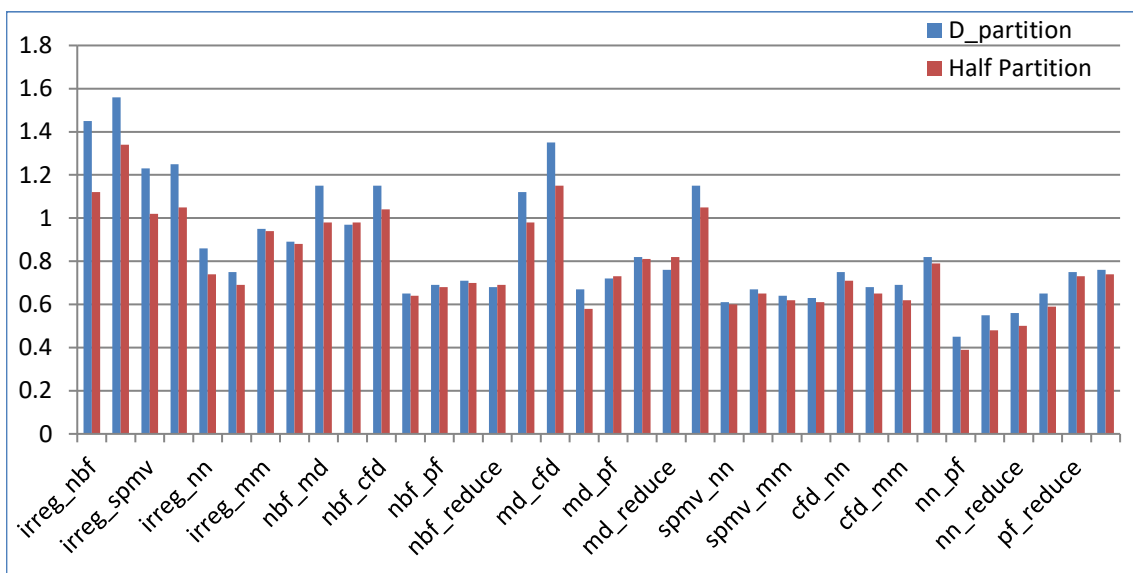


Figure 4-8: Speedup of two different kernels with different computational throughput before using dynamic partition and after on GTX970.

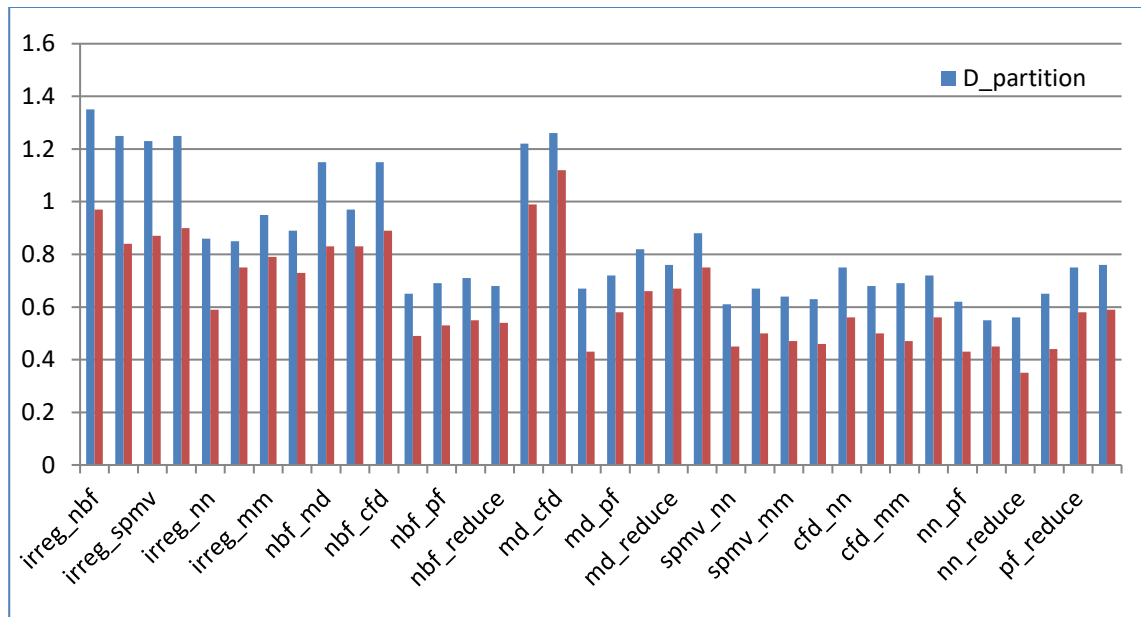


Figure 4-9: Speedup of two different kernels with different computational throughput before using dynamic partition and after on Tesla M2050.

4.6.4 Experiment Conclusion

From the result of all three cases, we find that our dynamic partition mechanism improves co-execution kernels' performance, which is computational-sensitive. However, for those memory-sensitive kernels, although we use our mechanism to enhance the performance of multi-kernel execution, the idea of co-running does not show any improvement on memory-sensitive kernels due to the limitation of memory resources on the hardware.

4.7 Conclusion

This chapter proposed a dynamic partition mechanism for GPU computing, it has broken the limitation of only allowing more than one kernel to be executed in the GPU at the same time. The simple mechanism first time offered the GPU a chance to smartly dynamically partition the SMs into multiple parts. Dynamic partition revealed the

potential of the enabled scheduling control for executions of multi-kernel co-runs.

Dynamic partition opened a new way to optimize the GPU programs without any change on the program. It improves the processing performed on the computational-sensitive program. Our mechanism did not show a significant improvement on those programs requiring large memory space, wide memory bandwidth, and GPU devices with a few SMs.

CHAPTER 5

A PARALLEL COMPUTING FRAMEWORK FOR PERFORMANCE ANALYTICAL MODELS

5.1 Introduction

Parallel processing has long been employed on many large-scale scientific and complex technical computing applications. They are typically resource-intensive applications, such as nuclear simulations, financial stock market analysis, oil exploration, weather forecasts, and simulated reality. During past decades, the emergence of the multi-core and many-core has transformed the parallel computing systems from the national supercomputing center, which only serves selected scientists and engineers, to the modern personal supercomputers for general usage. With dramatic processing power on the recent processor unit, especially the graphics processing unit (GPU), machine learning and deep learning applications have made revolutionary progress. To achieve more efficiency for these parallel programs, performance prediction becomes a burning desire of the parallel computing users to fine-tune their applications. Fortunately, analytical models are widely used to describe performance characteristics. Moreover, many analytical performance models have been recently developed for complex parallel applications such as deep neural networks (DNN), with many layers processed on a heterogeneous system. However, a successful parallel computing analytical performance model is not a silver bullet for general performance prediction. The reason is that the

parallel computing analytical performance model can be quite complex, essentially including several aspects. Unfortunately, with the complex parallel systems, various hardware and software components, it is challenging to develop an accurate analytical performance model for general hardware architecture and software logics. Furthermore, the parallel computing architecture and program continue to evolve drastically. A minor change in the processing unit architecture, interconnection network, or parallel algorithm may require extensive work to adapt to the change. The successful parallel computing analytical performance model must endure and adapt to these conditions. Therefore, a robust framework is a vital requirement as an enabling parallel computing tool and must be flexible to model users' logic on targeted hardware while predicting accurate performance.

In this chapter, we propose a framework for building the parallel computing abstraction models and an analytical performance model. Our framework is aimed to guide users to derive useful information from the hardware architecture and application, then feed those parameters into our two abstract models, which describe the logic of the parallel application and system architecture. Then the users can use the performance metric to evaluate the parallel system and algorithm. Our framework can describe various parallel computing behaviors, such as task executions, data passing, and communication. The contributions of this work include:

- Two parallel computing abstract models are introduced to represent the processing steps and simplify the workload distribution behaviors.
- An extension to Flynn's taxonomy is proposed to support heterogeneous systems and consider the communication time.

This chapter is organized as follows: In 5.2, we discuss related work and list some general issues in building the analytical performance model and the overview of our framework. In 5.3, we introduce the parallel application abstract model to represent the parallel program by dividing the whole task into essential pieces. Also, we propose the computing system abstract model and an extension to Flynn's taxonomy. Finally, section 5.4 is the conclusion.

5.2 Background and Related Work

Parallel computing refers to a process of breaking down larger problems into smaller, independent, and similar parts that can be processed simultaneously by multiple processors, the results of which are combined upon completion as part of an overall algorithm. The main goal of using parallel computing is to increase the efficiency of current computing capacity for faster task processing and thus to speed up the performance. The parallel computing paradigm typically presents as distributing the tasks to multiple partitions on many processing units and collecting the results from each processor unit.

In recent years, parallel computing has become increasingly popular to solve problems such as machine learning. AI researchers have proposed new algorithms and solutions such as Convolution Neural Network (LeCun, et al., 1989), unique resource management, modern communication methods such as NCCL (Luehr, 2016), Ring Allreduce (Sergeev & Del Balso, 2018), PS (Cui, et al., 2016, April), and new hardware to speed up the computation. However, despite these significant advances, it is not easy to analyze and optimize performance for these applications. Thus, it is vital to find an appropriate tool or method such as appropriate analytical modeling to disclose the

abstract views of hardware and software components. Building the analytical performance model is a common method in the performance evaluation for parallel computing. During the past years, researchers have developed analytical performance models based on specific hardware, such as a cluster of CPUs, and recently GPU, for various parallel applications. These models can be outdated quickly, especially when there are rapid changes in hardware and parallel system.

There are three kinds of performance evaluation techniques: analytical modeling, simulation modeling, and measurement. Analytical approaches are less accurate than simulation approaches; they are also simpler and quicker to provide insights since the parallel computing behaviors are described through mathematical equations. Moreover, analytical modeling provides an abstract view of hardware and software. Parallel performance analytical models based on system parameters, like LogP (Culler, et al., 1993, July), LogGP (Alexandrov, et al., 1995, July), are widely used to evaluate parallel applications. The performance model enables users to understand the behavior of the applications, supports users to make decisions during the execution. The model can also represent the composition of the program logic, which means the ratio of parallelism among subtasks to the total tasks. Thus, we can estimate the execution time of the applications, potentially identify the performance bottleneck and scalability of the system before we run the program on the target machine.

Constructing an appropriate analytical model is quite useful and helps us break down a complex problem into more manageable pieces. The typical model consists of two parts. First, it must well describe system characteristics and should be as accurate as possible. Second, it must be as simple as possible to represent the problems. The first

feature requires that the model includes all the necessary details that define the system characteristics and the second implies that the model should be in a simple expression. These two features contradict each other. It is a dilemma facing the modeling practitioners. Including too many parameters may help increase the accuracy of the model, the following issue is model getting too complex to be solved. An overly complicated model takes too much time to be solved or is even too complex to do so. Thus, an unsolvable model is completely useless. Hence, care must be taken in selecting parameters, and a reasonable trade-off should be made for an appropriate performance model.

Traditionally, the analytical performance model is created and based on either deterministic analysis, probabilistic analysis, or benchmark. The deterministic analysis involves only the summation of all tasks in the sequential work, and also has considered synchronization cost. The overhead of the deterministic analysis is manageable, and therefore the model could be built in a short time. On the other hand, a probabilistic analysis starts from an assumption about a probabilistic distribution of the set of all possible inputs. This assumption has a lot of uncertain information. Although benchmark suits most of the conditions no matter how complicated they are, and micro-benchmark helps to reduce the evaluation time. It is still needed to implement on the target machine and collect the processing results. Consequently, for a particular application and machine type, users need to find or build a suitable performance model for them.

There has been numerous parallel architectures and their implementations. Flynn's taxonomy (Flynn, 1972) is a general classification that describes computing

architectures and paradigms by considering various instruction and data streams that can be processed simultaneously.

One of the most critical evaluations in parallel computing is to measure how much faster a parallel task can run with respect to the best possible sequential one. This measure is known as speedup. To evaluate the performance of the parallel application, Amdahl's law and Gustafson's law are widely used for measuring speedup.

Our modeling and evaluation framework is an extension of Flynn's taxonomy and a combined Amdahl's law (Amdahl, 1967, April) and Gustafson's law (Gustafson, 1988) to create a novel model with familiar taxonomy and performance metrics. Our proposed framework offers a simple way to build an analytical performance model which adapts to modern hardware and applications. The framework allows an easy way to parametrize both computational logic and various hardware architecture that are well-suited for general parallel applications in practice.

5.3 Framework for Parallel Application Analytical Modeling

Our framework provides a workflow for users who wish to build their analytical performance model based on parallel computing. We consider a general parallel computing system of which it is partitioned into a collection of nodes, and the node may consist of multiple CPUs and perhaps coprocessing units such as GPGPUs. Our framework describes the parallel application and targeted hardware architecture in the two abstract models. These two models enable users to build their analytical performance model for specific hardware and application logic. The framework and workflow are shown in **Figure5-1**.

The analytical performance model will help users to identify possible bottlenecks and improve their parallel program performance. In our framework, the parallel application abstract model describes the logic of the parallel application. The model represents breakdowns of the whole program into many simple pieces that can be evaluated and summed up for the estimated completion time. The parallel system abstract model helps to guide users to estimate each part's computing time and communication time. The performance metric assists users in evaluating and comparing the application performance before and after optimization.

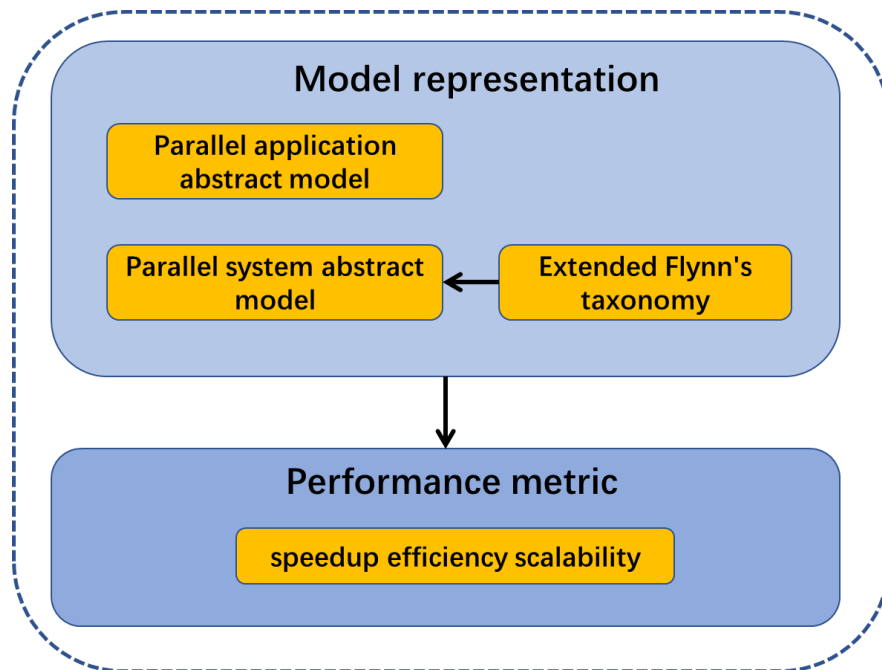


Figure 5-1: Overview of the parallel computing performance modeling framework.

5.3.1 Parallel Application Abstract Model

The parallel application abstract model is a representation of internal application structures that allow individuals to describe the application logic. This model simplifies parallel computing representation into a collection of computational subtasks. The model

defines how many subsets or the computing tasks are in the whole application. Computing tasks could be parallel or serial. The application abstraction model is guidance for building the analytical model for parallel computing. There can be various kinds of computing in parallel applications: parallel, serial, consecutive serials in parallel, and numerous parallels in serial, as shown in the example of four task types in **Figure 5-2**. The dark blue box represents the tasks that could be executed in sequential, and the green box represents the parallel tasks. Each type has a time estimation method. With the parallel application abstract model, users break down the parallel application into subtasks and can describe their parallel algorithms in more generic ways.

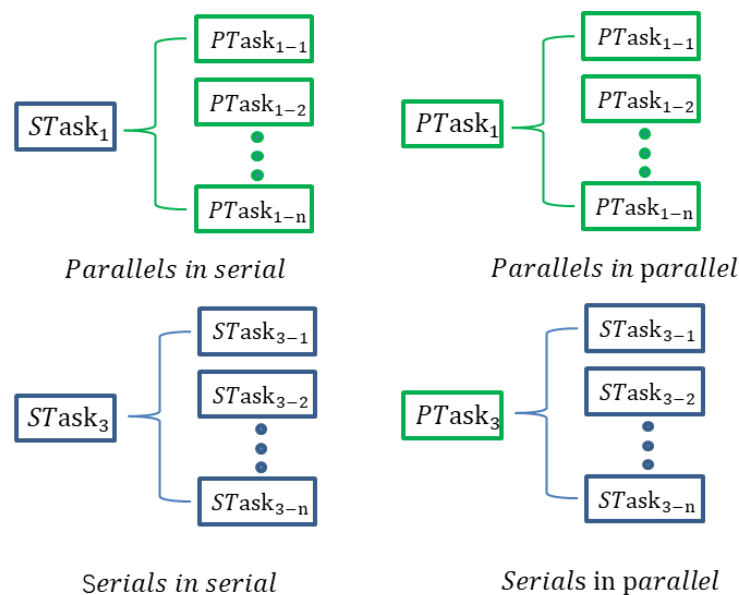


Figure 5-2: Examples of the four types of tasks.

In the parallel program, there may be many sub-tasks including parallel tasks or sequential tasks, sometimes even the mixture between parallel and sequential ones. The total execution time is the summation of the parallel tasks and the sequential tasks. In this abstract model, we introduce two parameters. First, the execution time of sequential

tasks. Second is the execution time of parallel tasks. **Figure 5-3** depicts an example of the application abstractions. From left to right, in layer 1, there are a sequence of tasks, and each box represents an independent task. The individual task may consist of parallel or sequential subtasks. The dark blue box means the tasks could be executed in sequential. The green box represents the parallel tasks. The next layer represents subtasks of the previous layer task. This representation provides recursiveness on how tasks can have several subtasks and generality of real-world applications.

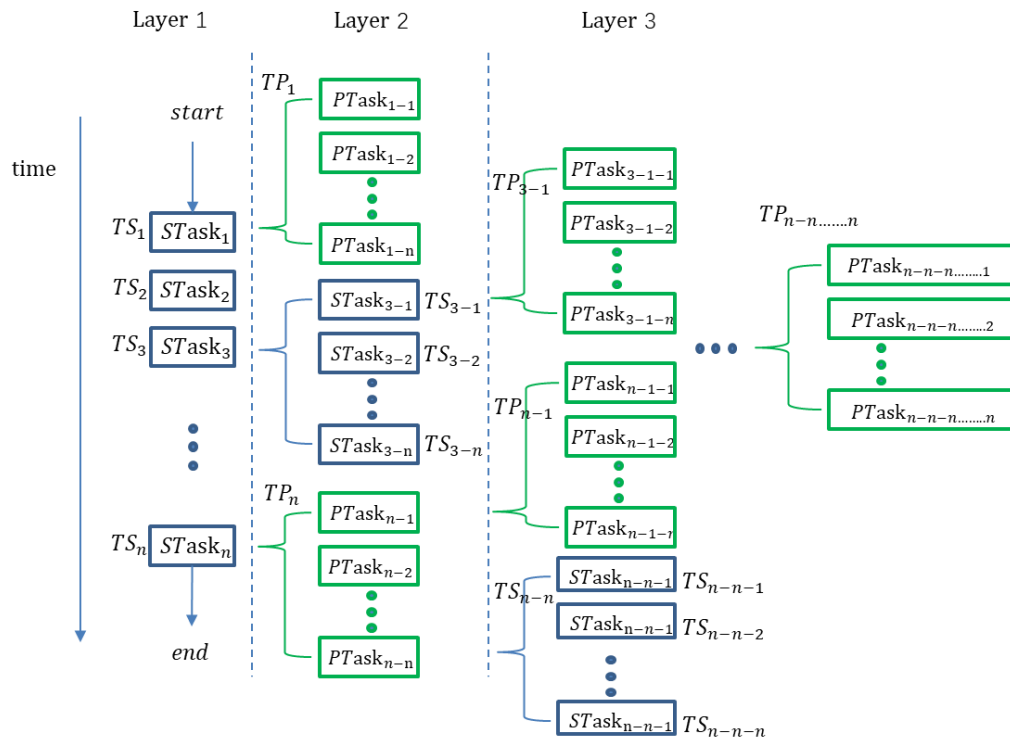


Figure 5-3: Application logic example.

Several parameters will be introduced in our framework before diving into the abstract model. Those parameters are shown in the following **Table 5-1**.

Table 5-1: Parameters for the Parallel application and system abstract model.

Notation	Description
Application abstract model	
TS	The execution time of a subset sequential task
TP	The execution time of a subset parallel task
V_D	The volume of transfer data
V_I	Volume of Instruction transportation
V_R	Volume of result transportation
N_I	Number of instructions
N_{Dmsg}	Number of times data transfer
N_{Imsg}	Number of times instruction transfer
CPI	Cycles per instruction
System abstract model	
L_D	The latency of data transfer
L_I	The latency of instruction transfer
N_P	Number of the processor unit
B	The total bandwidth of the system

The total execution time depends on the number of individual tasks as shown in **Figure 5-3** layer1. The total execution time is the sum of the independent tasks as shown in **Eq. 5-1**.

$$T_{total} = TS_1 + TS_2 + \dots TS_n \quad \text{Eq. 5-1}$$

Some tasks may consist of a collection of parallelized tasks or sequential tasks. For example, in **Figure 5.3**, $STask_1$ consists of parallel subtasks, $PTask_{1-1}$, $PTask_{1-2}$ to $PTask_{1-n}$ and the time of the $STask_1$ is represented by TP_1 . All these

parallel tasks could be executed at the same time. The execution time of the sub-tasks TS_1 is equal to TP_1 . TP_1 is derived from the longest execution time in parallelable subtask of $STask_1$ as shown in **Eq. 5-2b**

$$TS_1 = TP_1 \quad \text{Eq. 5-2a}$$

and

$$TP_1 = \max_{1 \leq n} \{TP_{1-n}\} \quad \text{Eq. 5-2b}$$

In another situation, for example. some tasks like $STask_3$ may consist of multiple sequential tasks: $STask_{3-1}$, $STask_{3-2}$ to $STask_{3-n}$. TS_{3-1} , TS_{3-2} ... TS_{3-n} represent the execution time of each sub-task in the $STask_3$, and the TS_3 is equal to the summation of all sub-tasks' execution times in the $STask_3$ as shown in **Eq. 5-3**.

$$TS_3 = TS_{3-1} + TS_{3-2} + \dots + TS_{3-n} \quad \text{Eq. 5-3}$$

TS_{3-1} is the subset of $STask_3$ which consist of multiple parallel tasks: $PTask_{3-1-1}$, $PTask_{3-1-2}$ to $PTask_{3-1-n}$. PS_{3-1} , PS_{3-2} ... PS_{3-n} represent each sub-task in the $STask_{3-1}$ and the TS_3 is equal to the time of processing all parallel sub-tasks in the $STask_3$ as shown in **Eq. 5-4**.

$$TS_{3-1} = TP_{3-1} \quad \text{Eq. 5-4}$$

In sub-task like $PTask_{n-1}$, it has multiple sub-tasks, which could be either parallel tasks or sequential tasks. The parallel task execution time depends on the longest job. The sequential task execution time is the summation of all sequential task execution time. The whole application execution time is the longest processing time of parallel tasks by adding the total execution time of the sequential tasks. The maximum processing time of the parallel task is shown in **Eq. 5-5**.

$$PTask_{n-1} = \max_{1 \leq j \leq n} \{TP_{n-j}\} \quad \text{Eq. 5-5}$$

In the sub-task $PTask_{n-n}$, it consists of multiple sequential tasks $STask_{n-n-1}, STask_{n-n-2}, \dots, STask_{n-n-n}$, the time of $PTask_{n-n}$ is the summation of all sequential tasks.

$$Tp_{n-n} = TS_{n-n-1} + TS_{n-n-2} + \dots + TS_{n-n-n} \quad \text{Eq. 5-6}$$

and

$$Tp_{n-n} = \sum_{i=1}^n TS_{n-n-i} \quad \text{Eq. 5-7}$$

The total execution time of the whole application T_{total} is the summation time of all sub-tasks.

$$T_{total} = \sum_{i=1}^I \left[TS_i + \max_{1 \leq j \leq n} \{TP_{n-j}\} \right] \quad \text{Eq. 5-8}$$

5.3.2 Parallel System Abstract Model

The parallel computing system abstract model is a representation of the system hardware architecture. The model represents important hardware aspects, such as processing capability and their connectivity, and capturing the computing and communication times. In the parallel system, there can be more than one processing unit to execute parallel tasks. Normally, several processing units are grouped into a node. The switch or network fabric connects nodes to form a larger computational capability. In this model, we denote the symbol B to represent the total bandwidth of the node connectivity. In the beginning, data will be loaded to the processing units, which could be from local storage, system memory, or network storage. At the abstraction level, we signify V_D to represent the volume of the total data transfer to the processing unit. All-important

parameters in this model are shown in **Table 5-1**. Our abstract model defines the total execution time into two parts. The first is computing time. The second is communication time. Each part could be defined as a building block based on various computing paradigms and architecture which are defined by our proposed extended Flynn's taxonomy. We have enhanced Flynn's taxonomy that includes the heterogeneous computing paradigm.

We consider the total time of the application with both the computing and communication time. The communication time consists of data loading, instructions passing, and result collection. The execution time represents the time from the processing unit received the data, computation, and instructions to send out the results.

$$T_P = T_{LD} + T_I + T_C + T_{Coll} \quad \text{Eq. 5-9}$$

Eq. 5-9 defines the T_P as the total time of parallel computing. It is a sum of the data loading time (T_{LD}), time for instructions passing (T_I), execution time (T_C) and time of results collection (T_{Coll}).

T_{LD} Loading Data

Parallel computing loading data time T_{LD} is the time representing how long it takes to perform the data transfer. Data transfer overhead depends on the size of the data and the bandwidth of the system. However, before each message is sent or received, there is a latency that needs to be considered. The latency depends on the interconnection fabric, computation protocol, and the times of message sending/receiving. Through the data loading analysis, we can calculate the overhead based on both communication time and latency. The total time of data loading is represented by the following function fT_{LD} .

$$fT_{LD} = F(\text{volume of Data, Bandwidth of system, latency, times of sending message})$$

$$T_{LD} = \frac{V_D}{B} + L_D \times N_{Dmsg} \quad \text{Eq. 5-10}$$

In **Eq. 5-10**, B is the Bandwidth of the system, L_D is the latency of each data transfer, and N_{Dmsg} is the number of times data transfers. The data loading time T_{LD} equals to the sums of the data transfer time with the latency of each data transfer.

In reality, V_D also depends on the communication method, such as data from one point to multiple points or multiple points to various points. Some of the messages will be passed multiple times when they need to be distributed to multiple nodes. The hardware's characteristics dictate the bandwidth of the system and capacity. Once the system sends too many messages at the same time, it may reach the bandwidth limitation. The system will hold some messages for a while. As such, the latency needs to be considered for estimating the data transfer time.

T_I Instructions Passing

Same as the data loading, Instructions can also be transferred as messages. The instruction passing time T_I is the time that represents how long it takes for dispatching instructions to processing units. The time of message sending depends on the volume of the instructions and the bandwidth of the system. However, before each message is sent, there is a latency that must be considered the same as data loading. This latency can be derived from the system hardware specification and the times of message passing. The total time of instructions passing is represented by the following function fT_I .

$fT_I = F(\text{volume of instructions, Bandwidth of system, latency, times of sending message})$

$$T_I = \frac{V_I}{B} + L_I \times N_{Imsg} \quad \text{Eq. 5-11}$$

In **Eq. 5-11**, T_I is the volume of Instruction transportation, B is the bandwidth of the system, L_I is the latency of each data transfer, and N_{Imsg} is the number of times instruction transfer. The time of instructions passing T_I equals to sums of the data transfer times with the latency of each data transfer.

T_C Execution Time

Parallel computing execution time is the time that elapses from the moment that data and instructions are received to the moment the task is completed (including, perhaps, sending the results). Execution time depends on the computing characteristic and the type of parallel computing algorithm, the number of processing units, and the processing unit's architecture. In the next section, we will introduce extended Flynn's Taxonomy. We propose a new classification which is based on the heterogeneous show in **Figure 5-4e**. The overview of extended Flynn's Taxonomy is shown in **Figure 5-4**.

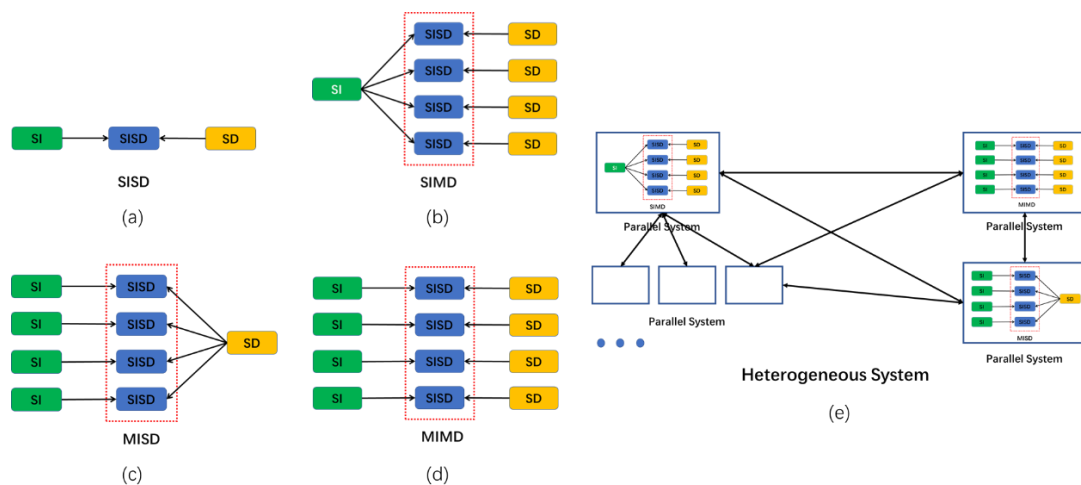


Figure 5-4: Overview of the extended Flynn's Taxonomy.

T_{Coll} Result Collection Time

The result collection time is the time that system transfers the results from each processing unit to the host. We assume that all processor units will pass the results

simultaneously and share the system bandwidth. The following function fT_{Coll} represents the total time of result collection.

$$fT_{Coll} = F(\text{Size of Data, Type of data, communication algorithm})$$

$$T_{Coll} = \frac{V_R}{B} \quad \text{Eq. 5-12}$$

In **Eq. 5-12**, V_R is the volume of results, and B is the bandwidth of the system.

The time of result collection is equal to the volume of results divided by the bandwidth of the system.

In reality, coalesced memory accesses (Che, et al., 2011, November) are much faster than uncoalesced memory accesses; when some data is stored in uncoalesced memory, the access time will be longer than the data stored in the coalesced memory. A memory access speed estimation model needed to be introduced for a different type of data.

5.3.3 Extended Flynn's Taxonomy

Computer architectures and computing paradigms can be classified by Flynn's taxonomy which represents computing into four categories. This classification depends on two aspects; first, the number of instruction streams, second, the number of data streams that can be handled in parallel. In this work, we propose one more category which represents heterogeneous computing (Khokhar, et al., 1993) (Mittal & Vetter, 2015). In the modern parallel computing system, there are systems with multiple-core CPUs coupled with a GPU that support SIMD instructions. When the SIMD processor cooperates with a traditional processor, the system is considered a heterogeneous parallel system. The following sections detail four classifications and the extended Flynn's taxonomy.

Single Instruction Stream, Single Data Stream (SISD)

A sequential computer exploits no parallelism in either the instruction or data streams. A single control unit loads a single instruction from memory. The control unit then generates appropriate control signals to direct a single processing element to operate on a single data stream. The following diagram shows the SISD is a single instruction with a single data stream. The SISD is the traditional computational model with a single core.

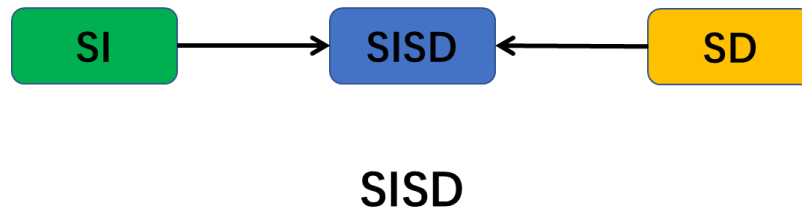


Figure 5-5: Single instruction multiple data streams.

Single Instruction Stream, Multiple Data Streams (SIMD)

A single instruction operates on multiple different data streams. We assume that multiple SISD operations only use a single instruction to process multiple data. All these same instructions can be executed in parallel with a different set of data, such as in parallel by multiple functional units like in the GPU computing system.

Single instruction multiple threads (SIMT) are an execution model used in parallel computing where single instruction and multiple data (SIMD) are combined with multithreading.

$$T_{csimd} = \max_{1 \leq i \leq n} \{TP_i\} \quad \text{Eq. 5-13}$$

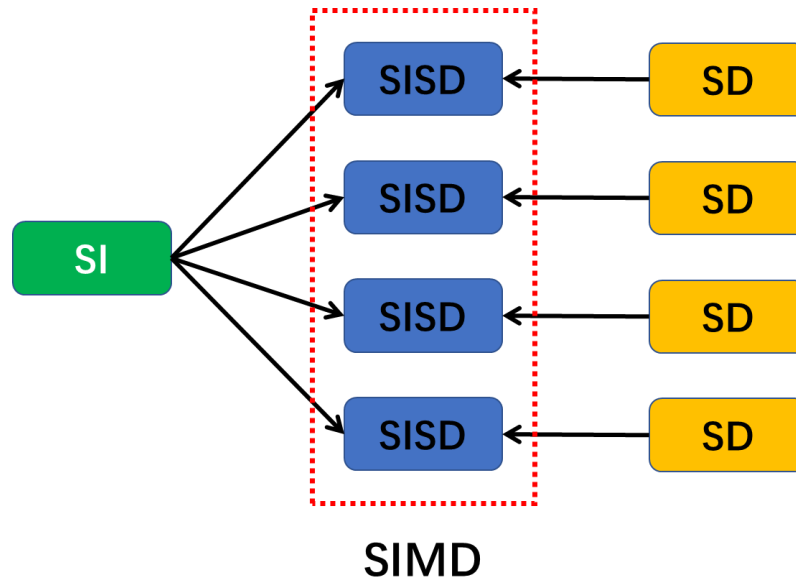


Figure 5-6: Multiple instruction single data stream.

Multiple Instruction Streams, Single Data Stream (MISD)

Multiple instructions operate on one data stream. This is an uncommon architecture that is normally used for fault tolerance, and heterogeneous systems operate on the same data stream. For example, the Space Shuttle flight control computer is using MISD for data processing.

$$T_{cmisd} = \max_{1 \leq i \leq n} \{TP_i\}$$

Eq. 5-14

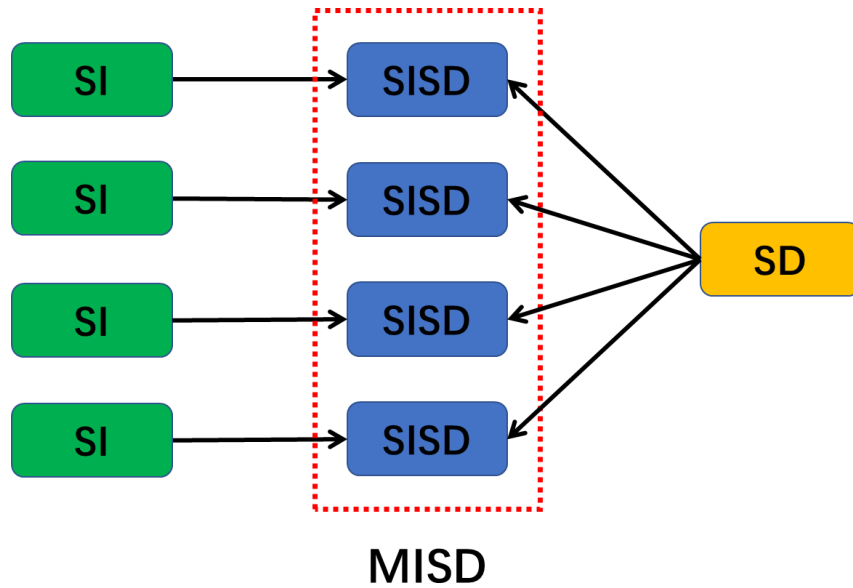


Figure 5-7: Multiple instruction single data streams.

Multiple Instruction Streams, Multiple Data Streams (MIMD)

Multiple autonomous processors execute different instructions on different data simultaneously. The MIMD architecture includes individual multicore processors and distributed systems with shared memory space or distributed memory space. The processors in the MIMD system operate independently and asynchronously:

$$T_{cmimd} = \max_{1 \leq i \leq n} \{TP_i\} \quad \text{Eq. 5-15}$$

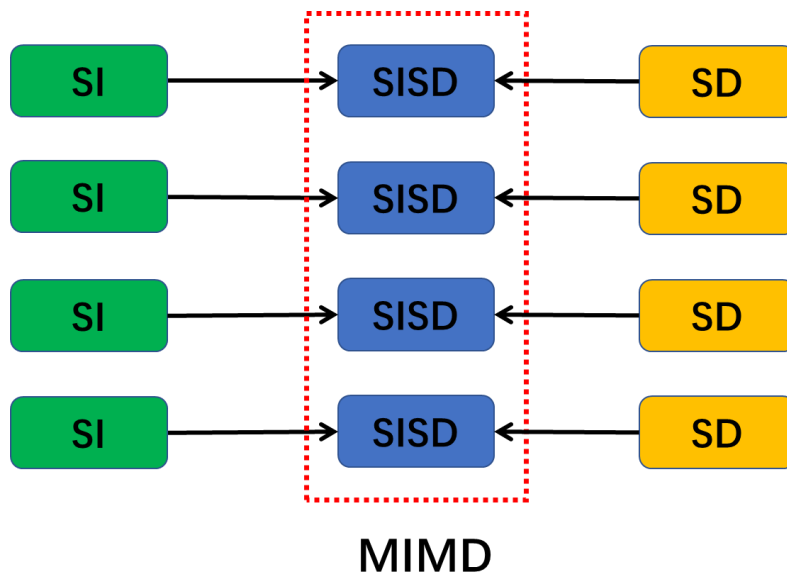


Figure 5-8: Multiple instructions multiple data streams.

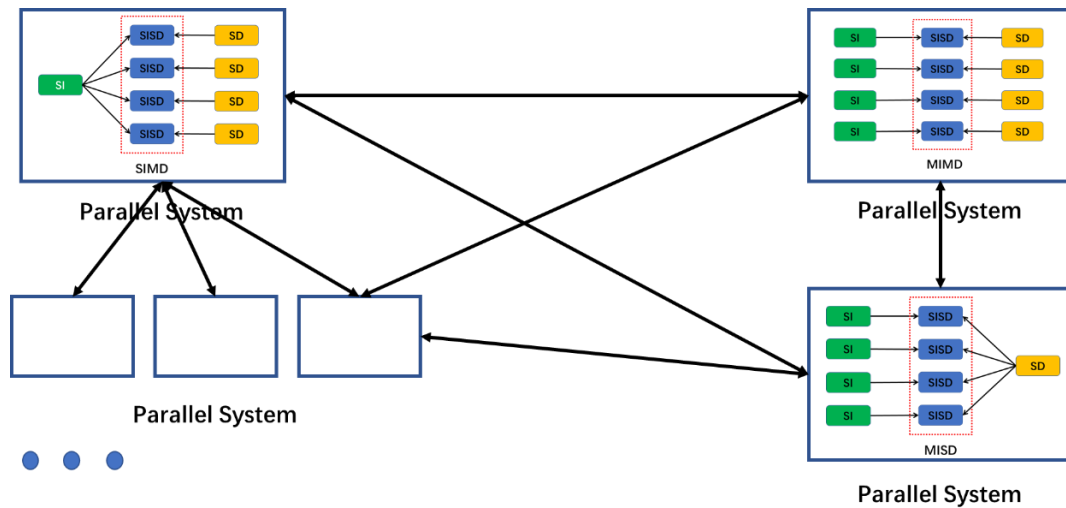
Heterogeneous Computing

Heterogeneous computing is a computational paradigm with more than one kind of processing unit or cores. These systems gain performance or energy efficiency not just by adding the same type of processors but by adding different co-processors, usually incorporating specialized processing capabilities to handle particular tasks. In modern parallel computing, host processing units work with cooperation processing units. The host is usually a CPU, and the cooperation processing units are many-core processing units like GPU and a field-programmable gate array (FPGA). In this system, there are many types of computing, like SISD, MIMD, and SIMD, as shown in **Figure 5-4**.

Traditional four class taxonomy has difficulty describing modern systems which including multiple types of computing.

In a heterogeneous system, the communication between the host and coprocessors is the most critical factor for the system's performance. In reality, users have to consider the thread synchronization, round-trip data transfer overhead between CPU and GPU, and

carefully design the communication algorithm. A performance model that includes the communication between different heterogeneous cores is necessary for heterogeneous system performance estimation.



Heterogeneous System

Figure 5-9: Heterogeneous system.

5.4 Conclusion

This chapter has systematically introduced the framework for building the parallel analytical performance model. We proposed the parallel application abstract model and parallel computing system abstract model. In addition, we introduced the extended Flynn's Taxonomy, which included heterogeneous computing. Today's parallel computing requires programmers to manually optimize the application performance when it is deployed on the new hardware. We anticipate that the proposed framework will enable users who want to create an analytical performance model to further enhance their parallel application performance.

CHAPTER 6

PERFORMANCE MODEL FOR CNN ON DISTRIBUTED GPU SYSTEM

6.1 Introduction

Deep neural networks (DNNs) have been very successful in various machine learning tasks, such as visual recognition (Krizhevsky, et al., 2012), speech recognition (Han, et al., 2017, February), and machine translation (Wu, et al., 2016). Among those applications, the convolutional neural network (CNN) proposed by LeCun (LeCun, et al., 1989) was one of the earliest successful DNN models that were used to classify images. CNN models equipped with deep learning techniques outperform previous machine learning techniques in various visual recognition challenges, such as ILSVRC (ILSVRC, 2020) and PASCAL (PASCAL, 2020). These neural networks use the larger data set and deeper neural network layers to train high accuracy models. These challenges require large-scale training and advanced computation. Fortunately, GPUs have increasingly become widely used in accelerating parallel computing applications due to their cost-effectiveness and recent advancements. Thus, GPU and GPU clusters have been employed in the training of the neural network and resulted in the aforementioned successful applications.

The crucial factors which affect the performance or training time of neural networks mainly include three parts. First, the structure of neural networks determines the

amount of computation, which directly affects time consumption. Second, the performance and efficiency of targeted hardware are key for training neural networks. Third, the selection of training algorithms determines the training processes of neural networks and their completion time. In the training algorithms, the communication algorithm is also one of the most influential factors.

With these crucial factors which affect the performance of training neural networks, researchers have studied and proposed many methods to reduce the training time. One approach to speed up CNNs is to reduce the time complexity of convolution algorithms. Fast Fourier Transform (FFT) algorithms (Nussbaumer, 1981), and Winograd's minimal filtering algorithm (Winograd, 1980) are popular methods and successfully reduce the algorithm complexity of the convolution computation in a CNN. GPU and GPU clusters are also introduced to accelerate the training speed. In multi-GPU training scenarios, data parallel and model parallel are used as the method to divide the whole task into multiple parts for parallel computations.

To evaluate the training performance of neural networks, users mainly rely on public benchmarks or numerous experiments to obtain the run-time, which brings a significant wastage of time and resources. To eliminate this problem, many performance models have emerged. However, an existing analytical performance model is not a silver bullet for neural network training performance prediction. The reason is that the neural network can be quite complex, essentially including several aspects and running on various kinds of hardware. Unfortunately, with the complex network, various hardware, and software components, it is challenging to develop an accurate analytical performance model for neural networks. A minor change in the neural layer, interconnection network,

or software environment may require extensive reworks to adapt to the change. In addition, the general performance model must be modified by highly skilled individuals when algorithm and hardware architecture are changed. To achieve more efficiency in training a neural network, performance prediction becomes a burning desire of performance-tuning.

The previous chapter proposed the parallel computing performance analytical model framework to aid practitioners who wish to predict performance and fine-tune their parallel computing applications. Our goal is to simplify ways to create the models and yet to represent the problems in wider domains. We extended the classic computational modeling of Flynn's taxonomy and combined ideas of two popular performance measurement methods, namely Amdahl's and Gustafson's laws.

Therefore, this chapter aims to validate our framework by demonstrating it with popular parallel computing applications such as CNN on a distributed GPU system. First, for the *Parallel application abstract model*, we separate CNN model network layers into multiple independent tasks. Second, by *Parallel system abstract model*, we use the hardware characteristic of targeted GPUs, the framework library chosen by programmers, and the communication method between multiple GPUs to determine the running time for each part. Also, with the *Extended Flynn's Taxonomy*, GPU cluster and heterogeneous system performance become countable. We validate our performance analysis framework with four popular CNN models AlexNet (Krizhevsky, et al., 2012), VGG (Simonyan & Zisserman, 2014), GoogLeNet (Szegedy, et al., 2015), and ResNet (He, et al., 2016) executed on two NVIDIA Pascal GPUs GTX 1080 and GTX 1080Ti, and show that our performance analysis framework is both accurate and robust across the diverse layers

memory accessing and library framework. We also demonstrate how our performance analysis framework can be used for the design-space exploration of future GPUs and identify interesting tradeoffs for efficient CNN execution by independently scaling different GPU resources.

In summary, our main contributions are:

- We present a comprehensive performance analysis model that can predict performance and understand bottlenecks for CNN on GPU.
- We analyze the optimizable method for CNN on multiple GPUs, which will help users evaluate their techniques before running on targeted machines/architecture.
- We validate the performance analysis model's accuracy and robustness across four popular CNNs on GPUs.
- We demonstrate how a performance analysis framework can efficiently explore the potential optimizable part of the CNN algorithm and the bottlenecks of current CNN.

6.2 Background

6.2.1 Convolutional Neural Network (CNN)

CNN is a type of deep learning technique that is commonly used for image analysis. The training process of CNNs is a feed-forward neural network, which means using the Backpropagation algorithm to adjust learnable kernels, thereby minimizing the cost function. The convolutional neural network uses a local receptive field, shared weight, and pooling to automatically provides some degree of shift and distortion invariance. The convolutional layer is the central part of CNNs. In the convolutional

layer, the neuron which in the same feature map using the same weight for input data to get the corresponding features. The shared weights mean that neurons share the same weights in the feature map. In the current layer, each neuron is connected to the previous layer. This kind of connectivity is called the local receptive field.

We are using Lenet, which was proposed by LeCun in 1989 to show the architecture of CNNs. The architecture as shown in **Figure 6-1**. Lenet-5 (LeCun, et al., 1998) includes a convolutional layer, a pooling layer, and two fully connected layers. The input images are first to get into the input layer and then the data from the previous layer pass to the next layer, like the convolutional and pooling layer.

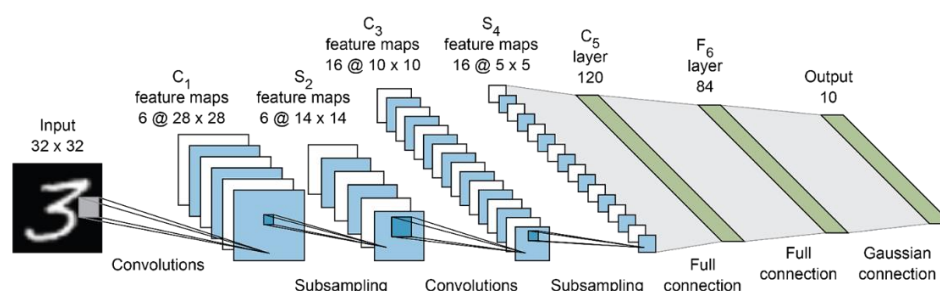


Figure 6-1: The architecture of LeNet-5.

6.2.2 Graphic Processing Unit (GPU) Architectures

Modern GPUs are designed for compute-intensive applications. With the development of GPU performance, deep neural network CNN gets a huge advantage of GPU performance. During the network training, it is essential to understand their general data structures and computation algorithm. Let us consider NVIDIA GPUs as an example. The GPU consists of many types of memory and many streaming multiprocessors (SMs). Each SM contains a variety of functional units. There is also a small size of low latency shared memory for SM, which the programmer can allocate the

memory. In the GPU, the smallest processing unit is warp, which contains 32 threads and is scheduled by the task scheduler in the current generation GPU. Multiple warps in the same block were executed by the same SM. The number of threads in each block and the number of active blocks in each SM are determined by the hardware specification and programmer setting. More details could be found in chapter 3.

6.2.3 CNN Training Process

The training process of neural networks depends on the error backpropagation algorithm. The training process involves a huge size of calculation and data transmission. CNN's main computing operations, convolution on GPU can be executed by the libraries such as CUBLAS (Nvidia, 2020), which is a matrix operation library of NVIDIA. The CUBLAS supports various operations based on General Matrix-to-matrix Multiply (GEMM). The most common parallelization strategy is data parallelism, which places the entire neural network copies on each device so that each processor group processes a subset of the training data with the whole neural network and synchronizes network parameters at the end of each iteration.

Another common parallelization strategy is model parallelism. Programmers assign subsets of a neural network to many devices; each device has a part of the neural network, and the training processing is like pipeline processing at the first iteration. After the first iteration, all parts can be trained parallelly. This approach does not need parameter synchronization between devices but requires data transfers between each device.

Parallelized CNN training can be executed on multiple GPUs. However, we only consider data parallel, which is most widely used, we use data parallel to demonstrate our

framework in this paper. Data parallel means that the training data set are divided and distributed on different computing devices, in each device, there is the same copy of the model or neural network. There are two common implementations of data transfer strategies in this process. One is parameter server (PS) mode as shown in **Figure 6-2**. In the parameter server mode, the CPU is usually used as a server node. Another mode is NCCL. The model also supported by NVIDIA collective multi-GPU communication library (NCCL) as shown in **Figure 6-3**, realizes parameter transfer and computing through the All Reduce Kernel function, which does not need CPU and the transmission bottleneck depends on the slowest network link.

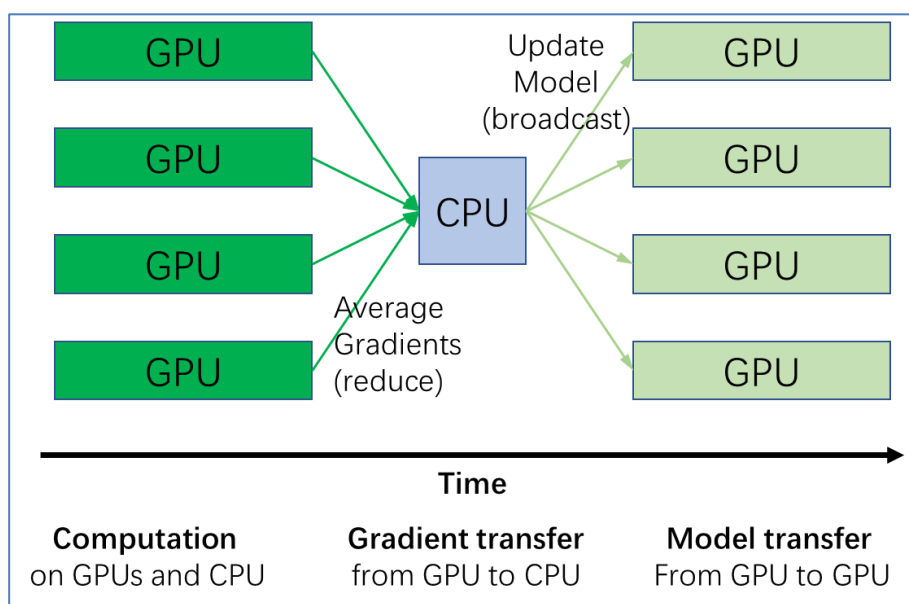


Figure 6-2: Parameter server (PS).

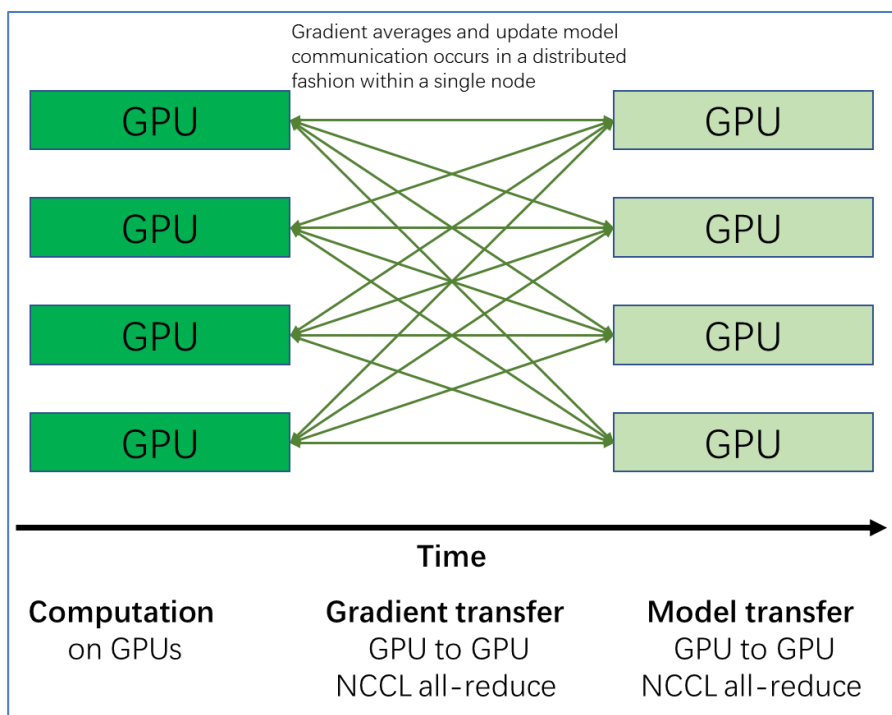


Figure 6-3: The NVIDIA Collective Communication Library (NCCL).

6.2.4 Programming CNN to GPU

As we know, the convolution operation is the most time-consuming part of the convolutional neural network training. The convolution can be easily mapped to the GPU in multiple ways and take advantage of GPU parallel computing performance.

Direct Convolution is the traditional way of processing convolution. A small window slides within an input feature map and a dot production between the filter bank and local patch during direct convolution. The result of dot production is passed onto a non-linear activation function after each execution. Outcome results from this activation function are organized into a new feature map as output. Repeating the above process for each filter, we can get a set of two-dimensional feature maps as the output of the convolutional layer. Presentative implementations of direct convolution include cuda-convnet2 (Krizhevsky, et al., 2012), and Theano-legacy.

Convolution can be easily converted into a multiplication of two matrices by unrolling all the involved convolution operations. Highly optimized GEMM kernels cuBLAS (Nvidia, 2020) can be invoked to compute matrix multiplications. This is a more suitable job for GPU; also, matrix multiplication is the default method in Caffe (Jia, et al., 2014). Recently, cuDNN (Nvidia, 2020) adopted a GEMM-like method that users could easily use.

6.3 Parallel Computing Performance Model

In this section, we demonstrate our performance model by representing the CNN problem on a distributed GPU system. There are three types of features for the input set in our model. They are hardware characteristics include both training side GPUs and host side CPUs; the CNN architecture includes each type of layer and communication network such as GPU cluster architecture. The algorithm for performance estimate is based on the training time of CNN on multiple GPU, the model analyzes the CNN architecture layer by layer and gets the number of forward and backward propagations through statistic counters. Then the algorithm collects the characteristic parameters of GPUs in the cluster and the instruction model of the GPU execution. With that information, the execution time of each layer can be predicted. After collecting each layer execution time, the total execution time is the sum of each layer execution time. in the transmission model, we have considered both the parameter server model and the NCCL model. The calculation of transfer time depends on the transmission model. Finally, the iteration time of a CNN is obtained according to the computation time and transmission time.

Table 6-1: Computation notations.

Name	Description
t_{total}	One iteration time of CNNs training
$t_{forward}$	Time of forward propagation
$t_{backward}$	Time of backward propagation
t_{update}	Time of parameter update
$t_{forward}^l$	Forward execution time of layer l
$t_{backward}^l$	Backward execution time of layer l
t_{kernel}^i	Execution time of CUDA kernel function
C_{kernel}^i	Number of GPU clock cycles required by kernel function
I	Number of global memory load instructions in one block iteration
M	Number of shared memory instructions in one block iteration
L	Number of CP instructions in one block iteration
K	Number of global memory store instructions in one block iteration

6.3.1 Time of One Training Iteration

According to the CNN training processing algorithm, we define t_{total} as one iteration time of training from the first layer to the last layer, and the formula can be described as **Eq.6-1** which is derived from **Eq.5-9** in the abstract model for building a parallel performance model.

$$t_{total} = t_{loading\ data} + t_{loading\ instr} + t_{exe} + t_{update} \quad \text{Eq. 6-1}$$

In this case, we combine the time of loading data and the time of loading instructions to loading time. $t_{loading}$.

$$t_{loading} = t_{loading\ data} + t_{loading\ instr} \quad \text{Eq. 6-2}$$

The execution time could be divided to forward passing time and backward passing time.

$$t_{exe} = t_{forward} + t_{backward} \quad \text{Eq. 6-3}$$

So, the new iteration time is shown as the following equation.

$$t_{total} = t_{loading} + t_{forward} + t_{backward} + t_{update} \quad \text{Eq. 6-4}$$

Before computing, the training data must be sent to GPU from memory or storage, and the data transfer time equals to $t_{loading}$. The update time t_{update} include computation time of parameters update and transformation time of data passing between each device. The detailed calculation process will be explained in 6.3.3. We split the CNN network into multiple layers and count each layer's processing time one by one. After getting the last layer processing time, we add all processing time together to get the forward time or backward time. We can get the calculation formulas of forward and backward time by adding the time of each layer as shown in **Eq. 6-5** and **Eq. 6-6**.

$$t_{forward} = \sum_{l=1}^N t_{forward}^l \quad \text{Eq. 6-5}$$

and

$$t_{backward} = \sum_{l=1}^N t_{backward}^l \quad \text{Eq. 6-6}$$

In the CNN training processing, most of the computation of training is the matrix operation between vectors which could be parallely executed on GPU, and that operations are well-optimized by the CUDA library like cuDNN. In each layer, the operations are executed by the CUDA kernel. For each CUDA kernel, the processing time can be estimated by the performance model. Therefore, according to the basic operation process and calculation order of matrix multiplication in CUDA, we regard the computation task as a serial execution process of multiple CUDA kernel functions, which can be expressed as follows, where M is the numbers of kernels.

$$t_{forward}^l = \sum_{i=1}^M t_{Kforward}^i \quad \text{Eq. 6-7}$$

and

$$t_{backward}^l = \sum_{i=1}^M t_{Kbackward}^i \quad \text{Eq. 6-8}$$

6.3.2 GPU Instruction Queue Model

In the abstract parallel model, we simplify the execution of kernel function into four steps: namely, loading data from the global memory, loading instructions from the shared memory, instruction execution, and collecting the results data and storing it to the global memory. Correspondingly, we define four types of abstract instructions to represent the four steps as we mentioned in 5.3.2; they are Global Load, Shared Load, instruction execution, and Global Store. The number of these instructions is dependent on warps and limited by the specification of hardware, such as the block size and shared memory size. In this performance model, we assume that the hardware resources of GPU have maximum utilization, there are no data conflicts in the transfer process, also all the instruction passing touch the top bandwidth of the memory (Nvidia, 2019). Therefore, we convert the execution time of a kernel to the count of operations. The time of each instruction required depends on the performance of the target GPU. The Execution time of the CUDA kernel function can be obtained as **Eq. 6-9**.

$$t_{kernel}^i = C_{kernel}^i \times t_{Gclock} \quad \text{Eq. 6-9}$$

In the actual work, there are different instructions, such as memory instruction and computation instruction. Memory instruction need to access the local memory or shared memory while computation instruction needs the computing core to execute the

instructions. Our GPU instruction queue model considers two conditions: memory-intensive and computation-intensive. The memory-intensive means that the number of memory load instructions is much larger than the number of computing operations, and **Figure 6-4** shows the general execution pipeline of the streaming multiprocessors.

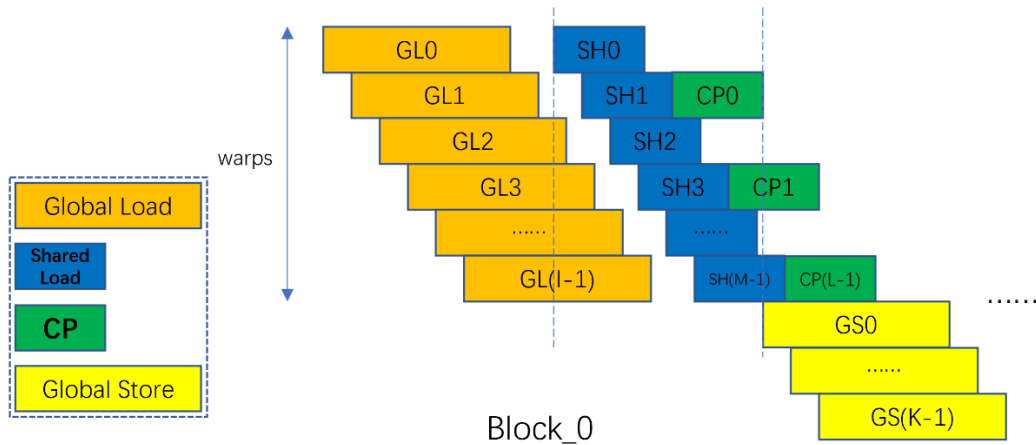


Figure 6-4: Memory intensive queue model. There are I Global Load instructions, M Shared Load instructions, L CP instructions, and K Global Store instructions in each block iteration.

In the kernel, all the instructions are executed by the warps, and multiple warps are executed in the blocks, the blocks executed in the same SM at the same time. The iteration end until all warps be executed in the block. The number of clock cycles and the GPU performance can be found in the GPU document. In this scenario, computing instructions are completely overlapped by Shared Load instructions as shown in **Figure 6-5**. The number of clock cycles used in the execution of a block is equal to the sum of the other instructions. **Eq. 6-10** shows the computation process of the queue execution time.

$$C_{kernel} = C_b \times N_{block} + L_{GL} \quad \text{Eq. 6-10}$$

Furthermore, the computation-intensive model means that the number of computing instructions is much larger than memory load instructions. Multiple computation instructions can be executed in parallel to take advantage of multiple SM units in the GPU. As shown in **Figure 6-5**, computing instructions can overlap the Shared Load instructions.

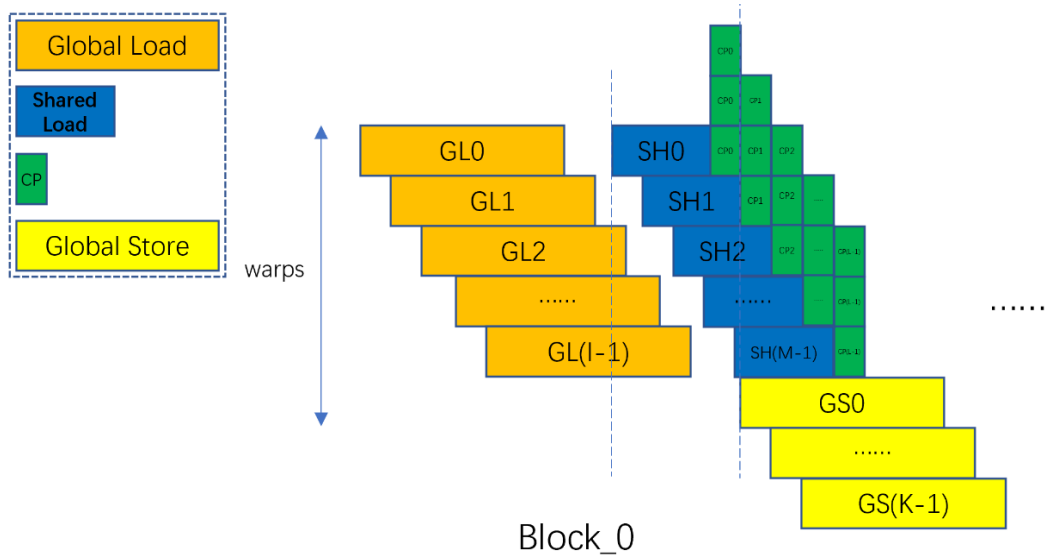


Figure 6-5: Computation intensive queue model. There are I Global Load instructions, M Shared Load instructions, L CP instructions, and K Global Store instructions in each block iteration.

In this situation, the number of computing instructions decides the block execution time. The computing instructions are executed by the SP unit. The transmission instructions time is decided by the speed of load store units. The coverage ability of computing instructions is limited by hardware conditions.

$$C_k = C_b \times N_{block} + L_{GS} \tag{Eq. 6-11}$$

6.4 Experiments and Evaluation

In this section, we evaluate our performance modeling concerning the scalability of the CNN on a distributed GPUs system. We focus on the image classification task where CNN is most successfully applied and very computationally intensive. The input of the model is a set of characteristic parameters including CNN layer information, GPU hardware characteristics, and network structure performance, as shown in **Table 6-2**. To verify the accuracy of our model, we compare the estimated results with the actual runtime of the target CNNs model. We first use CIFAR-10 (Karpathy, 2011) to train the Alexnet and Resnet-50 network and ensure the accuracy of two CNNs close to the results in the original paper, and then we generate random numbers of the same size as the dataset in the same environment to get the real CNN runtime. After 10 rounds of hot start, the average time of 20 iterations is used as the result. The experimental platform is Intel I9 CPU model 9900K 3.60 GHz, Ubuntu 16.04.1, python 3.5.2, tensorflow-gpu 1.8.0, CUDA 9.0, cuDNN 7.1.4, NVIDIA GTX 1080 Ti, and GTX 1080.

Table 6-2: Inputs parameters.

Description	Source
Layer type	CNN architecture
Input feature map (height, width)	CNN architecture
Output feature map	CNN architecture
Filter (height, width)	CNN architecture
Batch size	CNN architecture
GPU version	Hardware feature
Calculate ability	Hardware feature
Number of CUDA core	Hardware feature
Memory bandwidth	Hardware feature
CPU frequency	Hardware feature
Transmission model (PS/NCCL)	Network feature
Number of GPU	Network feature
Network bandwidth	Network feature

6.4.1 Layer Time Evaluation

We choose Alexnet and Resnet-50 to be the examples to evaluate the training time. **Figure 6-6** and **Figure 6-7** reflect the estimated time and runtime of each layer. In Alexnet, our model found that conv1 is the main bottleneck of the whole network. The reason for the issue is the filter size of conv1 is 11*11, the big size filter needs a large amount of calculation in the input layer. In the real world, to improve the performance of training processing. Researchers try to use a smaller filter that reduces the computing requirements and increases the performance efficiency. We find out that the estimated results of the model do not always follow the actual results. The reason for that is that in the actual execution when the matrix dimensions increase, the execution performance does not increase linearly. In the model, we only consider convolution, pooling, and full connected, which are the critical and most time-consuming parts of CNN.

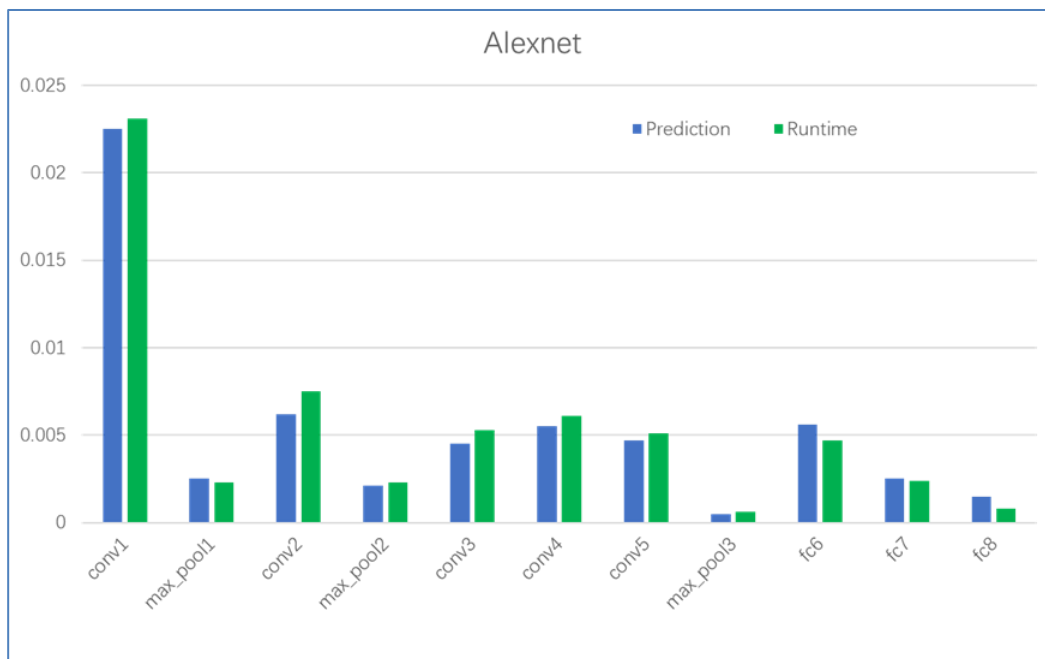


Figure 6-6: Comparison of runtime prediction for each layer in Alexnet (batch size 256).

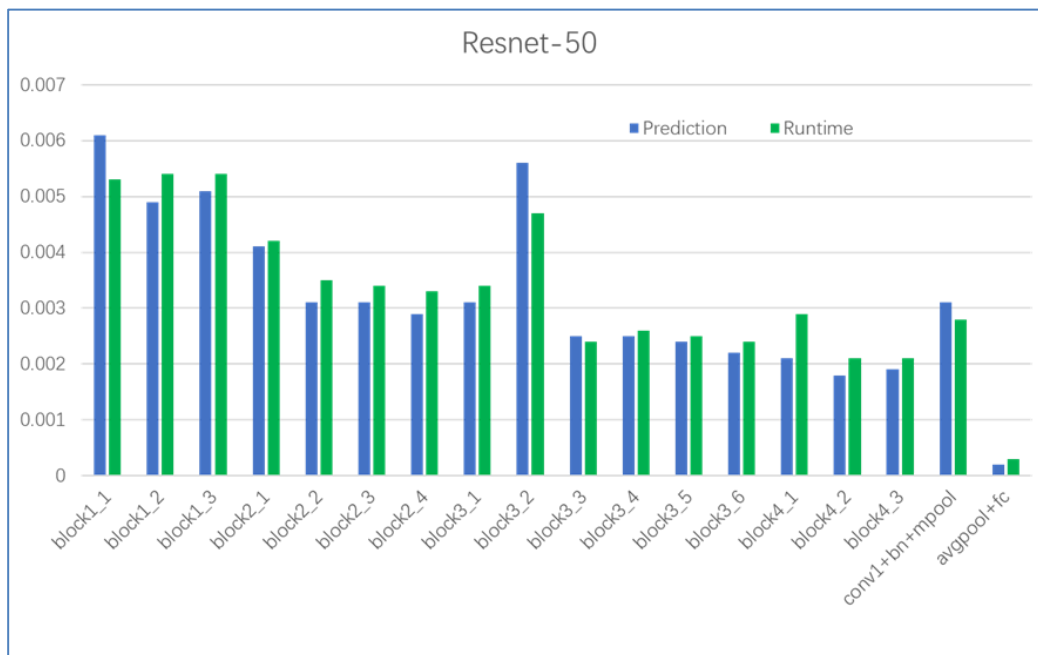


Figure 6-7: Comparison of runtime prediction for each layer in Resnet-50 (batch size 256).

6.4.2 Transmission Time Evaluation

Multi-GPU execution mode in TensorFlow is one of the most important cases and our performance model supports such a model. To fully use the performance of the GPU cluster and eliminate other interference, we use data parallelism to train Resnet-50 on NVIDIA GTX 1080 Ti and GTX 1080. According to the differences in transmission methods, we compare iteration times in the PS mode and the NCCL mode, respectively, as shown in **Table 6-3** and **Table 6-4**.

Table 6-3: Resnet-50 data parallel comparison between actual runtime and model prediction in the PS mode.

PS mode	Runtime(s)	prediction(s)
GPU	Total	Total
GTX 1080	1.72458	1.68548
GTX 1080 Ti	1.64589	1.62549
GTX 1080 GTX 1080 Ti	0.92458	0.90158

Table 6-4: Resnet-50 data parallel comparison between actual runtime and model prediction in the NCCL mode.

NCCL mode	Runtime(s)	prediction(s)
GPU	Total	Total
GTX 1080	1.73458	1.65489
GTX 1080 Ti	1.65489	1.63258
GTX 1080 GTX 1080 Ti	0.94589	0.92158

Compared with PS mode, NCCL has better performance on the transmission bandwidth. It needs more GPUs' computing resources, which could impact computational efficiency. Comparing the predicted result of our performance model will help users make decisions to pick a suitable transmission mode, especially when using TensorFlow. Our experiment employs two different GPU cards, and the results did not

show too much improvement due to the system bottleneck is the GTX 1080. In this experiment, we choose data parallelism, which runs the same model on both cards. From our observation, GTX 1080 processing performance is slower than GTX 1080 Ti, perhaps because GTX 1080 has less memory size than GTX 1080 Ti.

6.5 Conclusion

In this chapter, we demonstrated our analytical performance model to predict the CNN's training time on a distributed GPU system. We constructed a GPU instruction queue model and transmission model, which ultimately will help AI researchers make the right decision to run their application effectively, such as the CNN training process from multiple perspectives. We verified our model on two different NVIDIA GPU cards and two CNN architectures. The training time includes both layer processing and network communication. Our results suggested that the accuracy of our model is up to 95.37%.

In addition, our model also had a decent performance prediction in multi-GPU. Finally, we showed that our abstract models and framework can fit more types of CNN architectures without too many changes. In the future, we will explore different classes of Deep learning architectures and parallel computing problems.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

The past decade has witnessed significant advancement in the performance of multicore and manycore processors, especially the general-purpose GPUs. With the rapid growth of computing cores in these highly parallel platforms and increasingly complex system architectures, comprehensive understanding of performance and efficient utilization of performance in heterogeneous systems becomes a serious challenge, especially in how to program, schedule, and allocate resources, e.g., registers, caches, and thousands of computing cores and threads. These ever-changing application requirements and various kinds of hardware co-processors and connectivity become normal. Effective analytical performance modeling even becomes a paramount important skill set for programmers. A framework for building the parallel computing abstraction models and analytical performance model guides users who want to represent their application logic and analyze performance outcomes for various systems quickly.

7.1 Conclusions

This dissertation started with motivation and current issues. In chapter 2, we gave an overall background of the GPU and related topics. Chapter 3 proposed our novel analytic performance model with the GPU block size estimation, in which the first model considered the effective block size on performance. Our model revealed the GPU

performance characteristics by analyzing hardware device characteristics, memory allocating, thread block organization, memory latency hiding, memory characteristic of memory hierarchies, coalesced memory, data reuse rate, and memory accessing pattern. The analytical GPU performance model can potentially identify bottlenecks without running the actual program. Using a suitable block size for GPU applications, users can improve the application performance with ease. Chapter 4 presented a novel method aiming to alleviate some limitations of GPU applications. We proposed the dynamic partition of the SMs for each kernel based on the computational throughput estimated by GPU performance modeling. Our result showed an increase in performance without any changes.

In the second part of this dissertation, we focused on a general parallel computing and distributed system analytical model. In chapter 5, we presented two parallel computing abstract models. These models represented program logic and algorithmic steps and simplify the workload distribution behaviors. An extension to Flynn's taxonomy was proposed to support heterogeneous systems with communication time consideration. Chapter 6 illustrated a demonstration of our proposed modeling techniques with real-world application on a distributed GPU system. The analytical performance model for the CNN application analyzed performance characteristics on multiple GPUs, enabling users to evaluate their techniques before running applications on targeted machines/architecture.

7.2 Future Work

We hope that the GPU performance model can be applied to future GPU hardware. The performance model can also be extended to support other multiple and many-core processors in the future, like the artificial intelligence accelerator processor.

BIBLIOGRAPHY

- Alexandrov, A., Ionescu, M. F., Schauser, K. E. & Scheiman, C., 1995, July. LogGP: incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pp. 95-105.
- Amdahl, G. M., 1967, April. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483-485.
- Anderson, R. J. & Miller, G. L., 1990. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33(5), pp. 269-273.
- Awatramani, M., Zambreno, J. & Rover, D., 2013, October. Increasing gpu throughput using kernel interleaved thread block scheduling. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pp. 503-506.
- Baghsorkhi, S. S. et al., 2010, January. An adaptive performance modeling tool for GPU architectures. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 105-114.
- Bauer, M., Cook, H. & Khailany, B., 2011, November. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, pp. 1-11.
- Che, S. et al., 2009, October. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pp. 44-54.
- Che, S., Sheaffer, J. W. & Skadron, K., 2011, November. Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, pp. 1-11.
- Cui, H. et al., 2016, April. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, pp. 1-16.

- Culler, D. et al., 1993, July. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 1-12.
- Danalis, A. et al., 2010, March. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 63-74.
- Flynn, M. J., 1972. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9), pp. 948-960.
- Gregg, C. & Hazelwood, K., 2011, April. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 134-144.
- Guevara, M., Gregg, C., Hazelwood, K. & Skadron, K., 2009, September. Enabling Task Parallelism in the CUDA Scheduler. In *Workshop on Programming Models for Emerging Architectures*, Volume 9.
- Gustafson, J. L., 1988. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5), pp. 532-533.
- Han, H. & Tseng, C. W., 2006. Exploiting Locality for Irregular Scientific Codes. *IEEE Transactions on Parallel and Distributed Systems*, 17(7), pp. 606-618.
- Han, S. et al., 2017, February. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 75-84.
- He, K., Zhang, X., Ren, S. & Sun, J., 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770-778.
- Helman, D. R. & JáJá, J., 1999, January. Designing Practical Efficient Algorithms for Symmetric Multiprocessors. In *Workshop on Algorithm Engineering and Experimentation*, pp. 37-56.
- Hong, S. & Kim, H., 2009, June. 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*, pp. 152-163.
- ILSVRC, 2020. *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)*. [Online] Available at: <http://www.image-net.org/challenges/LSVRC/>
- Jang, B., Schaa, D., Mistry, P. & Kaeli, D., 2010. Exploiting Memory Access Patterns to Improve Memory Performance in Data Parallel Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1), pp. 105-118.

- Jia, W., Shaw, K. A. & Martonosi, M., 2012, June. Characterizing and improving the use of demand-fetched caches in GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 15-24.
- Jia, Y. et al., 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675--678.
- Karpathy, A., 2011. *Lessons learned from manually classifying CIFAR-10*. [Online] Available at: <http://karpathy.github.io/2011/04/27/manually-classifying-cifar10/>
- Kayıran, O., Jog, A., Kandemir, M. T. & Das, C. R., 2013, September. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pp. 157-166.
- Khokhar, A. A., Prasanna, V. K., Shaaban, M. E. & Wang, C. L., 1993. Heterogeneous Computing: challenges and opportunities. *Computer*, 26(6), pp. 18-27.
- Krizhevsky, A., Sutskever, I. & Hinton, G. E., 2012. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in neural information processing systems*, Volume 25, pp. 1097-1105.
- Kumar, V. & Rao, V. N., 1987. Parallel depth first search. part II. analysis. *International Journal of Parallel Programming*, 16(6), pp. 501-519.
- LeCun, Y., Bengio, Y. & Hinton, G., 2015. Deep learning. *Nature*, 521(7553), pp. 436-444.
- LeCun, Y. et al., 1989. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4), pp. 541-551.
- LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P., 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), pp. 2278-2324.
- Luehr, N., 2016. *Fast Multi-GPU collectives with NCCL*. [Online] Available at: <https://developer.nvidia.com/blog/author/nluehr/>
- Mei, X. & Chu, X., 2016. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1), pp. 72-86.
- Mittal, S. & Vetter, J. S., 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys (CSUR)*, 47(4), pp. 1-35.
- Nussbaumer, H. J., 1981. The Fast Fourier Transform. In *Fast Fourier Transform and Convolution Algorithms*, pp. 80-111.

- Nvidia, 2019. *CUDA Toolkit Documentation*. [Online]
Available at: <https://docs.nvidia.com/cuda/>
- Nvidia, 2020. *CUBLAS*. [Online]
Available at: <https://developer.nvidia.com/cublas>
- Nvidia, 2020. *NVIDIA CUDNN Documentation*. [Online]
Available at: <https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html>
- Pai, S., Thazhuthaveetil, M. J. & Ramaswamy, G., 2013. Improving GPGPU concurrency with elastic kernels. *ACM SIGARCH Computer Architecture News*, 41(1), pp. 407-418.
- PASCAL, 2020. *The PASCAL Visual Object Classes Homepage*. [Online]
Available at: <http://host.robots.ox.ac.uk/pascal/VOC/>
- Ranade, A., 1998. An introduction to parallel computation. *Resonance*, 3(1), pp. 47-60.
- Sergeev, A. & Del Balso, M., 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*.
- Simonyan, K. & Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.
- Szegedy, C., Liu, W. & Jia, Y., 2015. Going Deeper with Convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1-9.
- TOP500, 2020. *Top 500 Supercomputing Sites*. [Online]
Available at: <https://www.top500.org/lists/top500/2020/11/>
- Winograd, S., 1980. Arithmetic complexity of computations. *Siam*, Volume 33.
- Wu, B. et al., 2015, June. Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pp. 119-130.
- Wu, Y. et al., 2016. ‘Google’s neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.
- Xiao, S. & Feng, W. C., 2010, April. Inter-block GPU communication via fast barrier synchronization. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1-12.
- Zhang, Y., Cohen, J. & Owens, J. D., 2010. Fast tridiagonal solvers on the GPU. *ACM Sigplan Notices*, 45(5), pp. 127-136.
- Zhang, Y. & Owens, J. D., 2011, February. A quantitative performance analysis model for GPU architectures. In *2011 IEEE 17th international symposium on high performance computer architecture*, pp. 382-393.