The Pennsylvania State University

The Graduate School

College of Engineering

A FRAMEWORK FOR ANALYZING APPLICATION INTERFERENCE ON GPUS

A Thesis in

Computer Science and Engineering

by

Tuba Kesten

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science

December 2014

The thesis of Tuba Kesten was reviewed and approved* by the following:

Chita R. Das
Professor of Computer Science and Engineering
Thesis Co-Adviser

Mahmut T. Kandemir
Professor of Computer Science and Engineering
Thesis Co-Adviser

Lee Coraor
Professor of Computer Science and Engineering
Director of Academic Affairs

*Signatures are on file in the Graduate School

ABSTRACT

Graphical Processor Units (GPUs) have a widespread usage in diverse areas such as manufacturing, research, health, life sciences, engineering etc. to accelerate general-purpose computation. To achieve better speedups in general-purpose computation, the available resources are increasing for each new generation of GPUs. In practice, most GPU applications in high-performance computing cannot effectively utilize all of the resources in the system due the lack of enough thread-level parallelism. This underutilization hurts the overall performance in terms up speedup and throughput. To fully exploit the capabilities of GPUs, concurrent execution of applications is critical. NVIDIA's recent GPU architecture Kepler achieves concurrency management by Hyper-Q technique which assigns a separate work queue to each application. We first propose a flexible and Hyper-Q like multi-application framework which is capable of simulating 2-application and 3-application workloads. Our framework supports 25 applications and 300 of 2-application workloads chosen from Parboil, Rodinia, SHOC and CUDA application suites. Our framework provides programmers to adapt their CUDA code for concurrent execution with little programming effort. Further, we study the application interference of multi-application workloads for different core partitioning schemes in this work. We characterize applications from our application suite based on Misses-Per Kilo Instruction (MPKI) values. We evaluate the 2-application workloads constructed from these applications using various performance metrics: Weighted Speedup (WS/STP), IT, Average Normalized Turnaround Time (ANTT), Fairness Index (FI), and Bandwidth Utilization. The experiments show that MPKI is not enough to analyze the interaction among applications, and the attained bandwidth[1] of each application also needs to be taken into consideration.

[1] "Attained bandwidth" and "bandwidth utilization" can be used interchangeably in this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am using this opportunity to express my gratitude to everyone who made this thesis possible and an unforgettable experience for me.

First of all, I express my warm thanks to Prof. Chita Das and Prof. Mahmut Kandemir for giving me the opportunity to study M.Sc. degree in Computer Science and Engineering Department at The Pennsylvania State University.

I would like to thank all my HPCL lab mates for their friendship and endless support. Among them, special thanks goes to Adwait Jog, Onur Kayiran and Ashutosh Pattnaik. Without guidance of Adwait and Onur, this thesis would not come so far. I was really lucky to work with Adwait Jog who is an excellent mentor.

I don't have any siblings, but throughout this graduate school, I had many friends who are now part of my family. Firstly, I want to thank to my sisters Dr. Nurshide Pulati, Irem Aksu and my brother Cihan Can for sharing both good and bad with me. I really appreciate their friendship and trust. Secondly, I want to thank my Bodrumers: Gokce Islertas, Togan Gencoglu, Didem Demirel and my dear friends Dilan Suslu, Gizem Demirel for encouraging me from all the way from Turkey.

I also would like to give my gratitude to my lovely roommates, Aigerim Jumatayeva and Ainur Ilyubayeva for their ultimate encouragement and making me laugh at my most stressful moments.

I would like to thank my dear friends, Nilsu Kistak, Melike Faiz, Ibrahim Ekrem Bardakci, Sema Erten, Firdevs Ilci and Ozge Atil for making my stay in State College happy and delightful.

Most importantly, none of this would have been possible without unconditional love and patience from my amazing family. The source of my happiness has always been them. Throughout this journey, their support helped me to manage the stresses of graduate school. They always reinforce me to be a better person in the life. This thesis is mainly dedicated to my beloved parents, Gonca Kesten and Ayhan Kesten.

Tuba Kesten

December 2014

To my beloved parents

Gonca and Ayhan Kesten,

Chapter 1

Introduction

Graphical Processor Units (GPUs) have traditionally been used in gaming industry to create real world experience to the users. Since the last decade, the widespread adoption of GPUs occurs in the general processing computing due to their tremendous computational power and high bandwidth. They play a big role in accelerating many high performance applications in many computational platforms. [**1**,**2**,**3**] The capability of running high number of threads in parallel reduces the work for CPU, especially for data-intensive computations. Currently, 64 supercomputers in TOP500 [**4**], including many near the top of the list, use GPUs to benefit in High Performance Computing (HPC) applications. Another example of how GPUs takes place in our life is Google's project Tango [**5**]. Tango project is trying to scale the physical world into the boundaries of digital screen on the phone. With the help of GPUs, Tango can record quarter millions of 3D measurements in a second.

As GPUs start to take an active role in general purpose computing, the GPU vendors increase the available resources in each new generation. Current NVIDIA Kepler [**3**] architecture has increased the CUDA cores and core resources such as warp schedulers where a warp is a group of 32 threads, register file size etc. compared to previous Fermi [**2**] architecture. To benefit from these available resources, CUDA framework provides developers simple compiler directives to exploit the parallelism in the program. However; recent works address that GPUs lead to resource underutilization due to single application execution. [**6**,**7**,**8**] To illustrate this phenomenon, we first execute a CUDA application alone, then schedule it with one of the other applications from our application suite. In order to show how effectively the core resources are used by the applications, we calculate the warp utilization of the system. In a 30-core system,

alone execution of LUD achieves 13.4% utilization. During the concurrent execution, we distribute the resources evenly to each application. Figure 1-1 shows the warp utilization for these workloads and the alone execution. All the cases for concurrent execution of workloads gives better utilization rate compared to the utilization rate of the alone execution. For instance, scheduling LUD with RAY achieves 35% utilization. The benefit from concurrent execution depends on the application characteristics. Hence, analyzing the interference among applications is crucial for future research directions in GPUs.

NVIDIA Fermi architecture exploits stream mechanism (Chapter 2.3) to execute multiple applications; however, the scope of the concurrent execution is limited. When an application is launched on the GPU, the second application can be launched only if there are enough resources. To handle with this problem, NVIDIA Kepler architecture improves concurrency mechanism by introducing Hyper-Q mechanism, in which each application has separate streams so the concurrent execution is guaranteed.

In this study, we propose a framework extended from GPGPU-Sim simulator that adapts the Hyper-Q mechanism. With the help of our framework, researchers can simulate two-application or three-application workloads constructed from 25 benchmarks. These benchmarks are chosen from various application suites: Rodinia [**9**], CUDA [**1**], Parboil [**10**], and SHOC [**11**]. We choose 9 representative applications and form 2-application workloads to analyze application interference, we examine the implications of different core partitioning schemes across applications. There are further research problems can be explored such as cache partitioning, dynamic resource scheduling etc. For this purpose, we release our framework for public.[github]

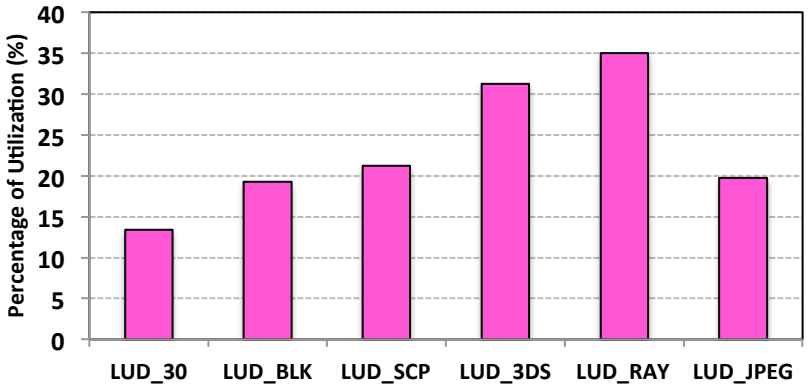


Figure 1-1: Fraction of warp utilization when LUD co-scheduled with various workloads

There are four main **contributions** presented in this thesis:

- We develop a flexible multiple application framework called MAFIA (**M**ultiple **A**pplication **F**ramework **I**n GPU **A**rchitectures) by extending GPGPU-Sim.
- We perform a detailed analysis of application characteristics and interference among multiple applications in GPUs through concurrent execution.
- We conduct experiment for analyzing SM partitioning schemes in multiple application context.
- We report experimental results that show the capability of our proposed framework.

The thesis is organized as follows. Chapter 2 gives insights about current NVIDIA GPU architecture and how the concurrency is achieved on these GPUs. Recent research studies on concurrent execution of GPU applications can be found in Chapter 3. Chapter 4 presents the details of the proposed framework. Chapter 4.2 provides information about GPGPU-Sim on which the proposed framework is developed. Chapter 5 describes the experiment setup and presents the performance results of the evaluated benchmarks. The thesis is concluded in Chapter 6 with a summary of our major observations and possible future research directions.

Chapter 2

Background

2.1 NVIDIA GPU Architecture

In this study, we use NVIDIA GPU architecture with CUDA programming model. There are two main components of GPUs: (1) Global Memory. (2) Streaming Processors (SM).In this chapter, we briefly discuss the basic terminology related to CUDA and CUDA programming paradigm.

2.1.1 Streaming Processor Model

Streaming Processors are responsible for the actual computations of CUDA programs. Each SM contains a number of CUDA cores. CUDA v2.x capable GPUs can have CUDA cores up to 48. In the recent NVIDIA GPU architecture Kepler, a fourfold increase occurs compared to Fermi (192 CUDA cores). Each CUDA core has a fully pipelined arithmetic logic unit (ALU) and floating-point unit (FPU). FPU provides the fused multiply-add (FMA) instruction to support both single and double precision.

As GPU computing has a widespread adoption through scientific applications, double precision arithmetic performance gains importance in architecture design. For this reason, GPUs have special function units (SFU) that execute transcendental instructions such as sine, cosine, and square root. The recent architecture Kepler provides eightfold number of SFUs in Fermi GF110 SM to increase the performance of HPC applications.

NVIDIA GPUs use a single instruction, multiple threads (SIMT) execution model that is a hybrid of single instruction multiple data (SIMD) and simultaneous multithreading (SMT) models. SIMT is more flexible than SIMD regarding that it can execute single instruction with multiple register sets, multiple memory addresses or multiple flow paths. SIMD vector architectures express parallelism within vector instructions, while SIMT provides parallelism among independent threads. SIMT architecture is more effective in terms of understanding the behavior of independent threads. However, it introduces extra cost due to the hardware barrier synchronization.

2.1.2 Memory Model

In this subsection, we present the details of GPU Memory hierarchy. Figure 2-1 shows the overview of memory on a GPU. The main component of the memory structure is global memory which is similar to RAM in CPU and accessible by both CPU and GPU.The nature of GPU computing allows multiple accesses to global memory, however; these accesses can cause memory incoherence. It only guarantees that the order of memory reads and writes for the same address will be preserved in the thread. The global memory is specialized for maximizing the throughput instead of minimizing the latency.

Each SM in GPU has a shared memory that resides on-chip. It provides higher bandwidth and reduces global memory bandwidth. To achieve high throughput, the shared memory is divided into modules called banks so the threads can access at the shared memory concurrently. If multiple threads want to access to the same bank, it can lead to serialization and decrease the effective bandwidth. To solve this problem, memory padding or changing address pattern can be used.[] Kepler GK110 architecture uses 64KB configurable shared memory and L1 cache that can

be divided as 48GB of Shared memory with 16KB of L1 cache or vice versa. Compared to Fermi architecture, the shared memory bandwidth is doubled to 64KB. Furthermore, Kepler introduces a new technique called warp shuffling that allows the exchange of data within threads of a warp without using the shared memory. The shuffle instruction has lower latency than shared memory and does not need any additional resources.

Besides L1 cache, Kepler adds a new 48KB read-only cache to the memory hierarchy. The data in this cache can be accessed through the lifetime of kernel. Previously in Fermi, this read-only cache can be used through texture memory that restricts mapping data into textures. By removing this restriction in Kepler, unaligned memory access patterns are supported by this cache.

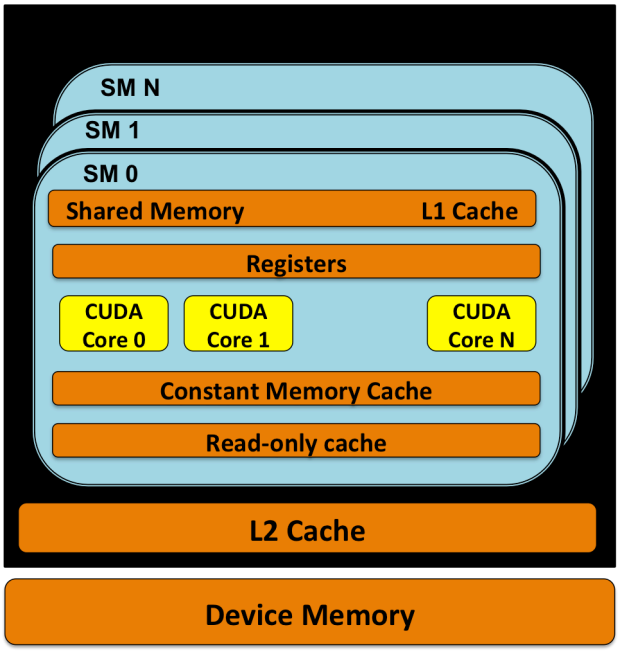


Figure 2-1: Memory Hierarchy in NVIDIA CUDA Streaming Processor

2.2 Application Execution on GPUs

2.2.1 Single Application Execution on GPUs

A typical CUDA program consists of multiple kernel grids as shown in Figure 2-2. A kernel includes multiple thread blocks that execute the parallel portions of the program. In CUDA terminology, each of these thread blocks refers to a cooperative thread array (CTA). Each SM executes one or more CTAs and one warp at a time. A warp represents a group of 32 threads. While the kernel and thread block are programming abstractions implemented in GPU hardware, warp is a machine object.

In the context of single application execution on GPUs, each kernel of application is executed sequentially. [3] When a kernel is launched, the CTA scheduler distributes the available resources in a round-robin fashion. The number of CTAs per SM is dependent on the SM resources such as the number of registers, memory size, the number of threads etc. Also, the requirements of a CTA in a particular kernel affect this number. For instance, assume that the CTA of kernel X demands 8KB for execution. Given a baseline architecture with 32 KB shared memory, kernel X can execute only 4 CTAs due to the size of shared memory.

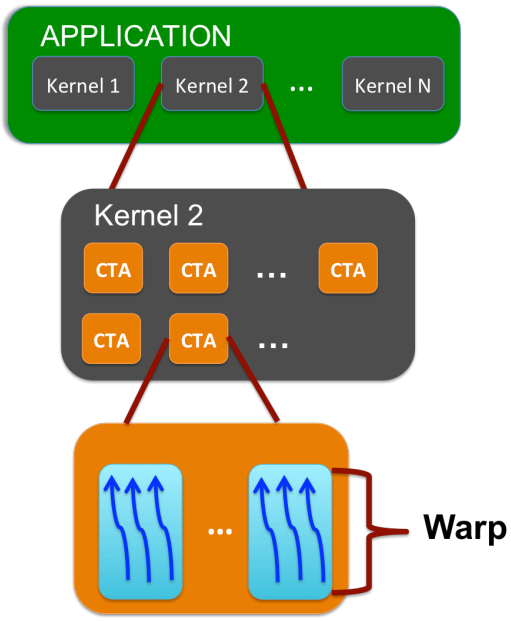


Figure 2-2: Software Architecture of GPGPUs

2.2.2 Multiple Application Execution on GPUs

Current GPUs achieve concurrent execution by using CUDA Streams. A CUDA Stream can be defined as a work queue that consists of operations inserted by the host program. Each of these operations can be a data transfer or a kernel execution. The order of these operations is always preserved during the execution. CUDA programming model allows us to create multiple streams in the same program context and execute their operations out of order based on the resource availability. If there is a dependency among kernels in different streams, it can be solved by using CUDA synchronization or using CUDA events.

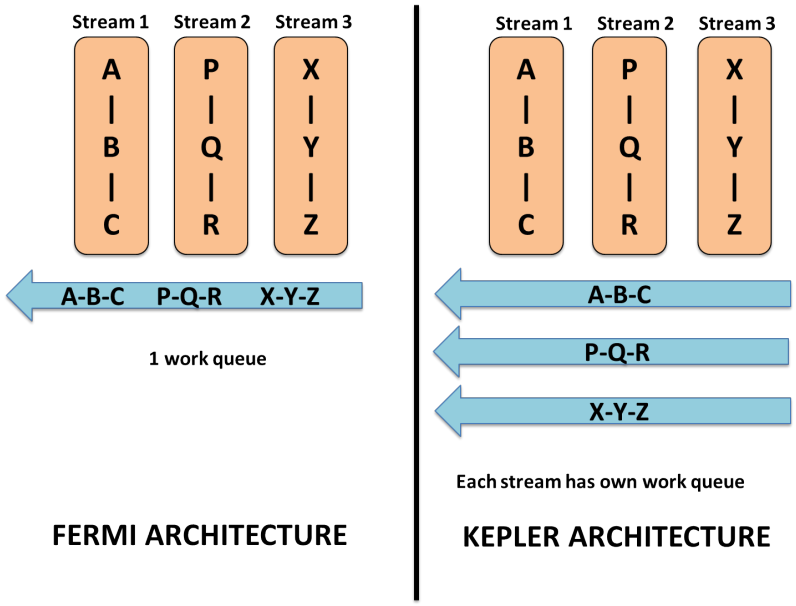


Figure 2-3: Concurrent application execution on current NVIDIA Architectures.

Figure 2-3 shows the concurrency schemes on current NVIDIA architectures. Fermi Architecture allows 16 kernel launches from different streams at the same time. However, the streams are multiplexed into the same work distributor, which introduces intra-stream dependencies. [4] The work distributor ensures that all the dependencies are resolved and then distributes the work into Streaming Processors (SMs). On the other hand, Kepler Architecture improves concurrency by introducing Hyper-Q feature. In order to reduce and eliminate the false dependencies, Kepler takes advantage of Grid Management Unit, which is capable of creating individual work queue for each stream. Compared to Fermi Architecture, Kepler supports 32-way concurrency of kernel launches from separate streams. In our work, we adapted Hyper-Q feature of Kepler architecture to GPGPU-Sim Simulator.

Chapter 3

Related Work

This section presents the recent studies done in GPU scheduling. In comparison to these prior efforts, our work presents a new concurrent application execution framework that consists of large number of CUDA workloads. Our framework provides flexibility to add new CUDA codes in the framework without much effort. We believe that this framework could be for further scheduling strategies in GPUs. Additionally, our study analyzes multiple application interference during concurrent execution.

3.1 Concurrent execution of Multiple Kernels on GPUs

Guevara et al. [**12**] introduce an issue queue that merges workloads to run on GPU simultaneously. They achieve task parallelism by using block partitioning. Their technique allows multiple kernels to be merged if the single kernel does not have enough data parallelism. They do not change SM partitioning dynamically while merging the kernels.

Gregg et al. [**13**] propose a framework called KernelMerge that executes two kernels concurrently for OpenCL benchmarks. They examine how the kernels interact and interfere each other during execution. KernelMerge uses two scheduling algorithms: Work-stealing algorithm, and partitioning algorithm. Work-stealing algorithm favors round-robin scheduling that means assignment is based on first-come first-served policy. In the partitioning scheme, resource allocation is fixed. For instance, assume that one-third of the resources are assigned to a short-running kernel and the remaining is assigned to a long-running kernel. When the scheduler workgroups finish, the work is replaced from their assigned kernel. Our study also resembles to

partitioning algorithm. However, in contrast, our framework statically allocates resources based on applications, not based on kernels.

Wang et al. [**14**] present the advantages of context funneling over context switching in multithreaded application environment. In context switching, each host thread defines its own GPU context and interacts with the GPU independently. This method implies several limitations such as serialization, synchronization overhead, and thread safety. So as to overcome these limitations; context funneling, which allows host threads to share a single context, is introduced. This method allows the concurrent execution to extend all across host threads.

Wang et al. [**15**] propose a kernel-scheduling scheme called Kernel fuse that determine the independent kernels in the application and then apply different kernel fusion methods to reduce energy consumption. The size of shared memory is selected based on the bigger one of the original kernels. Also, the operations of multiple threads are mapped into one thread to reduce the thread space. These optimizations moderate the cost of kernel-fusion.

Pai et al. [**8**]emphasize how single execution on GPUs causes resource underutilization and present the reasons behind the serialization of the kernels. To solve this problem, they propose elastic kernels mechanism that enables fine-grain resource allocation on GPUs. They implement several concurrency policies at the Stream Scheduler level and evaluate elastic kernel with these policies. Moreover, they implement time-slice kernel execution with elastic kernels to allow fair execution among long-running kernels, and short-running kernels.

Jog et al. [**6**] propose a round-robin memory scheduler for multiple GPU applications that preserves the characteristics of FR-FCFS memory scheduling scheme. Using round-robin mechanism among concurrently scheduled applications, the proposed mechanism improves both fairness and the system performance.

The closest work to our framework is proposed by Adriaens et al. [**16**] which introduces spatial multitasking that allows GPU resources to be partitioned among multiple applications simultaneously. They compare cooperative and spatial multitasking to show how their method benefits from different SM (Streaming Multi-processor) partitioning heuristics. They evaluate their scheduling scheme using GPGPU-Sim simulator. A typical GPU application consists of multiple kernels. While scheduling two kernels in cooperative multitasking, e.g. kernel A and B, kernel B can start after the completion of kernel A. The obvious disadvantage of cooperative multitasking is that the kernel will never finish execution due to the malfunction. This will yield other kernel to never start its execution. In order to solve this issue, each kernel is assigned a time slice to execute in and switch contexts at the end of each time slice, which is called preemptive scheduling. However, preemptive scheduling causes context switching overhead. Spatial multitasking overcomes these problems by dividing GPU resources among kernels rather than the execution time. In experimental evaluation of spatial multitasking, they restrict the simulations to 5M GPU cycles to observe the characteristics of whole execution. In our simulations, we simulated two-application workloads till each of them finish execution at least once. Furthermore, we present a fixed number of cycles simulation option in our framework. This option is explained in detail in Chapter 3.

3.2 Work Scheduling in GPUs

Kayiran et al. [**17**] propose a dynamic CTA scheduling mechanism called DYNCTA that decides the number of optimal CTAs considering the application characteristics. Concurrent execution of high number of threads will not be optimal for system performance. This leads to high cache contention and miss rates due to the degree of thread-level parallelism. DYNCTA

decreases this cache and the memory congestion by allocating fewer resources based on application's demands.

To solve resource underutilization problems in memory context, Jog et al. [**18**] propose four CTA-aware warp scheduling techniques which improve both L1 hit rates and DRAM bandwidth utilization. The first two techniques are CTA-aware two-level warp scheduler and locality-aware warp scheduling that improve per-core performance. The third technique enhances GPGPU performance by improving DRAM bank-level parallelism. The fourth technique takes advantage of open DRAM rows and applies opportunistic memory prefetching to boost the system performance.

Jog et al. [**19**] propose a prefetch-aware warp scheduling technique which organizes the scheduling of consecutive warps not to execute back-to-back. This technique is an effective way to tolerate memory latency and also it enhances memory bank parallelism.

The recent work proposed by Kayiran et al. [**7**] explores concurrency management in CPU-GPU architectures. They address how GPUs dominate the system performance due to the high thread-level parallelism. They propose two schemes to minimize this interference. One scheme favors CPU performance when the GPU interference occurs; other one balances both CPU and GPU performance and improves the overall system performance.

Chapter 4

Concurrent Application Framework on GPUs

4.1 Why Is Concurrent Execution Necessary?

The amount of available resources is increasing in each new generation of GPU. However, the utilization rate of resources decreases, which implies that single application execution is one of the major impediments of GPUs. NVIDIA Fermi architecture introduced the conservative approach of concurrent execution using streams in 2009. Since then, the limitations and advantages of concurrent execution have been discussed in context of memory scheduling, SM scheduling, and warp scheduling. [**6**,**7**]

Recent NVIDIA Kepler architecture extends concurrency with new concepts: Hyper-Q and dynamic parallelism. Dynamic Parallelism allows GPU to create new work for itself and Hyper-Q improves the concurrency by creating independent work queue for each stream. So far, there is no simulation framework available that supports these new concepts of GPU architectures. Our main goal is to provide a framework that supports a Hyper-Q-like concept to explore the potential research problems.

Our framework is adapted from GPGPUSim V3.x which models GPU microarchitecture similar to NVIDIA Fermi Architecture. We try to choose the closest configuration parameters regarding the latest release Kepler Architecture.

4.2 Existing Simulation Tool

GPGPU-Sim is a simulation tool [**20**] that provides timing model for SM cores, caches, interconnection network, and memory partition on GPUs. It reports the number of cycles spent

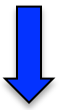
for each launched kernel. It does not consider the time spent on the memory transfer time between CPU and GPU memory. The tool allows the CPU to run concurrently with asynchronous kernel launches.

GPGPU-Sim emulates the NVIDIA's virtual GPU instruction set called Parallel Thread eXecution (PTX). PTX is a pseudo assembly instruction set that cannot execute on NVIDIA hardware directly so it needs to be assembled into native instruction set (SASS) by the hardware. Based on hardware, PTX is compiled into multiple versions of SASS. Since each hardware generation has a different type of SASS, the binary version of the PTX is stored to meet the future hardware requirements.

4.3 Extending GPGPU-Sim for Multiple Application Execution

Recent GPU studies focus on kernel-based concurrent scheduling rather than application-based as described in previous Chapter 3. Most of them evaluate their proposed solution based on several benchmarks, at most 11 benchmarks, taken from different application suites. Our main purpose is to provide a flexible and extensive framework that will help to explore multiple application context on GPUs. However, building a framework is an arduous task since the diverse features of application suites. We adapted benchmarks from 4 application suites (CUDA SDK, Parboil, SHOC, and Rodinia) and each of them includes different main function and arguments. Thus, we need to merge them in one executable file to provide access to them from same platform. To form an n-application workload, we create a new thread for each application resulting in an n-threaded program. As mentioned in the previous sections, we take advantage of CUDA streams to execute multiple applications. The framework can launch different streams in parallel, but internal operations of the stream are executed in serial manner.

```
printf("Executing GPU kernel...\n");
cutilSafeCall( cudaThreadSynchronize() );
cutilCheckError( cutResetTimer(hTimer) );
cutilCheckError( cutStartTimer(hTimer) );
scalarProdGPU<<<128, 256>>>(d_C, d_A, d_B, VECTOR_N, ELEMENT_N);
cutilCheckMsg("scalarProdGPU() execution failed\n");
cutilSafeCall( cudaThreadSynchronize() );
cutilCheckError( cutStopTimer(hTimer) );
printf("GPU time: %f msecs.\n", cutGetTimerValue(hTimer));
printf("Reading back GPU result...\n");
//Read back GPU results to compare them to CPU results
cutilSafeCall( cudaMemcpy(h_C_GPU, d_C, RESULT_SZ,cudaMemcpyDeviceToHost) );
```



Modify CUDA code for concurrent execution

```
printf("Executing GPU kernel...\n");
cutilSafeCall(cudaStreamSynchronize(stream_app));
cutilCheckError(cutResetTimer(hTimer));
cutilCheckError(cutStartTimer(hTimer));
scalarProdGPU<<<128, 256, 0, stream_app>>>(d_C, d_A, d_B, VECTOR_N, ELEMENT_N);
cutilCheckMsg("scalarProdGPU() execution failed\n");
pthread_mutex_unlock (mutexapp);
cutilSafeCall(cudaStreamSynchronize(stream_app));
cutilCheckError(cutStopTimer(hTimer));
printf("GPU time: %f msecs.\n", cutGetTimerValue(hTimer));
printf("Reading back GPU result...\n");
//Read back GPU results to compare them to CPU results
cutilSafeCall(cudaMemcpyAsync(h_C_GPU,d_C,RESULT_SZ,cudaMemcpyDeviceToHost,stream_app));
```

Figure 4-1: Modification of Scalar Product kernel for concurrent execution

For each thread function, the framework creates a separate CUDA Stream, and issues all its necessary commands to this stream. To protect shared data between n applications and thread synchronization, the framework uses POSIX mutex variables [**21**]. Since the original benchmarks do not include stream operations, the framework needs to modify the synchronous operations in each application. Our framework converts the CUDA API memory transfer operations to

asynchronous CUDA Stream API calls. (e.g CudaMemCpyAsync()) To ensure the correct execution of multiple streams, our framework also adds asynchronous synchronization calls (e.g. CudaStreamSynchronize( )) to necessary places in the source code. After the basic modifications, our n-application workload is ready to launch on GPGPU-Sim which is capable of simulating multiple CUDA or OpenGL applications.

Figure 4-1 presents a code snippet that shows the modified version of SCP for concurrent execution. The highlighted parts show the modifications done by our framework. Firstly, we convert synchronous memory transfers to asynchronous memory transfers. (e.g. cudaMemcpy() to cudaMemcpyAsync()) Secondly, we pass the required parameters such as stream id to the kernel function. Thirdly, we add necessary mutex operations to ensure safety of data during kernel execution. Lastly, we modify the synchronization commands (e.g. cudaThreadSynchronize() to cudaStreamSynchronize()) to block the stream till the operations are completed.

Initial release of the framework consists of 300 (25 choose 2) two-application workloads and 2300 (25 choose 3) three-application workloads. We provide the basic guidelines on how to modify the existing CUDA code to our framework. We release MAFIA framework for public use and future research studies in this domain.

4.4 Evaluation Methodology

Our framework is capable of simulating two- and three-application workloads on GPGPU-Sim. The concurrent execution of applications cannot be completed simultaneously since the unique characteristics of applications. To illustrate this problem, assume that SCP and JPEG are running on the simulator concurrently as shown in Figure X. JPEG's execution time is three times shorter than SCP's execution time; so, when JPEG completes the execution, SCP only

completes 1/3 of the simulation. To deal with this problem, we use the Tuck and Tullsen method which relaunches the applications till both of them are completed at least once. We collect the statistics of individual application at two points based on their respective completion times. The applications used in this study follow this trend; however, some of the applications in the remaining portion take several days to complete. For the sake of feasibility, we assume that the application to be tagged as finished when it reaches to M million cycles. The user can modify this M parameter in configuration file. This option provides user to observe the performance trends till that cycle.

4.5 Benchmarks

The benchmarks are listed in Table 4-1 with their main properties such as block dimension, grid dimension, and the application domain. We adapted applications from four different application suites: CUDA SDK, Parboil, Rodinia and Shoc. Our framework supports 25 benchmarks from these suites. The detailed description for each benchmark is presented below:

4.5.1 CUDA SDK

These benchmarks are taken from CUDA 4.0 Code Samples. These samples include wide range of techniques to show how CUDA is applicable in all the domains.

Graph Algorithm: Breadth-First Search (BFS): [15] This benchmark is developed by Harish and Narayanan performs BFS problem using level synchronization. In the BFS problem, the aim is to find minimum number of edges needed to reach every vertex of given unweighted graph from source vertex. In CUDA implementation of BFS, each thread represents one vertex therefore; the degree of parallelism is scaled by the input size. However, the input size for the

algorithm is limited by the size of device memory. As the algorithm uses global memory instead of shared memory, global memory traffic is one of the drawbacks for the performance. We use 32k vertex and 500k edges input for our experiment.

Black-Scholes option pricing (BLK): The option pricing is one of the important problem in financial engineering. This application implements the Black-Scholes model for European options. The Black-Scholes model provides a partial differential equation (PED) for the optimization of an option price under certain assumptions. From GPU perspective, the organization of the data is the main factor to ensure memory coalescing. Since G-80 class GPUs can load 4, 8 or 16 bytes words, the arrays of structures around these boundaries creates non-coalesced memory accesses. In order to prevent this problem, the data is arranged based on structure of arrays strategy that allows memory coalescing for any size of input.

MUMmerGPU (MUM): MummerGpu is a fast, low-cost application to align multiple query sequences (eg. DNA sequences) that generated in genotyping, genome resequencing, and metagenomics projects. The purpose of sequence alignment algorithm is to find similar regions between query sequence and reference sequence. Same as in the serial version, suffix tree is used to find exact alignments. The suffix tree encodes every suffix of a sequence on a unique path from root to a leaf. MummerGPU executes alignment kernel in parallel contrary to serial version of the algorithm. MummerGPU uses 2D texture cache to store suffix tree. It rearranges the nodes and its children relatively close in memory, which improves cache hit rate during sequence alignment. MummerGPU performs 10xs better than CPU when the input query size is than 800 bp.

Neural Network (NN): An artificial neural network is information process system that is inspired by how human brain is processing the information. This application implements the recognition of the handwritten digits with the help of convolutional neural networks. Bakhoda et al. [**20**] modifies this algorithm to enable multiple digit recognition. They simulate the

recognition of 28 digits from the Modified National Institute of Standards Technology database of handwritten digits. We used that version in our study.

Ray Tracing (RAY): is a popular technique for rendering images in computer graphics. It generates high –quality photos by simulating a variety of reflection and refraction of light. In CUDA implementation of RAY, each thread is responsible for one pixel of the image. We use 256 x 256 image for this study.

Scalar Product (SCP): Scalar Product is one of the common applications for parallel programming. This application is the implementation of dot product of two vectors on GPU. After product stage, accumulation is achieved by tree-based reduction. There are some restrictions on parameters to boost the performance. For instance, number of elements per vector should be a multiple of warp size to meet memory coalescing constraints. On G80 GPUs 24-bit multiplication takes 4 clocks per warp on the other hand, 32-bit multiplication takes 16 clocks per warp. The algorithm prefers using 24-bit version if operands fit into 24 bits. We use 256 vectors; each of them contains 4096 elements in this study.

JPEG Compression (JPEG): is a compression algorithm that reduces the file size without affecting image quality. This algorithm requires 4 steps: Divide input to 8 x8 blocks, DCT (Divine Cosine Transformation, quantization, and IDCT (Inverse DCT). In CUDA implementation, these four steps are covered with two methods. Encoding performs DCT and quantization, and decoding performs IDCT. Both encoding and decoding take a bmp file as input. Decoding first encodes the file on the CPU then decodes on the GPU.

Fast Walsh Transform (FWT): Fast Walsh Transform, also known as Hadamond Transform, is example of generalized Fourier transforms. It is applied in data encryption, data compression, and signal processing. This benchmark implements the efficient version of naturally ordered Walsh transform in CUDA and its application to dyadic convolution computation. It is a memory intensive benchmark that requires 64 MB input and output buffers.

3D Stencil (3DS): [**22**] 3D stencil refers to 3 input elements in each direction without counting the elements at the intersection. This algorithm processes on 3D volume by slicing into 2D thread blocks in order to increase the data reuse. 3D Stencil is used in seismic computing.

Computational Fluid Dynamics (CFD): [**23**] Computational Fluid Dynamics is a technique to compute the flow around, or through the objects. CFD is applied in diverse areas ranging from biology to astrophysics. The benchmark consists of a structured multi-block solver that divided into 3D hexahedral blocks. Mass, momentum, and energy equations are applied to each block to compute flow properties.

Separable Convolution (CONS): [**24**] This benchmark implements convolution filtering that is widely applied in image processing for smoothing and edge detection. The image convolution algorithm is suitable for the banked structure of the shared memory and memory coalescing. The amount of the shared memory affects the performance gain.

GUPS (Giga Updates per Second): Random memory accesses have a significant effect on the system performance. GUPS is a performance metric to measure the peak performance of the memory architecture of the system. It is calculated by dividing the number of randomly updated memory locations with one billion. An update refers to read-modify-write operation on 64-bit table. The memory size is generally chosen the power of two.

Histogram (HISTO): [**25**] Histogram is an application that calculates the frequency of occurrence of each element. It is commonly used in data mining and image processing. CUDA has two implementation of Histogram: Histogram64 and Histogram256. We use Histogram256 that provides higher precision with an implementation of 256-bin histogram. The input array is divided into sub-arrays; then each result of sub-array is stored in sub-histogram, and for final step the sub histograms are merged into a single histogram. Since multiple threads can share the same sub-histograms, this can lead to bank conflicts and intra-warp branching divergence based on the input value.

4.5.2 SHOC Benchmark Suite

Scalable Heterogonous Computing Benchmark (SHOC) [**11**] consists of benchmarks to measure the performance and stability of high performance computing architectures. It supports both OpenCl and CUDA to make a comparison between them. SHOC has two levels: First level benchmarks were implemented to measure low-level architectural features; other level includes common benchmarks from parallel processing. The benchmarks presented below taken from level one and two.

Reduction (RED): Reduction performs the sum of elements in a single-precision array or a double-precision array. While the input data is stored in global memory, the intermediate calculations are stored in the local shared memory to hide high memory latency.

Scan (SCAN): Scan also known as parallel prefix sum, is common algorithm in parallel computing. It takes an array as an input and outputs prefix sum based on array index. In GPU, the performance can degrade due to the shared memory bank conflicts. In order to prevent this, padding is added to the computed shared memory index. [**26**]

Quality Threshold Clustering (QTC): [**27**] The Quality Threshold clustering Algorithm is applicable in various fields such as mathematics, data mining, and chemistry. First, it introduced by Heyer at al. to measure clustering time to find co-regulated genes. Compared to k-means, the number of cluster (k) is not necessary for computation. Moreover, for given threshold parameter, it guarantees to meet that requirement. GPU implementation of QTC shows outer loop parallelism and the work in inner loop is memory bound. Since the matrices used during the algorithm are large-scaled, they need to be stored in global memory, which means high memory latency. In order to optimize latency, the global memory accesses are coalesced.

Triad STREAM (TRD): The STREAM Triad benchmark takes two floating-point vectors and a scalar value (A, B, s as respectively) as inputs and calculates sustainable memory

bandwidth ($MB = A + s * B$). The vector addition operation is achieved with no temporal reuse. Since data is stored in the global memory, memory operations are more expensive than vector addition. Therefore, memory operations dominate the performance.

4.5.3 Parboil Benchmark Suite

Parboil Benchmark Suite [**10**] consists of scientific benchmarks chosen from diverse range of application domains such as image processing, fluid dynamics, astronomy, and biology. These benchmarks emphasize on throughput computing by using dynamic task kernels, dense matrix operations and data-dependent memory accesses. The benchmarks we include in our framework are listed below:

Sum of Absolute Differences (SAD): In image processing, the input video frame needs to be compared with the several reference frames to find the most similar one. For this purpose, the sum of absolute differences benchmark is used. First, the frames are tiled into 4x4 blocks and the comparison between each pair is calculated. Then, the summation of these differences provides us the final result. SAD is a memory intensive benchmark that benefits from register tiling optimizations to exploit data reuse. In GPU implementation, coarsening is applied to reduce the texture cache accesses. For the calculation of larger blocks, grouping the vector additions on GPU reduces memory traffic.

Matrix Multiplication (MM): This dense matrix multiplication benchmark is implemented based on register tiling CUDA SGEMM code from Volkov []. Compared to Volkov's implementation, they introduce parameters to setup the degrees of register tiling and shared memory tiling to adapt MM implementation to the new generation of GPUs.

4.4.4 Rodinia Benchmark Suite

Rodinia Benchmark Suite developed by Che. et al. [**9**] includes benchmarks which are selected based on Berkeley's dwarf taxonomy. These benchmarks present different behaviors in terms of parallelism, memory- access patterns, and data sharing characteristics. The benchmarks used in this study are listed below:

Speckle Reducing Anisotropic Diffusion (SRAD): SRAD is an image processing algorithm that removes the noise in an image without changing important features of it. The computation is done by partial differential equations by using four neighbors of each pixel.

Hotspot (HS): Hotspot is a thermal modeling methodology for VLSI systems, developed by Huang et. al., measures processor temperature and power based on architectural floor plan. In GPU implementation of HS, the kernel-based calculations are done by applying differential equations on blocks of temperatures. Each output represents the average temperature of the cell on the chip-area.

Back Propogation (BP): This benchmark is an implementation of the neural network algorithm in which the data includes complex sensory input. The algorithm has two phases: First phase is called Forward phase, the input generates the output activations through the neural network. The backward phase estimates the error between the observed values and the activations. Then, the error value is propogated backwards to adjust the weights and bias values.

Needleman-Wunsch (NW): The Needleman-Wunsch algorithm is used in bioinformatics to align DNA sequences. This algorithm uses divide and conquer approach in which processes small block to construct the final solution. The full sequence is represented in 2-D matrix. There are two matrix constructed during the process. The first one is score matrix, which consists the value of maximum weighted path. The second one, the trace-back matrix,

helps us to find best alignment. The trace-back matrix is processed in diagonal strips for parallelism.

Benchmark	Abbr.	Block Dim	Grid Dim	Domain
Ray Tracing	RAY	128	512	Image Processing
JPEG Compression	JPEG	64	512	Image Processing
Neural Network	NN	168	169	Machine Learning
		1400	25	
		2800	1	
		280	1	
MUMmerGPU	MUM	256	196	Bioinformatics
Scalar Product	SCP	256	128	Mathematics
Hotspot	HS	256	1849	Physics Simulation
Histogram	HISTO	256	240	Data Mining
Separable Convolution	CONS	128	9216	Image Processing
		64	18432	
Computational Fluid Dynamics	CFD	192	1008	Computational Physics
Back Propogation	BP	256	4096	Pattern Recognition
Breadth-first Search	BFS2	512	63	Graph Algorithm
Black-Scholes Pricing	BLK	128	480	Finance
	GUPS	128	512	System Performance
3D Stencil	3DS	256	128	Seismic Computing
Fast Walsh Convolution	FWT	512	4096	Computational Mathematics
		256	128	
		256	8192	

Table 4-1- List of benchmarks used in the framework

Chapter 5

Experimental Setup and Results

5.1 Experimental Setup

Simulated System

For this study, we use GPGPU-Sim 3.2.2 simulator as mentioned in the previous chapter. We modified the simulator to support concurrent application execution. Table 2 shows the hardware configuration we used for GPGPU-Sim. We use the default parameters that are based on NVIDIA GeForce GTX 480 Fermi architecture.

GPU Core Config.	30 Shader Cores, 1400MHz, SIMT Width = 16 x2
GPU Resources/ Core	Max. 1536 Threads (48 warps, 32 Threads/warp), 48 KB Shared Memory, 32684 Registers
GPU Caches / Core	16 KB 4-way L1 Data Cache, 12KB 24-way Texture, 8KB 2-way Constant Cache, 2KB 4-way I-cache, 128B Line Size
Features	Memory Coalescing, Inter-warp Merging, Post Dominator
Interconnect	1 crossbar/direction (30SMs/ 6 Memory Controllers), 1400MHz
Memory Model	6 GDDR5 Memory Controllers (MCs), FR-FCFS scheduling, 16 DRAM-banks/MC, 924 MHz memory clock

Table 5-1: Simulated baseline GPU configuration

Application Classification

We classify the applications based on their memory sensitivity values measured by their MPKI (Misses per Kilo instructions) using 15 SMs of 30 SMs in the system. Table 1 shows the classification of GPU applications based on their MPKI values. Benchmarks with less than and equal to 1 MPKI are considered as low memory sensitive (Type L); benchmarks with less than and equal 3 MPKI are considered as medium memory sensitive (Type M) and the others are classified as high memory sensitive (Type H).

We form two-application workloads from applications listed in Table 1. We simulate all workloads till completion since some applications include multiple kernels. Each kernel of these applications can show different behavior that will lead us to inaccurate analysis. Simulating till completion ensures us to observe all phases of the application. We evaluate each workload using three core-partitioning schemes: 10-20, 15-15 and 20-10. We do not scale the L2 Cache partitioning based on core-partitioning schemes. Regardless of the core-partitioning scheme, each application receives the half of L2 cache.

Benchmarks	MPKI	Type
GUPS	30.294	Type H
MUM	22.278	Type H
SCP	2.683	Type M
FWT	2.155	Type M
BFS2	2.08	Type M
BLK	1.596	Type M
3DS	1.168	Type M
JPEG	1.092	Type M
NN	0.208	Type L
RAY	0.158	Type L

Table 5-2: GPU Applications: L2 MPKI and Classification

5.2 Evaluation Metrics

We report the performance of the each workload and their interactions to each other in two different categories. In this section, we present the metrics used in the experiment.

Instruction Throughput (Eq. 4) is the sum of raw IPCs, and is commonly used for the evaluation of multi-application system. However, recent studies [**8**,**28**] prefer weighted speedup due to fairness and consistency. They address that raw IPC can unfairly favor high-IPC applications. The weighted speedup (Eq.1) is defined as the sum of per application slowdowns experienced during concurrent execution with respect to alone execution on entire system. Weighted speedup can be interpreted as system-level throughput (STP), which defined as the average number of jobs finished per unit of time. The maximum value of WS depends on the number of applications that run concurrently (e.g. for 2 application, it will be 2). We provide both metrics to discuss which one of them is more suitable for our study. We measure Average Normalized Turnaround Time (ANTT) (Eq.3), which defined as the arithmetic average across the slowdown of turnaround time per application in multi-application workload. Turnaround time is execution time of the application. ANTT indicates the amount of the interaction among applications. Thus, lower ANTT is better. Fairness index (FI) (Eq. 4) presents the maximum ratio of slowdowns between two applications. The system is fair when FI is equal to 1.

$$Slowdown(APP_i) = \frac{IPC_i^{shared}}{IPC_i^{alone}}$$

Eq. 1 $$STP\ (Weighted\ speedup) = \sum_i Slowdown(APP_i)$$

$$HWS(APP_i) = \frac{nApp}{\sum_i \frac{1}{STP_i}}$$

Eq. 2 $$ANTT = \frac{1}{\sum_i HWS(APP_i)}$$

Eq. 3 $$Fairness\ Index\ (FI) = MAX\left(\frac{Slowdown(APP_1)}{Slowdown(APP_2)}, \frac{Slowdown(APP_2)}{Slowdown(APP_1)}\right)$$

Eq. 4 $$Instruction\ Throughput(IT)) = \sum_i IPC_i$$

Table 5-3: Evaluation Metrics

5.3 Experimental Results

5.3.1 Bandwidth Utilization

Figure 5-1 presents the memory bandwidth components of several workloads chosen from different categories. The memory bandwidth components are listed as follows:

- App1 and App 2 BW: The percentage of DRAM cycles used moving data over DRAM interface with respect to App1 and App2.
- Wasted BW: The percentage of DRAM cycles in which there is a pending request in the memory request queue, but no data is transferred on DRAM.
- Idle-BW: The percentage of DRAM cycles when DRAM is idle.

We plot the concurrent execution of two-application workloads for 3 different core configurations. We also provide memory bandwidth values for single application execution to understand how the applications interfere with each other during the execution. For instance, Alone1_10, Alone1_15 and Alone1_20 show the single APP1 executes by using 10 SMs of 30 SMs system, 15 SMs of 30 SMs system and 20 SMs of 30 SMs of system respectively.

As shown in Figure 5-1, MUM, GUPS, BLK, and FWT have high attained bandwidth. When two of these applications execute together (MUM_GUPS and BLK_FWT), there will not be crucial change in bandwidth utilization through different core configurations. As one application gains, the other one will lose due to the race between applications for the available bandwidth. However, for NN_GUPS, GUPS will benefit from using more cores than NN due to the low bandwidth utilization of NN. In Figure 5-1, RAY is co-scheduled with BLK and BFS2. RAY has a scalable bandwidth utilization based on assigned number of cores. For stand-alone execution, it has 60% and 37% of bandwidth utilization with 10 cores and 20 cores, respectively. Thus, assigning more cores to RAY will decrease the negative impact on the co-scheduled application. These observations show that high bandwidth applications such as GUPS, MUM (Table 5-2) can cause the co-scheduled application in negatively and dominate the overall DRAM bandwidth utilization.

Figure 5-1 also confirms the necessity of concurrent application execution. If the single application itself serves a huge portion of Idle BW, the concurrent execution of that application improves the DRAM bandwidth utilization as seen in BLK_BFS2 and NN_GUPS. In the case of BLK_BFS2, BFS2 has 60% of idle DRAM cycles during single execution. By scheduling with BLK, the idle DRAM cycles becomes 0.006%.

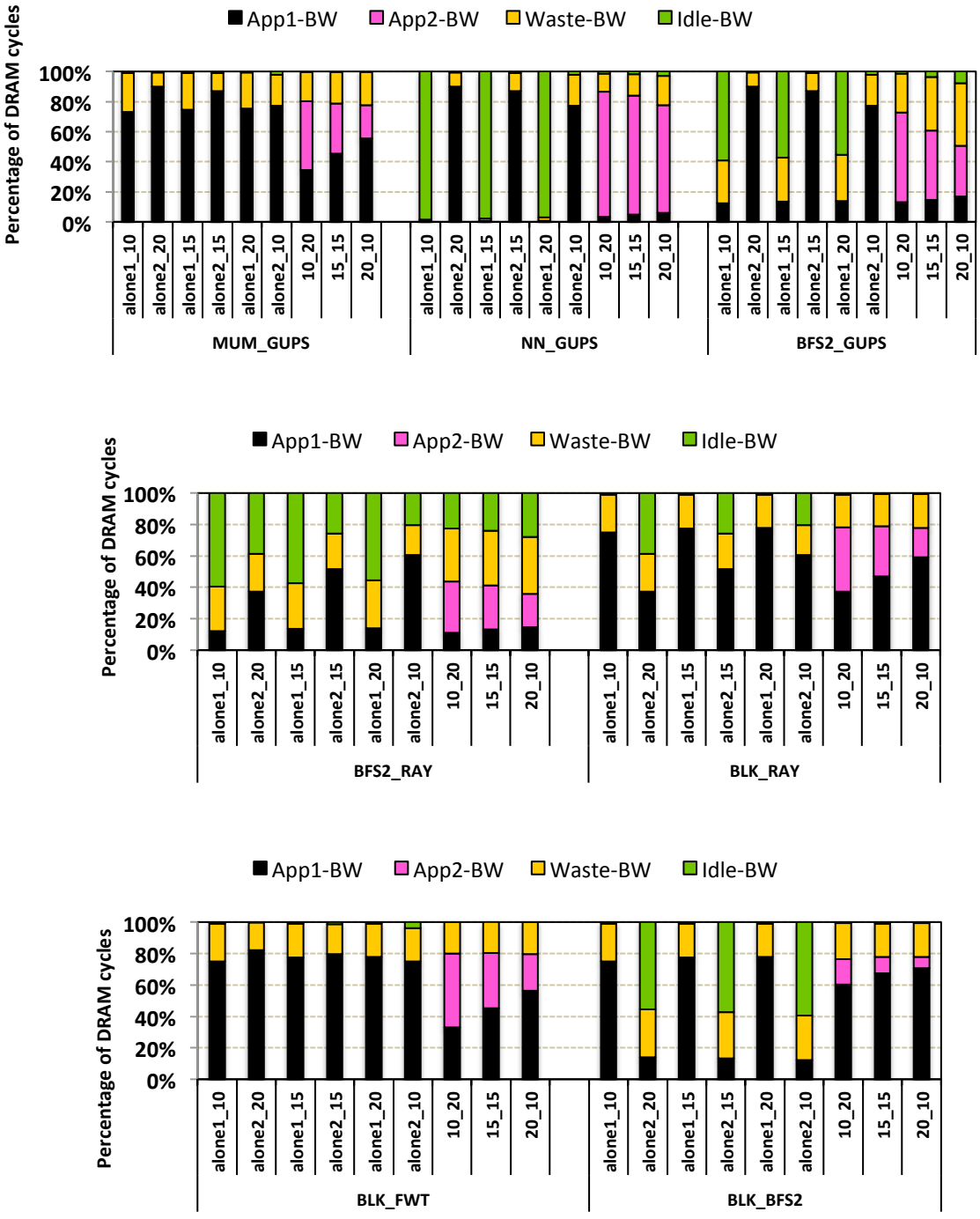


Figure 5-1: DRAM bandwidth utilization distribution across several workloads for different core configurations.

5.3.2 System Throughput

Co-scheduling Type H applications

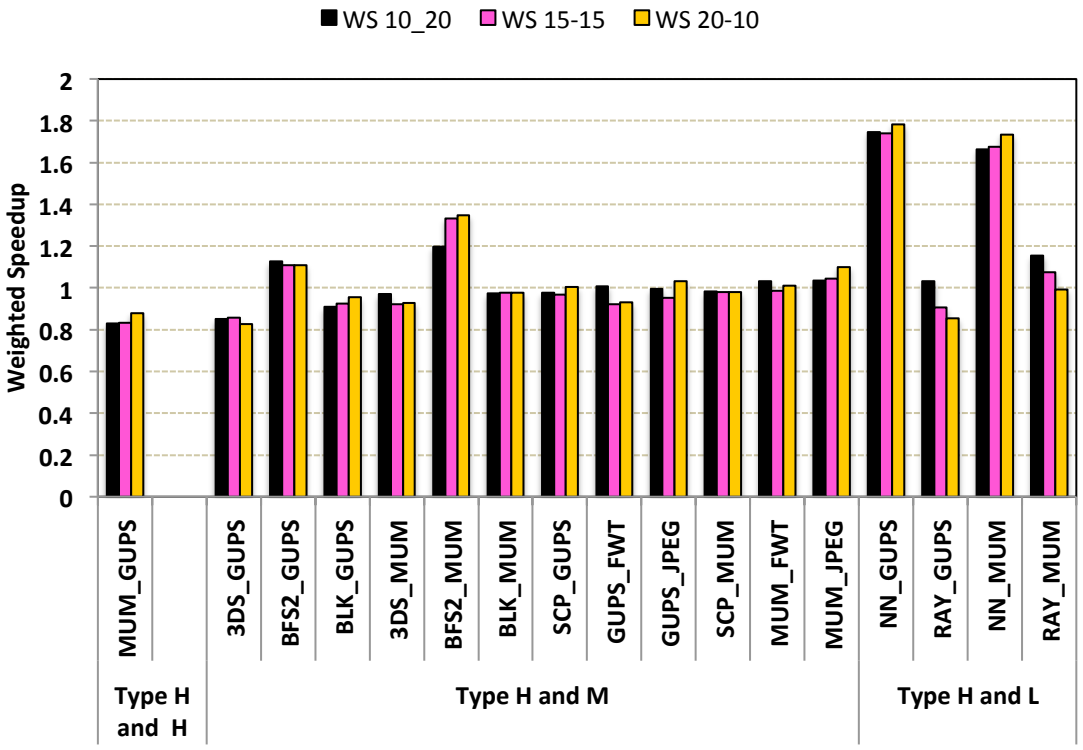


Figure 5-2: Weighted speedup for the workloads in which one application is Type H

Figure 5-2 shows WS of three core partitioning schemes (10-20, 15-15 and 20-10), using workloads in which App1 chosen from Type High MPKI. For App2, we have 3 cases: Type High MPKI (MUM), Type Medium MPKI (FWT, BLK, SCP, 3DS, JPEG, and BFS2) and Type Low MPKI (RAY, NN).

In case 1, we examine the performance of MUM-GUPS. In general, we expect that assigning more cores to lower MPKI application will boost the weighted speedup. MUM has

lower MPKI than GUPS as shown in table 5-2. Thus, scheme 20-10 shows better speedup comparing to others.

In case 2, considering only MPKI values of applications are not enough to observe the trend of medium MPKI applications. Due to the high variance between bandwidth utilizations of the applications, we also add this metric into our discussion. When we co-schedule GUPS or MUM with BLK or FWT, the lower MPKI application is favorable since they have close attained bandwidth. For SCP-MUM and SCP-GUPS, the performance values are close for all core-partitioning schemes and assigning more cores to lower bandwidth application is preferable. For the remaining applications, their bandwidth utilization is scalable based on assigned core number. (Table 5-2) We give individual slowdowns for these applications when they are co-scheduled with MUM or GUPS in Figure 5-3.

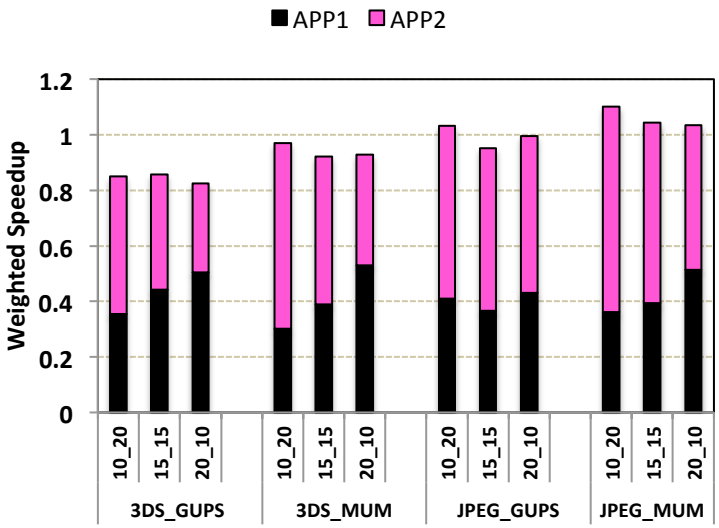


Figure 5-3: Weighted speedup of several workloads with individual slowdowns

As expected, individual speedup of lower MPKI application increases while the number of assigned cores increase. However, assigning more cores causes dramatic change in bandwidth utilization of the application (e.g. JPEG for 10 cores, BW = 0.55 and for 20 cores BW=0.72).

This change implies an important amount of performance degradation in individual speedup of GUPS or MUM. Thus, assigning more cores to GUPS or MUM provides better system performance.

In case 3, we examine the co-scheduling of Type-H MPKI application with Type-L MPKI application. As discussed before, low MPKI application benefits from more number of cores. NN follows this trend and achieves the best performance compared to other workloads when it is co-scheduled with MUM or GUPS. Moreover, NN does not hurt the performance of co-scheduled application due to its low bandwidth utilization. We cannot expect the same thing for RAY, because of the scalable bandwidth utilization based on the number of cores. Like applications 3DS and JPEG, explained in Case 2, RAY interferes the co-scheduled application in negative way so assigning fewer cores achieves better overall WS.

Co-Scheduling Type-M applications

MPKI values of Type-M applications are listed in Table 5-2. Bandwidth utilization of these applications is high in general except BFS2. There are two cases we need to consider: Type-M with Type-L and Type-M with Type-M. In the first case, higher number of cores should be assigned to Type-L application if the bandwidth utilization of that application is low like NN. For workloads that include RAY, the opposite situation occurs. As the assigned core number increases for RAY, it interferes the Type-M application more due to the bandwidth utilization.

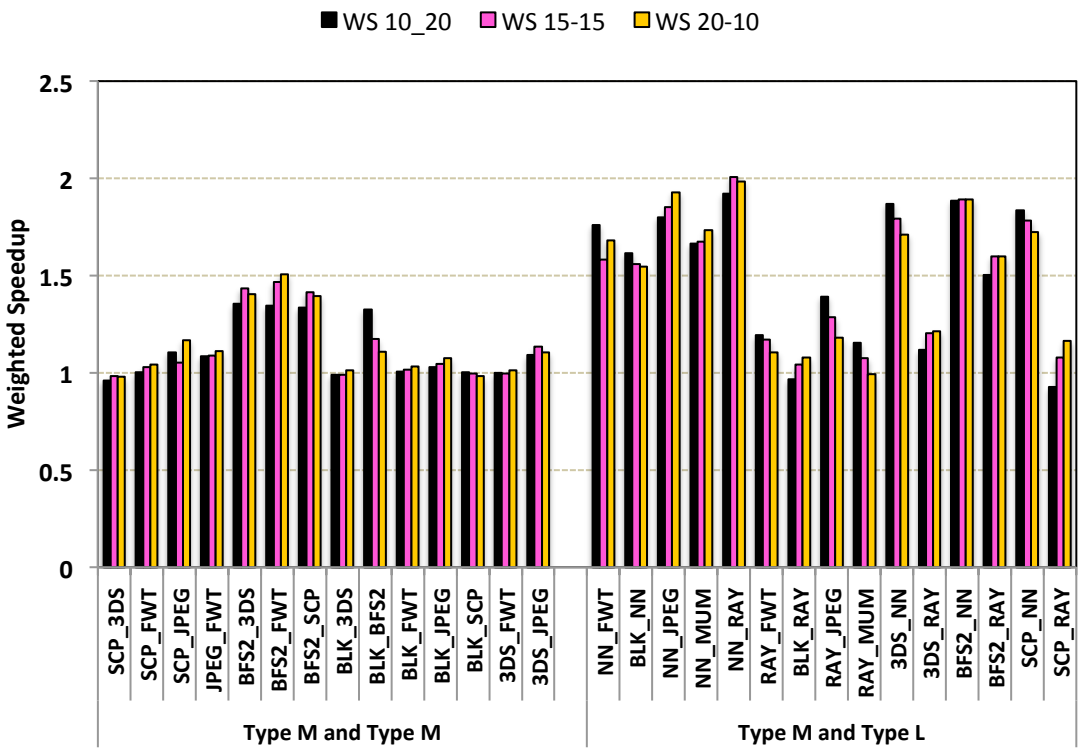


Figure 5-4: Weighted speedup for the workloads in which one application is Type M

When both applications in a workload are Type M, we need to take into account both MPKI and bandwidth utilization. For instance, JPEG-3DS performs better with even core partitioning compared to other partitioning schemes, because of the similar MPKIs and bandwidth utilizations. If we rank the applications by their bandwidth utilizations, SCP has the highest nearly 82% and BFS2 has the lowest 13%. Due to the L1 cache, BFS2 gains improvements from more resources when it is co-scheduled with the other applications. BFS2 is a very L1 cache sensitive application.

Co-Scheduling Type L applications

In our classification, there are only two applications which have low MPKI values: RAY and NN. To analyze this case in details, we also include SAD and HS, which have 0.119 and 0.319 MPKI values respectively. Figure 5-5 shows the system throughput obtained from scheduling two Type L applications together. All the applications have low MPKI values, so we need to consider bandwidth utilization of applications while assigning cores. Assigning more cores to the application with lower bandwidth compensates the performance loss incurred from the bandwidth contention.

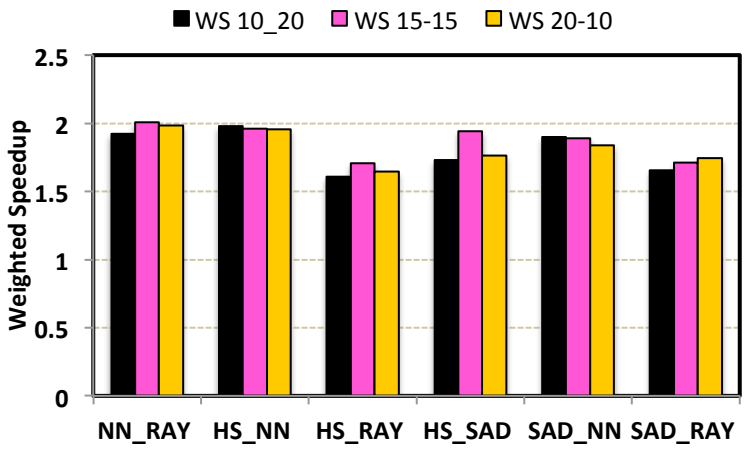


Figure 5-5: Weighted speedup for the workloads in which both applications are Type L

NN has the lowest the attained bandwidth and it does not degrade the co-scheduled application's performance much. Thus, the overall system throughput for workloads that include NN, is higher compared to other two-application workloads. As mentioned before, RAY has a scalable bandwidth utilization based on number of assigned cores. For instance in SAD_RAY workload, when RAY gets 20 cores, it hurts SAD so much due to RAY's bandwidth utilization becomes 67%.

5.3.3 Average Normalized Turnaround Time

Figure 5-6 presents the average normalized turnaround times (ANTT) of workloads for with three core-partitioning schemes. ANTT refers to the amount of interaction between applications. In ideal case, when application do not interfere each other, ANTT is 1. A smaller value of ANTT will be advantageous for the system performance. Almost %75 of the workloads prefers the same core-partitioning scheme as in the weighted speedup figure. That means, the core-partitioning scheme gives the highest weighted speedup and also ensures the shortest time. However, when we examine the workloads that include GUPS, STP and ANTT do not favor the same core-partitioning scheme. When GUPS uses more cores than the co-scheduled application, the other application cannot satisfy its memory requests quickly due to the bandwidth usage of GUPS. Thus, it increases the overall execution time. Same reason can be given in order to consider the behavior of BLK-FWT and SCP-FWT workloads since they both have high bandwidth utilization.

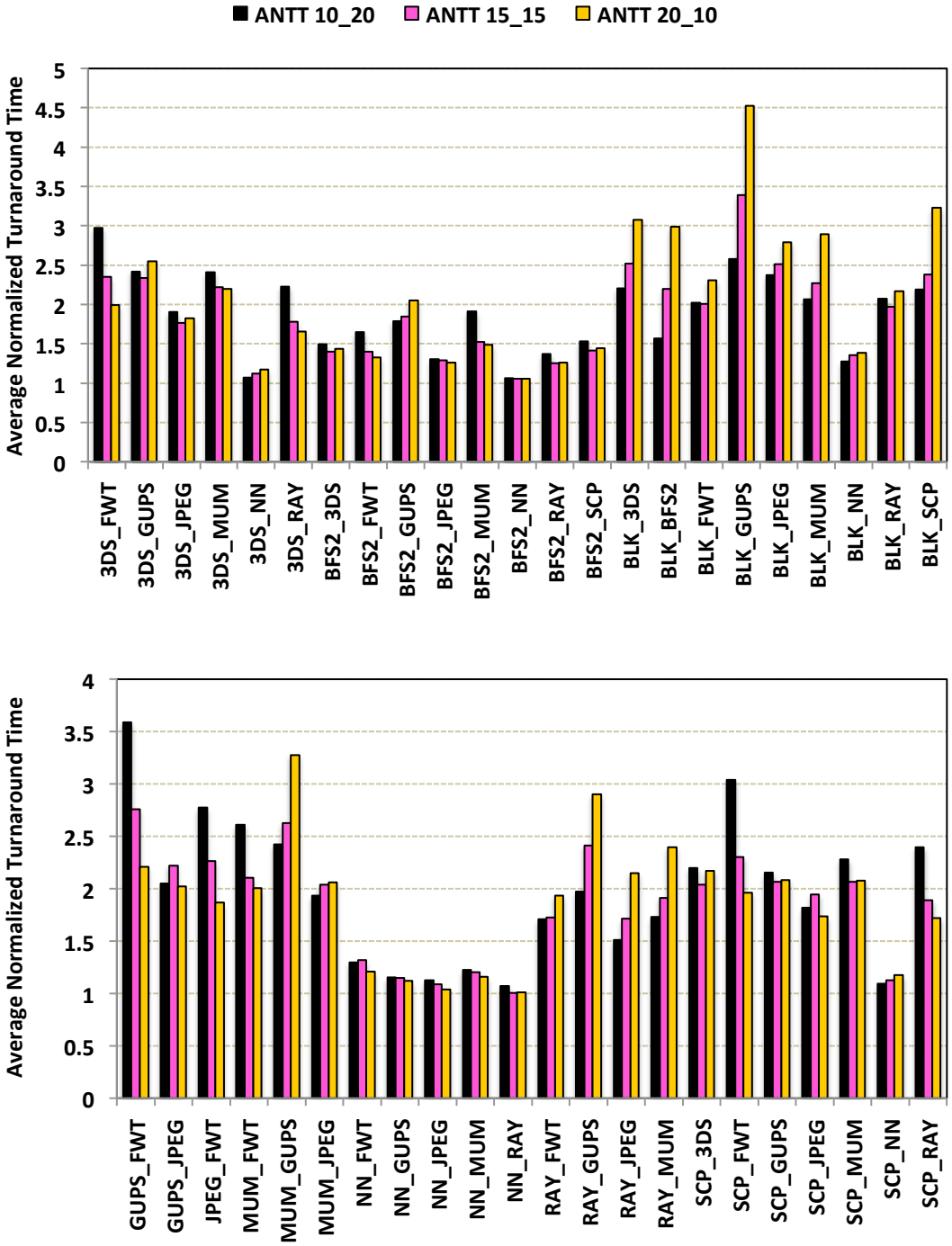


Figure 5-6: Comparison of ANTT for three core partitioning schemes.

5.3.4 Instruction Throughput

Instruction throughput (IT) is defined as the total number of committed per cycle. Table 5-4 gives the ITs for single application execution when 15 SMs of entire system assigned to that application.

Benchmark	IT
RAY	461.3289
3DS	440.2151
BLK	342.5075
JPEG	309.9041
SCP	305.604
FWT	235.7992
NN	37.7007
MUM	31.3921
GUPS	29.0296
BFS2	20.1847

Table 5-4: IT values of applications for 15 SMs

To show how instruction throughput is affected in concurrent execution, we conduct experiments for three core-partitioning schemes. Figure 5-7 presents the results of these experiments. The major observation is that the application that has more IPC in single application execution takes advantage of more resources compared to co-scheduled application. For instance, BFS2 and RAY have IT 20.19 and 461.33, respectively. When we check the throughput of workload BFS2-RAY, it decreases as we assign more cores to BFS2. For BFS2-GUPS, the instruction throughput is not changing through the different SM partitioning schemes due to the similar IT values. 3DS-RAY shows the highest throughput among the other workloads since both have high IT values. We conclude that the instruction throughput of co-scheduled applications is proportional to their single IT values.

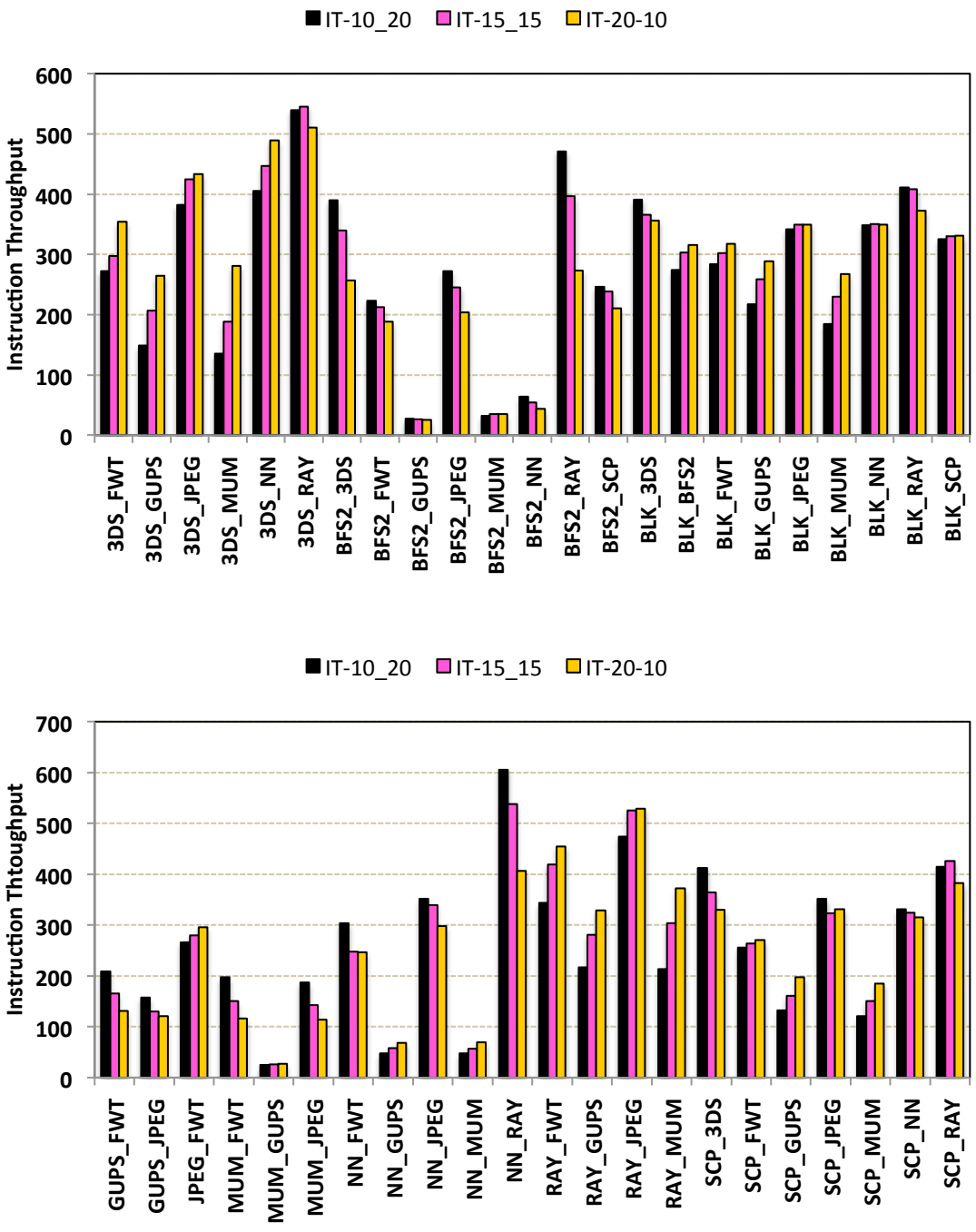


Figure 5-7: Comparison of instruction throughput (IT) for evaluated workloads

5.3.5 Fairness Index

Fairness index is used to measure if each of co-running application in the workload makes equal progress compared to their alone executions. It is a crucial metric for multi-application workloads due to the shared resources. Based on application characteristics assigning more resources to a specific application can lead performance imbalance between applications and also influence the overall system performance. Figure 5-8 compares the Fairness index (FI) of workloads for different core partitioning schemes. For some applications like NN, the fairness is balanced in most of the workloads. The applications that have similar bandwidth utilizations like SCP-GUPS benefit more from partitioning resources evenly. RAY's bandwidth utilization changes based on the core allocation, so assigning fewer cores to RAY has a positive impact on the co-scheduled application from the fairness perspective. BLK-GUPS presents the highest fairness index among all workloads, nearly 6.5 for core partitioning 20-10. Since GUPS is the most memory intensive application, having fewer resources compared the co-scheduled application interferes GUPS more. Moreover, BLK shows the same behavior as GUPS. Assigning more cores to BLK creates more memory contention. Assigning fewer cores to BLK ensures fair SM allocation.

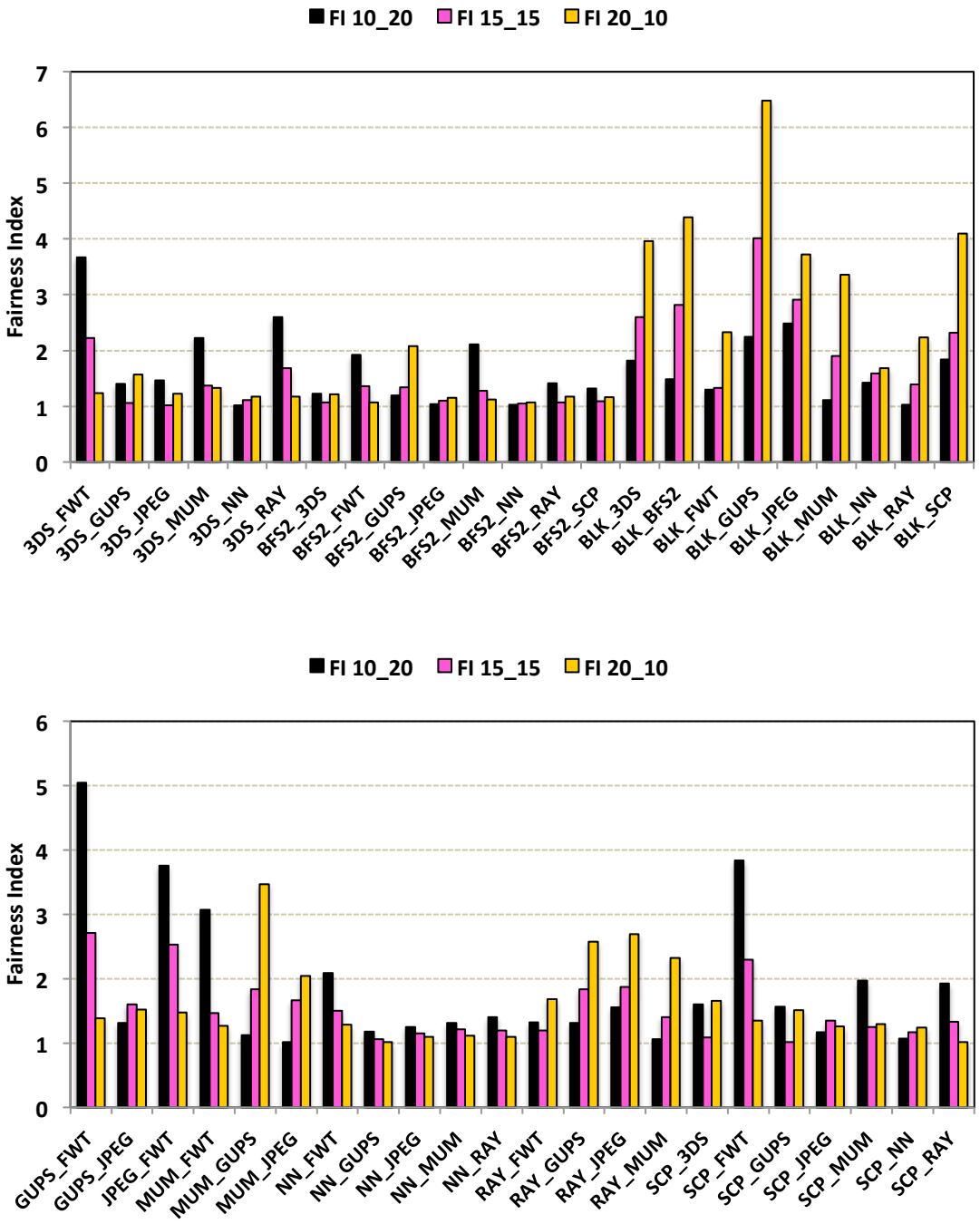


Figure 5-8: Comparison of FI based on several core partitioning schemes

Chapter 6

Conclusion and Future Work

In this thesis, we explore the application interference of GPU applications during concurrent execution. We propose a flexible and adaptable framework that supports 25 applications from different benchmark suites. We characterize the applications based on their memory sensitivity by using MPKI metric. Then, we conduct experiments for three different SM partitioning schemes: 10-20, 15-15, and 20-10. Experiment results show that MPKI is not enough to determine the best core-partitioning configuration. Thus, we also consider the attained bandwidth of each application during evaluation. We use different metrics, bandwidth utilization, WS, ANTT, FI and IT, to understand whether they benefit from the same core-partitioning scheme. However, all metrics do not favor the same SM partitioning scheme. Therefore, the performance trade-offs need to be considered based on the metric. One of the major observations is that the application with high bandwidth hurts the performance of co-scheduled application for nearly all metrics.

For future work, we plan to investigate a dynamic SM and cache partitioning in multi-application execution domain. Moreover, we are planning to extend the application suite by adding some applications from popular domains such as cloud computing. We believe that this framework will be helpful for further research studies in this area.

Bibliography

[1] NVIDIA. NVIDIA CUDA. [Online]. http://www.nvidia.com/object/cuda_home_new.html

[2] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture. [Online]. http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf

[3] NVIDIA. (2012) NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. [Online]. http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf

[4] Top 500. [Online]. http://www.top500.org/lists/2014/11/

[5] Google Project Tango. [Online]. https://www.google.com/atap/projecttango/#project

[6] Adwait Jog et al., "Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications," in *ASPLOS GPGPU Workshop* , 2014, p. 8.

[7] Onur Kayiran et al., "Manaing GPU Concurrency in Heteroeneous Architectures," in *MICRO*, 2014.

[8] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU Concurrency with Elastic Kernels," in *ASPLOS*, New York, NY, 2013.

[9] Shuai Che et al., "Rodinia : A benchmark suite for heterogeneous computing," in *IISWC*, 2009.

[10] John A. Stratton et al., "Parboil: A revised Benchmark Suite for Scientific and Commercial Throughput Computing," 2012.

[11] Anthony Danalis et al., "The Scalable Heteogeneous Computing Benchmark Suite," in *GPGPU*, 2010, p. 12.

[12] Marisabel Guevara, Chris Gregg, Kim Hazelwood, and Kevin Skadron, "Enabling Task Parallelism in the CUDA Scheduler," in *Workshop on Programming Models for Emerging Architecture (PMEA)*, 2009.

[13] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron, "Fine-Grained Resource Sharing for Concurrent GPGPU Kernels," in *HotPar*, 2012.

[14] Lingyuan Wang, Miaoqing Huang, and T. El-Ghazawi, "Exploiting concurrent kernel execution on graphic processing units," in *HPCS*, Istanbul, 2011.

[15] Guibin Wang, YiSong Lin, and Wei Yi, "Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU," in *GREENCOM-CPSCOM*, 2010.

[16] Jacob T. Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte, "The case for GPGPGU Spatial Multitasking," in *HPCA*, New Orleans,LA, 2012.

[17] Onur Kayiran, Adwait Jog, Mahmut T. Kandemir, and Chita R. Das, "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs ," in *PACT*, 2013.

[18] Adwait Jog et al., "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.

[19] Adwait Jog et al., "Orchestrated Scheduling and Prefetching for GPGPUs," in *ISCA*, 2013.

[20] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, Boston, 2009

[21] Blaise Barney. POSIX Threads Programming. [Online].

https://computing.llnl.gov/tutorials/pthreads/

[22] Paulius Micikevicius, "3D Finite Difference Computation on GPUs using CUDA," NVIDIA, 2009.

[23] Graham Pullan and Tobias Brandvik. Computational Fluid Dynamics. [Online]. http://www.many-core.group.cam.ac.uk/projects/CFD.shtml

[24] Victor Podlozhnyuk, "Image Convolution with CUDA," NVIDIA, 2012.

[25] Victor Podlozhnyuk, "Histogram calculation in CUDA," NVIDIA, 2012.

[26] Mark Harris. (2007, Feb.) Parallel Prefix Sum with CUDA. [Online]. http://beowulf.lcs.mit.edu/18.337-2008/lectslides/scan.pdf

[27] Anthony Danalis, Collin McCurdy, and Jeffery S. Vetter, "Efficient Quality Threshold Clustering for Parallel Architectures," in *IPDPS*, 2012, p. 12.

[28] Stjin Eyerman and Lieven Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," in *IEEE MICRO*, 2008.