12-1-2017

THROUGHPUT OPTIMIZATION AND RESOURCE ALLOCATION ON GPUS UNDER MULTI-APPLICATION EXECUTION

SRINIVASA REDDY PUNYALA
Southern Illinois University Carbondale, srinu4@outlook.com

THROUGHPUT OPTIMIZATION AND RESOURCE ALLOCATION ON GPUS UNDER MULTI-APPLICATION EXECUTION

by

SRINIVASA REDDY PUNYALA

B.S., Jawaharlal Nehru Technological University Hyderabad, 2015

A Thesis
Submitted in Partial Fulfillment of the Requirements for the
Master of Science Degree

Department of Electrical and Computer Engineering
in the Graduate School
Southern Illinois University Carbondale
December 2017

THESIS APPROVAL

THROUGHPUT OPTIMIZATION AND RESOURCE ALLOCATION ON GPUS UNDER MULTI-APPLICATION EXECUTION

By

SRINIVASA REDDY PUNYALA

A Thesis Submitted in Partial

Fulfillment of the Requirements

for the Degree of

Master of Science

in the field of Electrical and Computer Engineering

Approved by:

Dr. Iraklis Anagnostopoulos, Chair

Dr. Arash komaee

Dr. Dimitrios Kagaris

Graduate School
Southern Illinois University Carbondale
October 31, 2017

AN ABSTRACT OF THE THESIS OF

SRINIVASA REDDY PUNYALA, for the for the Master of Science degree in Electrical and Computer, presented on October 31, 2017, at Southern Illinois University Carbondale.

TITLE: THROUGHPUT OPTIMIZATION AND RESOURCE ALLOCATION UNDER MULTI-APPLICATION EXECUTION ON GPUs

MAJOR PROFESSOR: Dr. I. Anagnostopoulos

Platform heterogeneity prevails as a solution to the throughput and computational challenges imposed by parallel applications and technology scaling. Specifically, Graphics Processing Units (GPUs) are based on the Single Instruction Multiple Thread (SIMT) paradigm and they can offer tremendous speed-up for parallel applications. However, GPUs were designed to execute a single application at a time. In case of simultaneous multi-application execution, due to the GPUs' massive multi-threading paradigm, applications compete against each other using destructively the shared resources (caches and memory controllers) resulting in significant throughput degradation. In this thesis, a methodology for minimizing interference in shared resources and provide efficient concurrent execution of multiple applications on GPUs is presented. Particularly, the proposed methodology (i) performs application classification; (ii) analyzes the per-class interference; (iii) finds the best matching between classes; and (iv) employs an efficient resource allocation. Experimental results showed that the proposed approach increases the throughput of the system for two concurrent applications by an average of 36% compared to other optimization techniques, while for three concurrent applications the proposed approach achieved an average gain of 23%.

DEDICATION

I dedicate this work to my family who made this possible.

ACKNOWLEDGMENTS

I would like to thank Dr. Iraklis Anagnostopoulos for his invaluable assistance and insights leading to the writing of this paper.

TABLE OF CONTENTS

Chapter **Page**

LIST OF TABLES

Table **Page**

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Graphics Processing Units (GPUs) are co-processors designed for rendering 2-dimensional and 3-dimensional graphics. Graphics workloads have abundant parallelism. GPUs utilize the available parallelism to accelerate graphics rendering. It did not take too long for programmers to realize that this computational power can also be used for tasks other than graphics rendering. Since 2003, many data parallel workloads have been ported to GPUs. Back then, there was no programming model for general-purpose tasks on GPUs. So, all the workloads had to be expressed in terms of graphics with pixels and vectors. This is because GPU pipeline in the beginning, was tightly bonded to the requirements of graphics applications. The programming paradigm shifted when the two main GPU manufacturers, NVIDIA and AMD, changed the hardware architecture from a dedicated graphics-rendering pipeline to a multi-core computing platform.

Graphics processing units have evolved to co-processors of a size larger than typical CPUs. While CPUs use large portions of the chip area for caches, GPUs use most of the area for arithmetic logic units (ALUs). The main concept GPUs use to exploit the computational power of these ALUs is executing a single instruction stream on multiple independent data streams (SIMD). This concept is known from CPUs with vector registers and instructions operating on these registers. For example, a 128-bit vector register can hold four single-precision floating-point values; an addition instruction operating on two such registers performs four independent additions in parallel. Instead of using vector registers, GPUs use hardware threads that all execute the same instruction stream on different sets of data. This approach is termed as Single Instruction Multi Thread (SIMT). The number of threads required to keep the ALUs busy is much larger than the number of elements inside vector registers on CPUs. GPU performance therefore relies on a high degree of data-level parallelism in the application. To alleviate these require-

ments on data-level parallelism, GPUs can also exploit task-level parallelism by running different independent tasks of a computation in parallel. This is possible on all modern GPUs through the use of conditional statements.

1.1 GPU COMPUTING

With the generalization of GPU architecture and hardware pipeline more and more general purpose problems have been ported to GPUs. General purpose workload has sequential and parallel parts. The sequential part is executed on the CPU and the parallel part, also known as kernel, of the problem is offloaded to GPU as shown in Figure 1.1. However, it is then observed that the general purpose workload does not have enough parallelism to exploit all the available resources on the GPU. Task level parallelism prevails as a solution, where multiple kernels can be offloaded onto a single GPU. These kernels can be launched from the same context or from multiple contexts. Each of the independent kernels again needs to involve a relatively high degree of data-level parallelism to make full use of the computational power of the GPU.

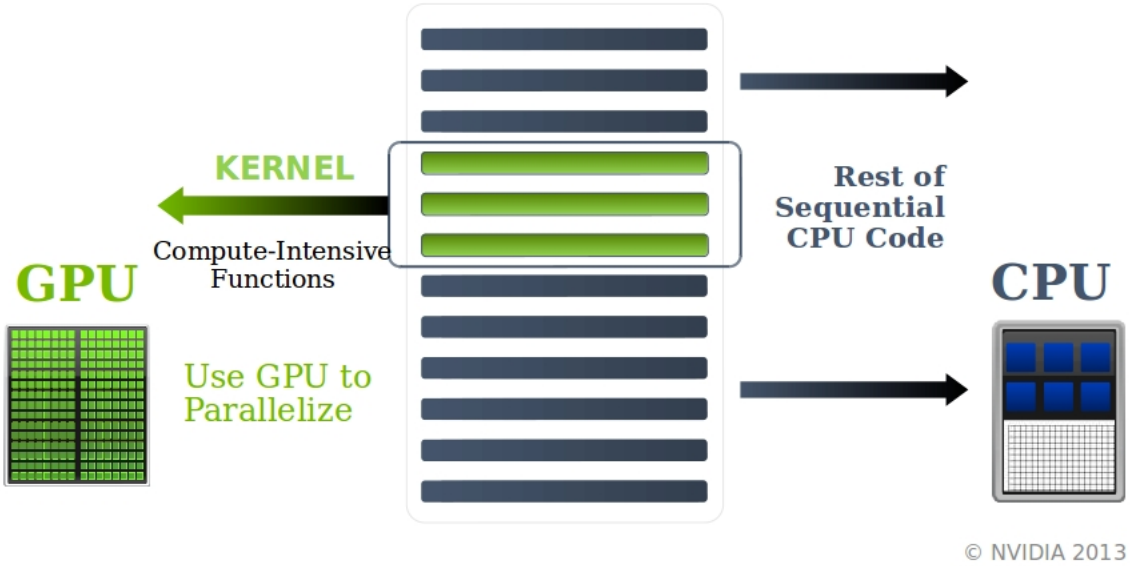


Figure 1.1: flow of execution of general purpose workload [2]

1.2 DEFINITIONS

1.2.1 Throughput

Throughput of a device is defined as the number of instructions executed in the total number of cycles simulated. The mathematical representation of Throughput is shown in the Equation 1.1

$$T = \frac{\sum_{n=1}^{k} I_n}{\sum_{n=1}^{k} C_n} \tag{1.1}$$

1.2.2 Utilization

Utilization is the measure of device occupancy achieved by an application. We measure utilization by comparing throughput of an application with the maximum throughput that can be achieved on the Device.

1.2.3 Streaming Multiprocessor(SM)

Each processor in a GPU is called a Streaming Multiprocessor. Each SM contains mutliple processing elements called as Streaming Processors as called by NVIDIA or Compute Units interms of AMD. Each SM also contains some Special Function units that execute FMA instructions.

1.3 MOTIVATION

The exploitation of task-level parallelism gives the programmer more flexibility and extends the set of applications that can make use of GPUs to accelerate computations. However throughput of the device depends on the application that are used to achieve task level parallelism. Improper selection of these applications can cause damage to the throughput of the GPU. This creates an interesting area of research, through which an efficient method of selecting applications for task level parallelism can be formalized.

Also every general purpose application may not have enough parallelism and may

not fully utilize the resources available to it. We simulated general purpose workloads from rodinia [1] benchmark suite on GPGPU-Sim with NVIDIA GTX-480 architecture. Figure 1.2 shows the utilization levels of different benchmarks we used. It is clear now that there is plenty of space for task level parallelism and resource allocation on GPUs.

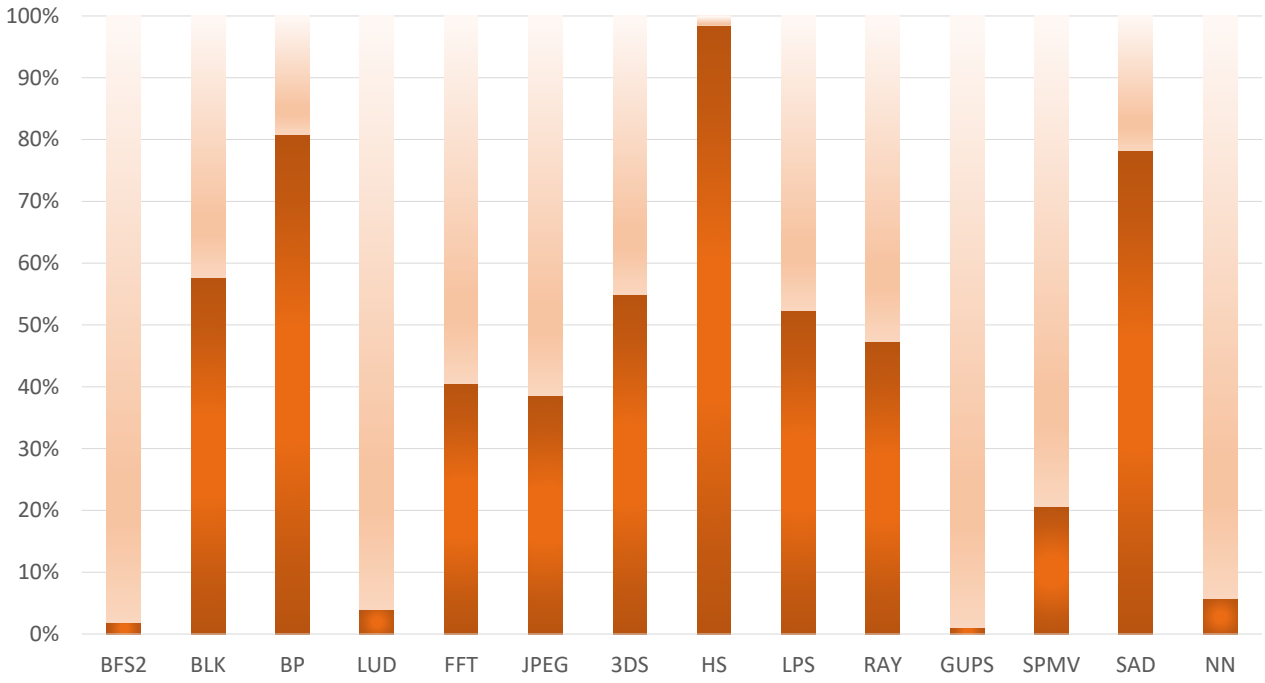


Figure 1.2: Max utilization of rodinia [1] Benchmarks

Multiple application execution can be performed in a *temporal* or *spatial* way. The temporal approach uses time multiplexing in order to allocate resources to different users and it is the common technique in GPU virtualization [6]. However, this approach leads to system underutilization and poor performance [7]. Unlike CPUs, where multi-application is architecturally supported concurrent execution of multiple applications on GPUs prevails as challenge in order to unlock system's performance [7, 8, 9]. Due to massive application parallelism and numerous generated threads, GPUs' performance is affected by contention on shared resources in multiple ways. Streaming Multiprocessors (SMs) are independent processing elements but share resources, such as caches and memory controllers. Threads, when running simultaneously, compete against each other using

destructively the shared resources [3].

1.4 INTEGER LINEAR PROGRAMMING

ILP is a mathematical approach to obtain the best result (maximum productivity and least resource consumption) when the problems can be expressed in linear functions. Integer Linear Programming is a branch of mathematical programming (mathematical optimization). In simple terms, linear programming is an optimizing technique for linear objective function, subject linear equality and inequality constraints. Solution region for these functions is a convex polytype. This region is defined as the intersection of many finite half spaces, each of which is defined by a linear inequality.Its objective function is a real-valued affine (linear) function defined on this polyhedron. A linear programming algorithm finds a point in the polyhedron where this function has the smallest (or largest) value if such a point exists.

e.g. problem

$$maximize f(x1, x2) = c_1 x_1 + c_2 x_2 \tag{1.2}$$

Subject to constraints

$$a_1 1 x_1 + a_1 2 x_2 \leq b_1 \tag{1.3}$$

$$a_2 1 x_1 + a_2 2 x_2 \leq b_2 \tag{1.4}$$

$$a_3 1 x_1 + a_3 2 x_2 \leq b_3 \tag{1.5}$$

In the above example problem, equation 1.2 is a function to be maximized subject to constraints shown in the equations below it. ILP forms a polygon using these constrains and chooses a value within the polygon that yields the maximum result for the function f.

1.5 CONTRIBUTION

In this thesis, we present a methodology for efficient concurrent execution of multiple applications on GPUs. We use ILP with an objective of obtaining maximum throughput while minimizing slowdowns. Specifically, the proposed methodology focuses on the maximization of GPU's throughput by (i) performing application classification; (ii) analyzing the per-class interference and slow-down; (iii) finding the best matching between classes; and (iv) it employs an efficient kernel-to-SM policy that reduces the destructive effects of applications' interference.

CHAPTER 2

LITERATURE REVIEW

Improving performance and throughput of GPUs has been researched previously in the context of thread-level parallelism, concurrent kernel execution and shared resource contention.

2.1 THREAD-LEVEL PARALLELISM

GPUs have tremendous compute power coupled with very high bandwidth memory. However, the performance of an application and throughput of the device rely on how efficiently the computaional power of the device is being utilized by the workload. In GPUs branch divergence within warps leads to partial utilization of compute units with in SMs. This is because only one branch can be active at any time. For example, assume a warp of 32 threads with each thread taking a separate branch. In this case only one thread will be active. This means only one compute unit is being utilized until all branches merge. Authors in [3] proposed to use large warps to improve the performance of GPU applications. These warps are divided into sub-warps of 32 threads each with a modified register file architecture. Their modified register file is shown in Figure 2.1. In case of branch divergence within warps they select sub-warps within the same warp which allows more number of threads to be active. Thereby they increase the utilization of the device and performance of the application. Even though this technique improves the performance of the application, it cannot improve the device utilization if the application does not exploit thread-level parallelism. The problem of warp divergence is also addressed by authors in [10]. In their work the authors, use a metric called *Warp Progression Similarity* (WPS) to measure the divergence of warp execution progress. They propose a divergence aware warp scheduler, that schedules warps to minimize WPS and maximize GPU throughput. To obtain WPS they use offline profiling data of bench-

marks.

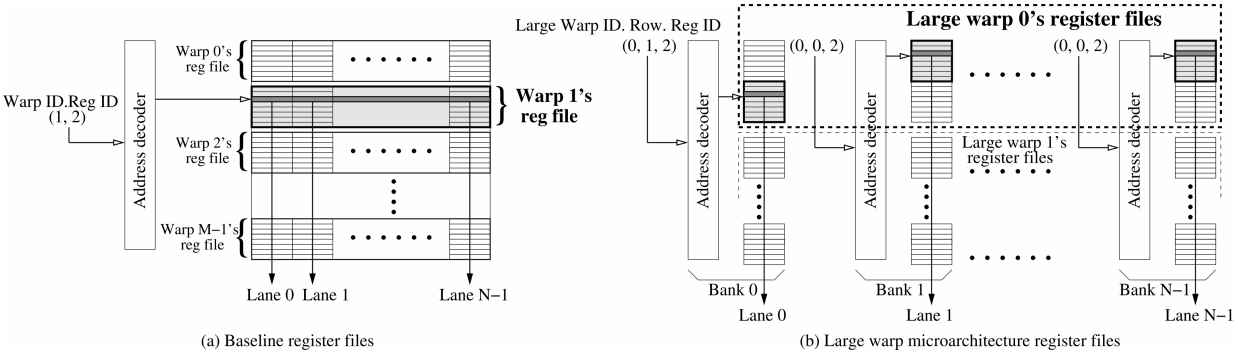


Figure 2.1: Large warp vs baseline register file design [3]

Throughput of GPUs can also get effected due to improper usage of available resources of GPU by programmer, that can lead to under utilization. In general a kernel is organized as grids and blocks. Programming APIs like CUDA, OpenCl allow programmers to organize the threads in their kernels. Programmers organization often may not fully utilize the resources available on the device. In [7] , the authors proposed a technique to address this issue using elastic kernels. They address the issue of less number of active threads per SM. They try to solve this issue by forming soft kernels by grouping threads from multiple blocks in a kernel. This allows applications to utilize all the resources available on each SM of the GPU. In their approach thread Ids change as the block configuration changed. As a result, The applicability of this technique is highly limited. Their technique cannot be applied to every kernel. Specifically, the presented scheme will not work for kernels that are effected when the hardware thread ids are different from software ids. In GPUs, the number of blocks a SM can serve at a time is limited due to capacity and scheduling limits. Authors in [11] suggest that number of blocks are limited mostly due to scheduling limits rather than resource constraints. So, they propose to use virtual threads. They schedule more number of blocks on a SM than it can host. These threads are marked as active and inactive. At any time there cannot be more active threads than the SM can serve. This concept helps in case of memory latency, as more number of threads are available for faster context switch.

The idea of improving Thread Level Parallelism can improve the throughput of the device. However, this gain is limited to parallelism available in the application. This limitation gave a new idea of allowing multiple kernels to co-exist on a single GPU.

2.2 CONCURRENT KERNEL EXECUTION

Current generations of GPUs support concurrent execution of kernels provided they are launched by the same CPU thread. Vendors like NVIDIA extended their support to allow multiple kernels to co-exist on GPUs at any time. To achieve this, the concept of streams was introduced. Execution within a stream is serial while multiple streams execute in parallel. To execute multiple kernels in parallel, they are launched into different streams from a single CPU thread. (Note: *Kernel launches are always asynchronous*). Soon a problem was identified in the hardware queue that lead to false serialization of kernels. NVIDIA then introduced hyperQ [12] mechanism to solve this issue. However, device utilization is still limited by the data dependencies among the kernels of the same application.

The key to increase the throughput further is to allow multiple kernels from different CPU threads to co-exist. Traditional way for multiple application execution on GPUs is time sharing. Nonetheless, this adds a high overhead of context switching. of selected classes to achieve higher device utilization. Authors in [13] proposed a mechanism for simultaneous kernel execution. In this work, a portion of threads from an already running kernel are preempted and the freed resources are given to a new incoming kernel. This mechanism still involves partial context switching, which is a big overhead. The authors in [14] proposed a technique to run kernels concurrently from different contexts through context funneling. NVIDIA introduced CUDA MPS [15] to support running kernels from multiple contexts. However, the kernels are not actually executed concurrently by the device. In MPS there is a MPS server and CPU threads that launch work are called clients. The MPS server serves a context only after the previous one has finished. However, with

CUDA-4.0 [16] NVIDIA allowed to launch multiple kernels from different CPU threads with the support of their hyperQ mechanism [12]. The authors in [17] proposed to execute multiple applications concurrently on a GPU through resource partitioning. Last, in [8, 18] GPU resource partitioning policies for multiple application execution on GPUs are presented. As an extension to their work, they propose resource partitioning policies for the co-executing applications using offline profiling data. The authors in [8, 17, 18] here show that general purpose applications do not scale linearly with cores. However, they do not have any policy on which applications can co-exist on the device. Improper selection of applications to co-exist can greatly damage the throughput of the device.

2.3 SHARED RESOURCE CONTENTION

All the SMs in a GPU share the last level cache and memory controllers. GPUs are very effective in hiding latency. The application needs to have enough parallelism to hide the latency. However, all the applications may not have enough parallelism and their latency can be noticed. For applications like these, shared resource contention can further increase the latency. The authors in [19] present a warp-aware memory scheduling method that focuses on minimizing inter-warp contention thereby reducing latency. The authors in [20] proposed a method to assign to GPUs the required bandwidth and reduce contention with other devices. In [6, 9], a memory scheduling policy for GPUs that supports concurrent application execution is proposed. The scheme improves device throughput by reducing the contention in shared resources. Whereas, in the presented work *we co-schedule applications that have less contention* while improving the utilization of the device.

In the scope of this work, *we use automatic context funneling* provided by CUDA API along with *hyper Q mechanism* and *modified work distributor* to run multiple applications concurrently. Our stream management unit and hyperQ with work distributor are presented in Figure 2.2. Our approach, instead of selecting applications to co-exist in

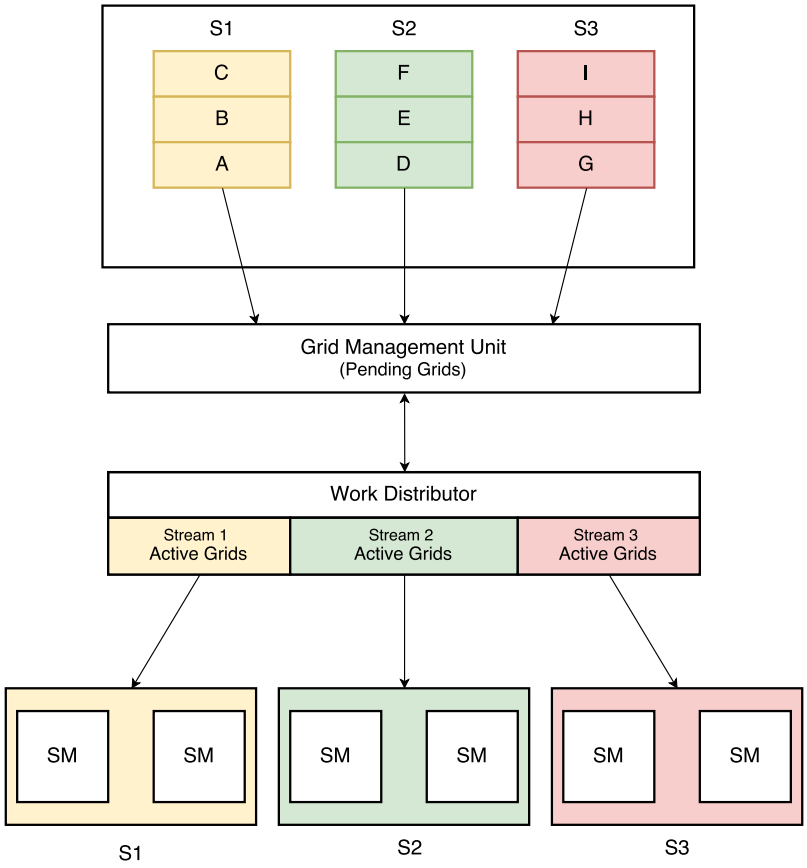


Figure 2.2: Stream Queue Management and Work Distributor

their order of arrival we propose a method presented in Section 3.2.3 in which we choose applications, using ILP, that yield maximum device throughput. We then use a dynamic resource allocation policy presented in Section 3.2.4 to further optimize throughput.

CHAPTER 3

DEFINITIONS - PROPOSED METHOD

In this Chapter a detailed discussion on the thesis is presented. For testing the proposed methodology rodinia [1] benchmarks and a modified version of GPGPU-Simulator [4] are used. Section 3.1 introduces GPGPU-Sim briefly. Section 3.2.1 describes our application classification criteria. Our proposed methodology is presented in Section 3.2.

3.1 SIMULATORS

This Section presents the simulators that were used for our work. We also present changes and features added to the simulators as part of our work.

3.1.1 GPGPU-Sim

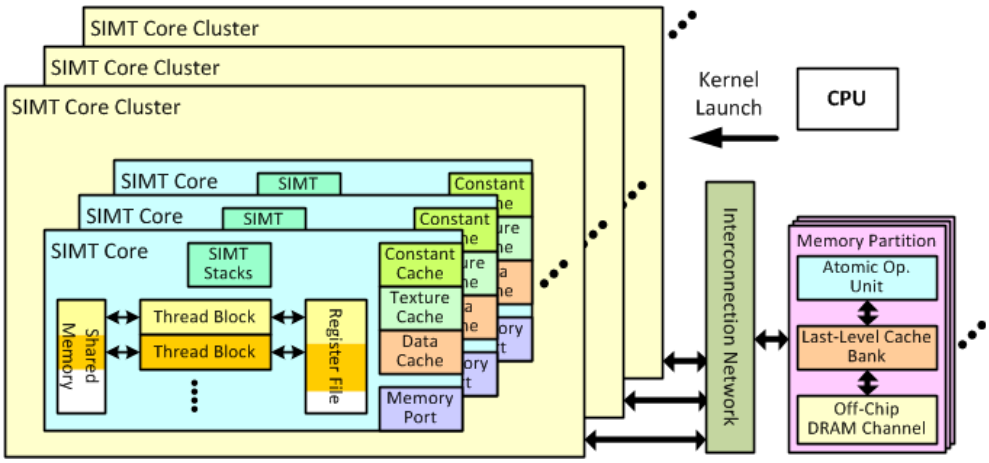


Figure 3.1: Overall GPU Architecture Modeled by GPGPU-Sim [4]

GPGPU-Sim [4] is a cycle accurate gpu simulator mainly focused on gpu-computing. GPGPU-Sim 3.x is the latest version og the simulator. This version supports Fermi and Tesla NVIDIA gpu microtectures. GPGPU-Sim supports OpenCL and CUDA. The simulator can run both CPU programs and GPU programs, but only the GPU timing is mea-

sured.

The overall GPU architecture model of the simulator is shown in the Figure 3.1. The micro architecture of GPGPU-Sim consists of SIMT cores which are connected to to GDDR DRAM with an on-chip network. Each SIMT core contains multiple processing elements, which are collectively called as Streaming Multiprocessor in NVIDIA terms. All the processing elements in a SM share L1 cache and a large register file. These SIMT cores consit two warp schedulers and two decode units each. Each Processing unit has it's own Load Store unit, which is one of the keys for high compute power of a GPU. In addition to general purpose processing elements each Streaming Multiprocessor also has multiple Special Function Units. All the processing elememts in a Streaming Multiprocessor are interconnected using a interconect network. A block diagram of a Streaming Multiprocessor is presented in Figure 3.2. The detailed microarchitecture of a SIMT core is shown in Figure 3.3.

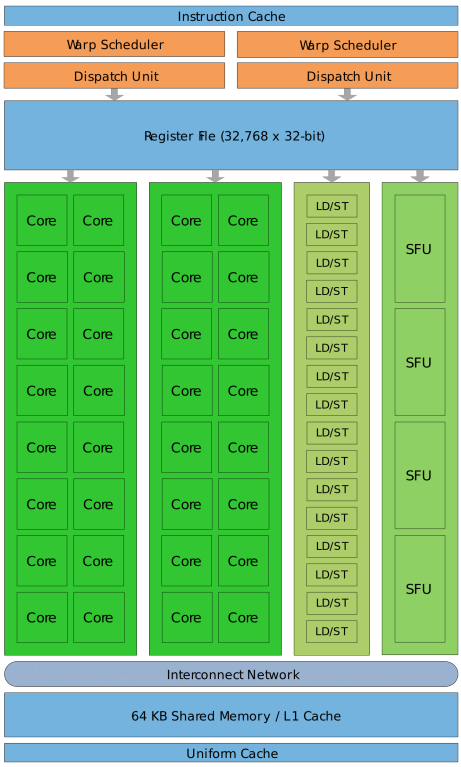


Figure 3.2: SIMT Core [5]

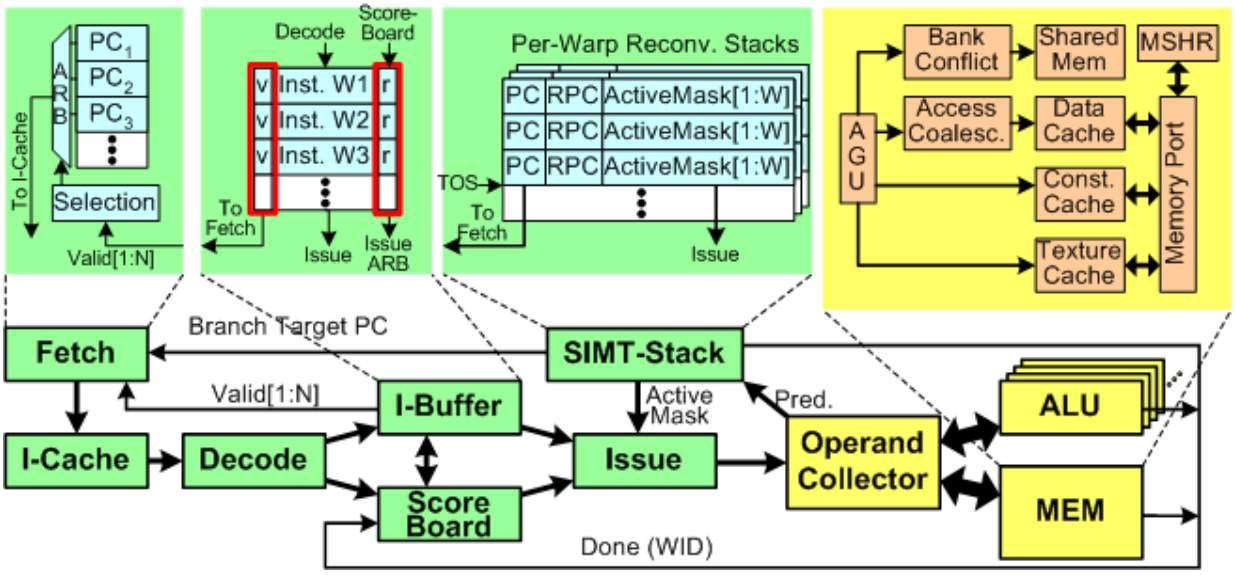


Figure 3.3: Detailed Microarchitecture Model of SIMT Core [4]

3.2 PROPOSED METHODOLOGY

Our work mainly focuses on optimizing device throughput when multiple applications offload kernels onto the device. The state of art method of task level parallelism in GPUs is to run multiple kernels concurrently in the order of arrival. The state of art method does not consider the contention one application can cause to other concurrently running application which will throttle the device throughput. All the SMs in a GPU share a common Last Level Cache and memory controllers. Multiple kernels with high interference when run concurrently, greatly reduce the throughput of the device. We propose a mechanism that selects which applications can run concurrently on the device.

Additionally in this thesis we worked on what portion of these resources is allotted to each concurrently running application. The traditional approach is to allocate equal partition the resources equal among the concurrently running applications. However, this approach is not feasible. We propose a dynamic resource allocation algorithm that allocates resources the applications based on the dynamic behaviour of each application.

The first step of our methodology is to profile each application when running alone and divide the applications into classes based on the profiling results. Section 3.2.1

presents our application classification details. The second step (Section 3.2.2) is to calculate interference of one class of application on other classes. Next step is to select application from the queue which will yield a high device throughput. This is presented in Section 3.2.3. Finally, in Section 3.2.4 we present our dynamic resource allocation algorithm which partitions resources among the running applications to furthur optimize the throughput.

3.2.1 Application Classification

For our methodology we first profile each application and divide the applications into four classes namely (i) Memory (class M) Intensive, (ii) Memory and Cache (class MC) Intensive, (iii) Cache (class C) Intensive and (iV) Compute (class A) Intensive. The first three classes focus on Shared resource contention, where as class A tell us how much of device resources the benchmark can use.

Table 3.1: Classification criteria

class	classification criteria
M	$MB > \alpha$
MC	$\beta < MB < \alpha$
C	$L2 \rightarrow L1 > \gamma$ $R > 0.2$ $IPC < \epsilon$
A	$R < 0.2$ $IPC > \epsilon$

If an application has Memory Bandwidth $> \alpha$ the application is classified as class M application. Applcations with memory bandwidth $> \beta$ AND $< \alpha$ are classified as class MC applications. Applications with memory bandwidth $< \beta$ AND L2→L1 bandwidth $> \gamma$ (OR) Memory to Compute Ratio > 0.2 (AND) Instructions per Cycle is $< 0.2 \times IPC_{max}$ fall into class C. If applications have IPC $> 0.2 \times IPC_{max}$ (AND) Memory to Compute Ratio < 0.2 then they go to calss A. The values of α, β, γ are chosen based on the GPU. For our work we used a GTX 480 architecture and the values of α, β and γ are

Table 3.2: classification of rodinia [1] benchmarks

Benchmark	$MemoryBandwidth$	$L2 \rightarrow L1$	IPC	R	class
BFS2	35.5	132.9	19.4	0.19	C
BLK	116.2	83.13	577.1	0.05	M
BP	84.06	142.7	808.3	0.06	MC
LUD	0.19	8.14	40.1	0.03	A
FFT	105.8	122.8	405.7	0.08	MC
JPEG	47.2	77.7	386.4	0.07	A
3DS	81.4	102.75	533.9	0.11	MC
HS	43.93	97.3	984.0	0.01	A
LPS	80.6	115.4	540.9	0.03	MC
RAY	59.7	69.1	523.9	0.1	MC
GUPS	108.75	97.1	10.61	0.1	M
SPMV	48.1	121.3	208.7	0.07	C
SAD	57.35	46.1	781.9	0.01	A
NN	1.3	35.3	56.8	0.15	A

$0.30 \times MB_{max} = 50$GBps, $0.55 \times MB_{max} = 107$GBps and 100GBps respectively. The value of ϵ is 200 instructions per cycle.

3.2.2 Interference Calculation

After classifying applications based on the profiling results, we run each application with every other application and calculate the slowdown of each application compared to their alone running time. We then based on the our classification presented in Table 3.2 calculate the average slowdown imposed by each class on every other class. The results are shown in the Figure 3.4.

The results show that class M applications impose slow-down on all the other classes. This is due to the fact that the memory controller is overloaded by the class M applications, plus the default memory scheduler (FR-FCFS scheduler [21, 22]) prioritizes row buffer hits which again favors the class M applications. Based on the results, the situation is same when class M applications are executed along with class MC applications. In this case class MC applications suffer more than class M applications. These results

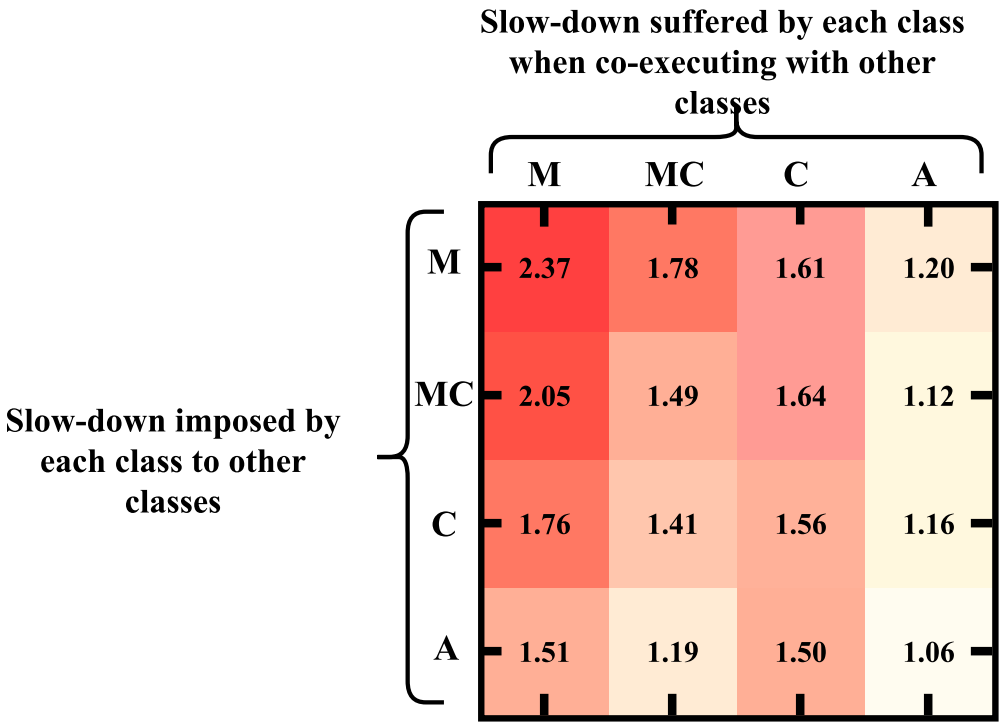


Figure 3.4: Average Application slowdown due to co-execution

are used in the next step to minimize contention under concurrent execution.

3.2.3 Contention Minimization

In this step, our goal is to select applications that can run concurrently on device that yield high throughput. Here we propose to use Integer Linear program, introduced in Section 1.4, to reduce contention and maximize throughput. ILP is generally used to maximize the outcome of a function subject to some constraints. In our work, our aim is to maximize throughput of the device. To achieve this, we focus on minimizing contention by using the slowdown values obtained from the previous section. We define s_i as slowdown of class i. We then take inverse of the slowdowns and add them for the whole queue of applications and try to maximize this value. This is given by Equation 3.3.

We present our methodology for running two applications which can be replicated for three application execution. For our work, we consider a GPU with N_{SM} number

of SMs. The SMs are divided into N_C groups, where N_C=2 indicates two application co-execution, N_C=3 indicates three applications co-execution. We have C collection of classes, where $C = \{c_1, c_2, \ldots, c_{N_T}\}$. N_T is number of classes. We assume length of queue to be N_q. The total number of groups of applications (L) formed with queue of length N_q is given by $L = \frac{N_q}{N_C}$. We define N_P as number of application pairs that can be formed with N_q length queue. P_K=$\{p_1, p_2, \ldots, p_{N_P}\}$. Where p_i one of many patterns that can be formed from the queue of applications. For example pattern p_i has two class MC applications then p_i is given by Equation 3.1.

$$p_i = \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \end{bmatrix} \tag{3.1}$$

$$N_P = \binom{N_T + N_C - 1}{NC} \tag{3.2}$$

Our aim is to maximize function f using ILP. Using ILP we obtain the values of $L_1, L_2, \ldots, L_n$ which give us the highest value for f. Where L_i represents how many times the pattern p_i to be used to obtain a maximum value for function f.

$$f = e_1 L_1 + e_2 L_2 + \cdots + e_{N_P} L_{N_P} \tag{3.3}$$

Where e_i is the inverse of slowdown of applications in L_i and is given by Equation 3.4. S_i is the slowdown of application.

$$e_k = \frac{1}{N_C}\Big(\frac{1}{S_1^k} + \frac{1}{S_2^k} + \cdots + \frac{1}{S_{N_C}^k}\Big) \tag{3.4}$$

N_q^i is the number of applications of i_{th} class in the queue. So, it can be said that the total number of applications in the queue is equal to sum of number of applications of each class of applications present in the queue.

$$N_q = N_q^1 + N_q^2 + \cdots + N_q^{N_T} \tag{3.5}$$

As previously mentioned p_i has the information of what classes of applications are present in it. Multiplying $p_i[1]$ with L_1 gives number of class 1(class M for example) applications present in the queue. This can be given by Equation 3.6.

$$\begin{bmatrix} P_1 & P_2 & \cdots & P_{N_P} \end{bmatrix} \begin{bmatrix} L_1 \\ L_2 \\ \vdots \\ L_{N_P} \end{bmatrix} = \begin{bmatrix} N_q^1 \\ N_q^2 \\ \vdots \\ N_q^{N_T} \end{bmatrix} \tag{3.6}$$

Solving Equation 3.6 will gives us constraints to be used to obtained the values of L_i $\forall$ i=1,2,3,...N_p.

We previously stated that L is the total number of groups that can be formed with a queue of length N_q. We also states L_i as number of times each pattern p_i appears in the result set. From these observations it is clear that sum of L_i $\forall$ i=1,2,3,...N_p should be equal to L. This is forms another constraint for calculating the final result set.

$$L_1 + L_2 + \cdots + L_{N_P} = L \tag{3.7}$$

So, we propose that maximizing function f subject to constraints 3.6 and 3.7 will give a high device throughput.

3.2.4 SM Allocation

In the previous section we find out which applications can run together. This will guarantee a higher throughput for the queue of applications. We achieve this gain by reducing interference among concurrently running applications. To further improve the throughput of the device we now look into available parallelism of each application. As mentioned in the previous section we divide the available SMs into N_C sets and each application gets one set of cores. In our initial work we divided the SMs in such a way that each set in N_C has same number of cores. We then tested each application with different number of cores. The results are presented in the chart 3.6. Soon, we observed that some applications cannot use all the resources available to it. Most notable scalability trends are shown in the chart 3.5.

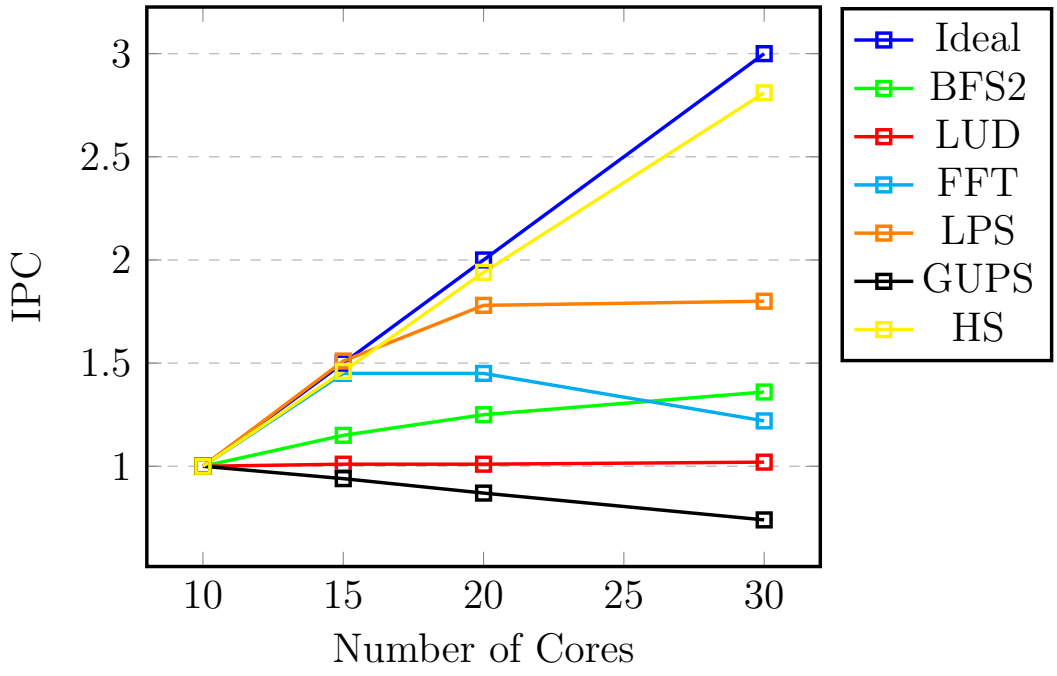


Figure 3.5: Scalability Trends of Benchmarks

The most noticeable thing is the behaviour of the benchmark GUPS. The IPC of GUPS decreases with increase in number of cores. This is because GUPS is a memory intensive application (from Table 3.2) and as number of active threads increase, number of memory requests increases. This increases contention in memory interconnect

which further throttles the throughput. Benchmark LUD has a constant IPC no matter what number of cores were given to it. Benchmarks like HS and SAD have enough parallelism and scale more close to the ideal performance curve. Some applications (like LPS) have moderate parallelism and saturate after certain number of cores. Some applications like FFT saturate and looses performance on further increase of cores. Applications like BFS2 and NN scale linearly, from Figure 3.5, with cores but have low device utilization. Maximum Device utilization of each benchmark is presented in Figure 1.2.

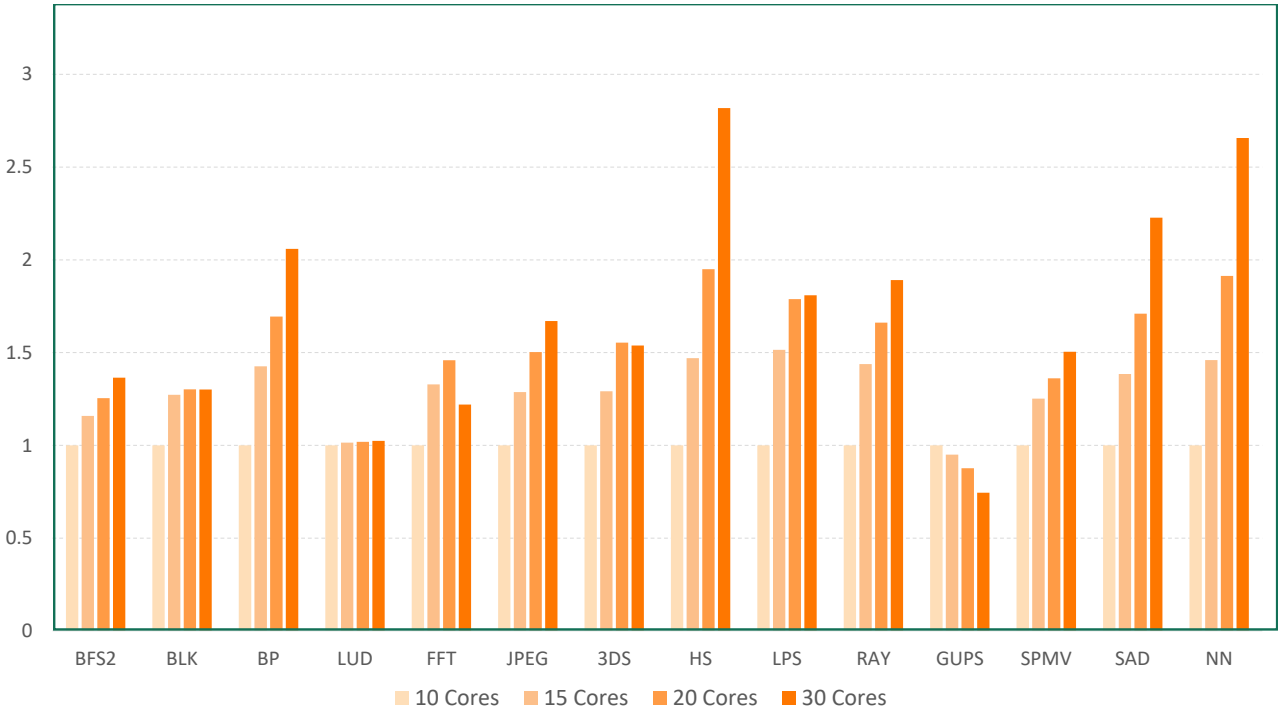


Figure 3.6: IPC of Benchmarks with different number of cores

From these observations we propose a dynamic SM allocation algorithm that allocates SMs to the co-executing applications based on their dynamic behaviour. Our algorithm is presented in Algorithm 1. Our algorithm needs three statistics as input, (i) Throughput of the device, (ii) Throughput of each co-existing application and (iii) Bandwidth utilization of each concurrently running application. Based on the inputs (ii) and (iii) the algorithm gives a score to each application. Input (i) is used to judge the effect of the new resource allocation, if the throughput of the device decreases than

the throughput before re-allocation then previous resource allocation configuration is restored.

We initially start with equal SM distribution. After T_C cycles we get the required statistics and a new decision is made on which application does not utilize the given resources. Then SMs from that application are transferred to other co-existing application. If all the co-existing applications have similar behaviour, then we stick with the present SM partitioning. In the beginning all applications have a score value of 0. The algorithm after every T_C cycles checks the device throughput and performance statistics for each executing application and updates the score of each application.

Based on the values, each application changes its score values. If the Instructions Per Cycle (IPC) of an application App_i is less that a value IPC_{thr}, the score of this application is $V[i] = 1$. If the bandwidth utilization is grater than a value BW_{thr} then $V[i] = 2$ and if both conditions are true then $V[i] = 3$ A high score means that the application negatively affects the throughput of the device. This happens because an application with low IPC and high memory bandwidth relies on data transfer the SMs that is has allocated can be used by another compute intensive application increasing the throughput of the GPU in total. Then, based on the score of each application, we deallocate n_r SMs from application with the highest score and we allocate them to the application with the lowest score. When the allocated resources for an application reaches R_{min} the score of that application is set to a negative value and will it is increased again.

The SM deallocation can be done in three ways. The first method requires partial context switching [13] which is expensive in terms of latency and interconnect bandwidth. The second way, is to completely discard the running kernel on the selected SMs. However, this approach imposes big performance slow-down. The last way is to let the selected SMs finish the currently running blocks and once they are finished, they are transfered to the other application. Algorithm 1 follows the third method which, even though it has a small performance overhead, it allows for smooth exchange of SMs at

run-time.

Algorithm 1 SM Allocation

T:Current Throughput
Tp:Previous Throughput
N:Total number of SMs
n:Number of applications running concurrently
R_i:Number of SMs for i_{th} application
$S[i]$:Score of i_{th} application
R_{min}:Minimum SMs Required for an application
$BWutil$:Memory bandwidth utilization

```
1: Initial:
2: for each app do
3:     V[i]=0;
4:     R[i]=N/n;
5: end for
6:
7: for every T_C cycles do
8:     while T > T_p and R_i(N) > R_min do
9:         for each app do
10:            if IPC[i] < IPC_thr then
11:                V[i]++;
12:            end if
13:            if BWutil[i] > BW_thr then
14:                V[i]++;
15:            end if
16:        end for
17:        for each app do
18:            if V[i] == V[i+1] then
19:                break;
20:            end if
21:            if V[i] == max(V) then
22:                R_i=R_i − n_r;
23:            elseIf V[i] == min(V)
24:                R_i=R_i + n_r;
25:            end if
26:        end for
27:    end while
28:    V[i]=0
29: end for
```

CHAPTER 4

EXPERIMENTAL RESULTS

In order to validate our methodology we have performed extensive simulation experiments using GPGPU-Sim [4], a cycle-level accurate simulator for GPUs, that supports NVIDIA CUDA, and Rodinia [1] benchmarks as high performance parallel applications. GPGPU-Sim was modified in order to support multiple streams and concurrent application execution. The experimental set up is described in Table 4.1.

Table 4.1: Experimental set up

GPU Architecture - GTX 480	
# of SMs	60
Core frequency	700MHz
Warps per SM	48
Blocks per SM	8
Shared Memory	48kB
L1 Data cache	16kB per SM
L1 Instr. cache	2kB per SM
L2 cache	768kB
Warp scheduler	GTO [23]

4.1 TWO APPLICATION EXECUTION

We first designed a queue with 14 applications with 2 class M and class C applications, 5 applications each of class MC and class A. We, then executed applications from the queue in Even approach and set this as baseline for our testing. The we run two applications together in FCFS way and then using out ILP method. The results are shown in Figure 4.1. We observed that the proposed method showed 21% better throughput than FCFS and over 80% improvement than Even approach in case of two application execution.

Also, from Figure 4.2 we can see that 5 of 7 pairs formed using ILP finished in less than 50% of time than their serial execution time while only 2 pairs formed using FCFS

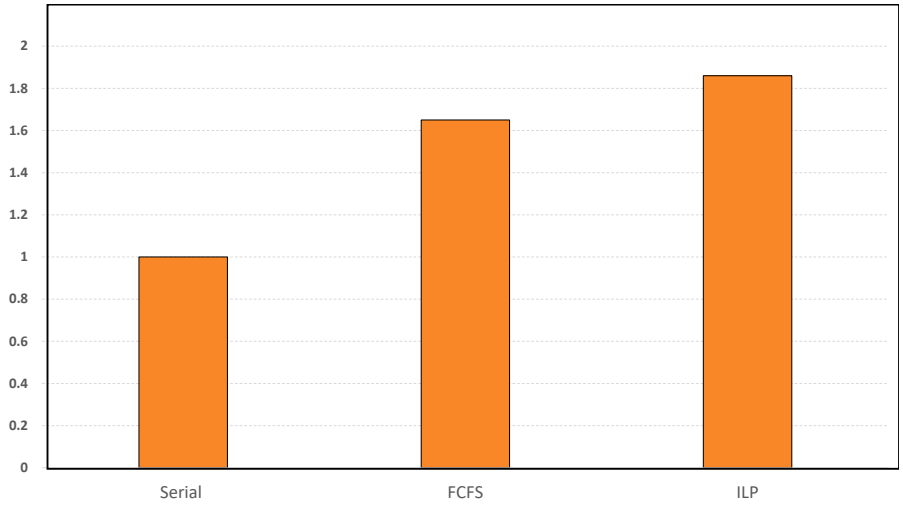
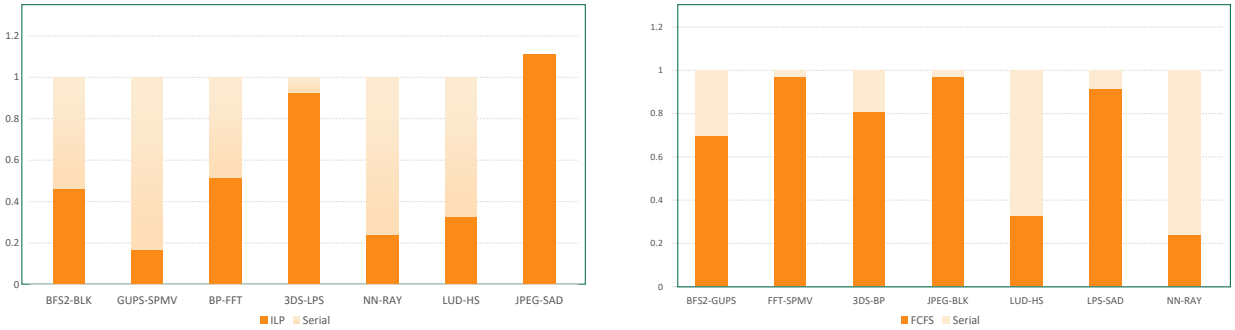


Figure 4.1: Throughput Comparison of two application execution When application pairs are formed using ILP and FCFS compared to their Even approach time



(a) ILP

(b) FCFS

Figure 4.2: Cycles taken by each pair of applications When application pairs are formed using (a) ILP (b) FCFS compared to their serial Execution time

finished in 50% of their serial execution time.

We then designed queues of 20 applications with varying distributions of different of classes of applications to verify the scalability of our approach. The distributions are (i) Equal distribution of each class (ii) 55% class M and 15% each of other classes (iii) 55% class MC and 15% each of other classes (iv) 55% class C and 15% each of other classes and (v) 55% class A and 15% each of other classes.

We name our methodologies presented in Section 3.2.3 and Section 3.2.4 as *ILP* and *ILP+SMRA* respectively. We compare our methodologies with (i)An *Even* approach that

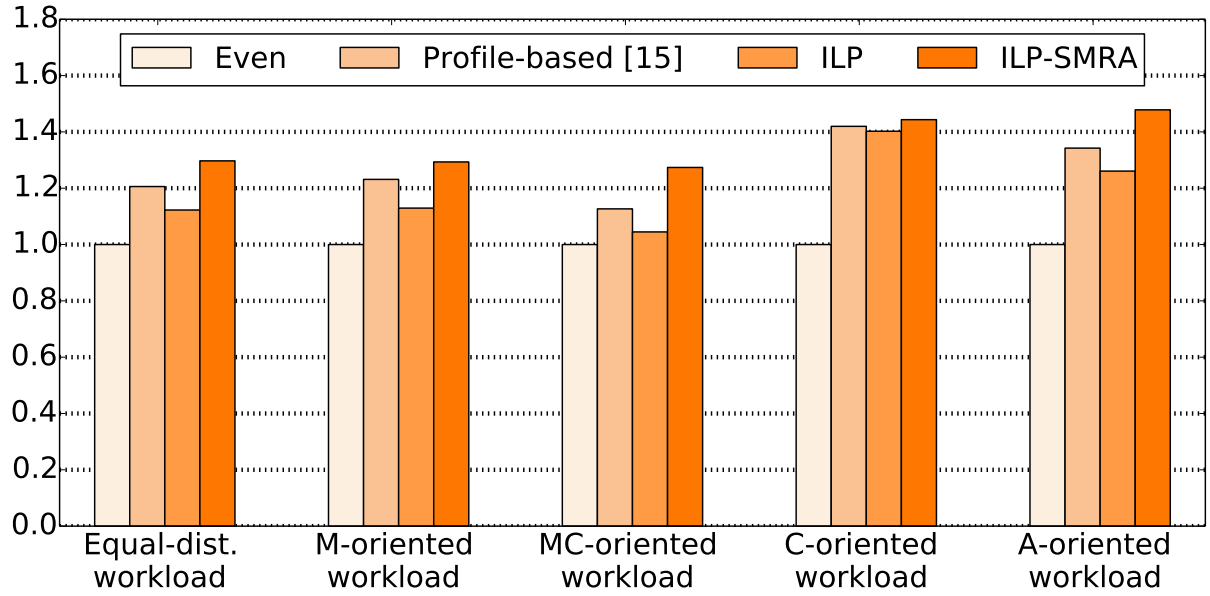


Figure 4.3: Concurrent execution of two applications

assigns equal SMs to each application and selects applications in the order of arrival. We consider this as baseline for our comparisons. (ii) *Profiling-based* Method proposed in [17], which assigns resources to applications based on the offline profiled data of each application and selects applications in the order of arrival. However, the *Profiling-based* method needs extensive profiling of every single application to obtain gain in throughput. This method does not consider the dynamic behavior of the applications.

Results of our simulations for different queue distributions are presented in Figure 4.3. Figure 4.4 presents the device throughput for different distributions of queue. ILP Method increased throughput by an average of 19% achieving the best gain of 40% when the queue has 55% Cache applications. As aforementioned, the *ILP* method focuses on finding the best application matching in order to reduce contention working at the granularity of classes. *ILP-SMRA* increases throughput by an average of 36%, compared to the *Even* method, achieving the best gain of 48% in the A-oriented workload. *ILP-SMRA* not only reduces contention due to the best matching of the classes, but it performs run-time SM reallocation that further boost the performance of the GPU.

4.1.1 Queue with equal class distribution

Here the queue contains equal number of all classes of applications. The profiling method is based on extensive off-line profiling and can guarantee maximum possible throughput for an application. So, for our comparisons the Even approach method sets a baseline while the profiling method sets the maximum throughputs that can be achieved. As mentioned earlier the profile-based method does not consider the dynamic behavior of applications. So, it is possible that this method cannot guarantee maximum throughput. This situation is observed with almost every queue distribution.

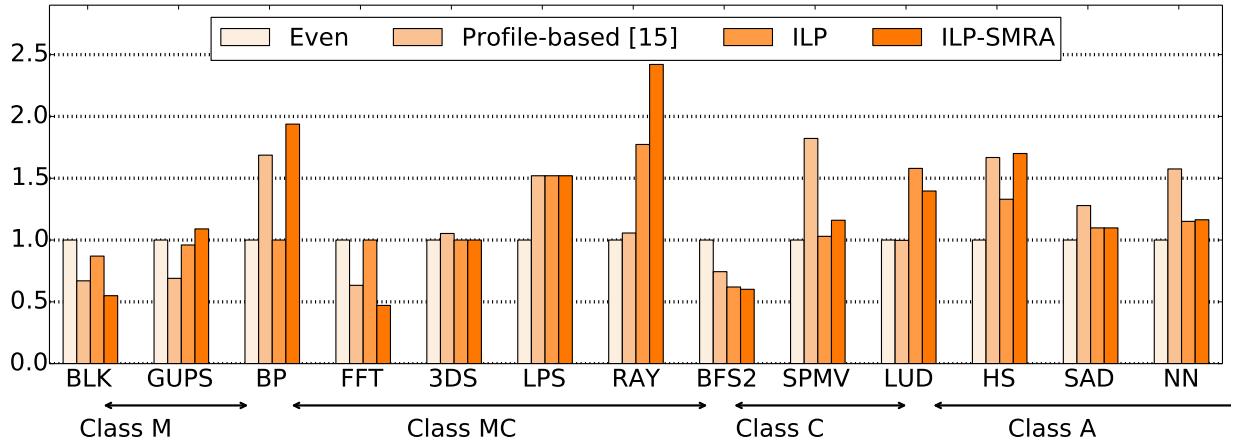


Figure 4.4: Concurrent execution of two applications with equal distribution

Figure 4.4 shows the performance results when queue has equal number of applications from every class. We can see that some applications suffer when they are co-executing with some other application. However, our methods ensure that the loss of one application is overshadowed by the gain of the application running along with it. ILP performed better than the Even approach(Even approach) by 9% on average having a maximum gain of 70% for RAY. When compared with the Profiling method [17] ILP performed on average 8% better with a maximum gain of 65% is observed for RAY than the Profiling-based method. The ILP+SMRA method obtained an average gain of 17% compared to the even approach for the applications. When compared with the profiling method the ILP+SMRA gained an average of 23.3% better throughput. ILP+SMRA ap-

proach obtained almost 1.5 times the throughput when compared to both the even and the Profiling-based approaches.

4.1.2 Queue with high class A distribution

Here the queue is dominated by computational intensive applications. We observed that in this situation our methods performed better with the computational applications while applications from other classes has suffered. we also observed that the Profiling-based method has also given a similar result. The Even approach performed better than ILP and Profiling-based method by 3% and 4% respectively. Our ILP+SMRA approach however has shown approximately 2% and 5% better throughput than Even approach and Profiling-based approaches respectively.

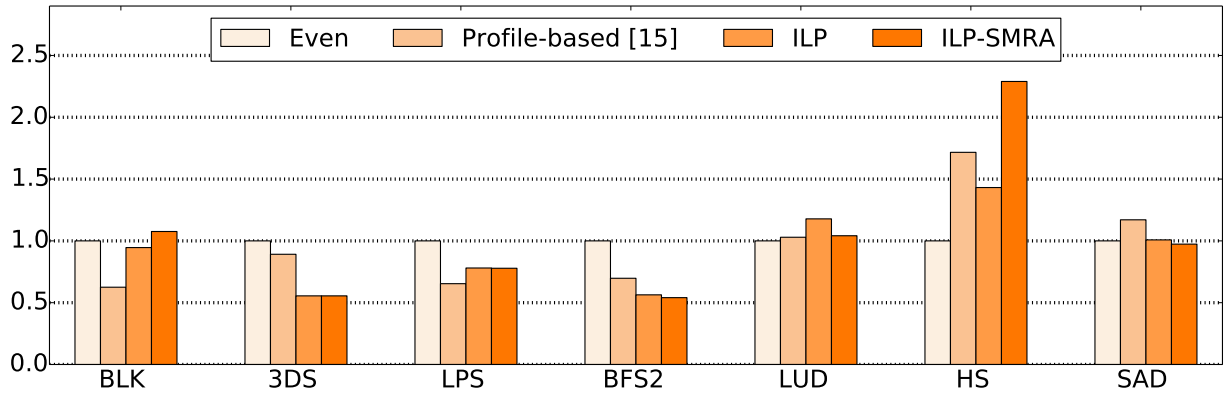


Figure 4.5: throughput with computational dense work queue

4.1.3 Queue with high class M distribution

In this scenario we have queue dominated by class M applications. We observed that our ILP method obtained 32.5% and 9% better throughput than the Even approach and the Profiling-based approach respectively. The ILP+SMRA approach has obtained an average 32% and 7% better throughput than the Even approach and the Profiling-based approach respectively.

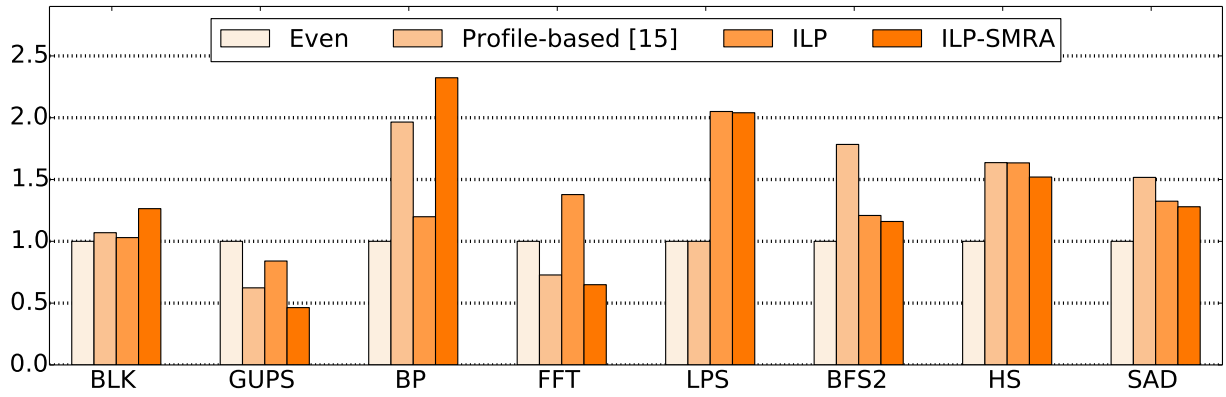


Figure 4.6: throughput with memory class dense work queue

4.1.4 Queue with high class MC distribution

In this scenario the queue is dominated by class MC applications. In this case ILP method performed almost similar to Even approach while the profiling method performed on average 5% better throughput than ILP. Our ILP+SMRA method has performed on average 3% better than Even approach and it performed almost similar to the Profiling-based method.

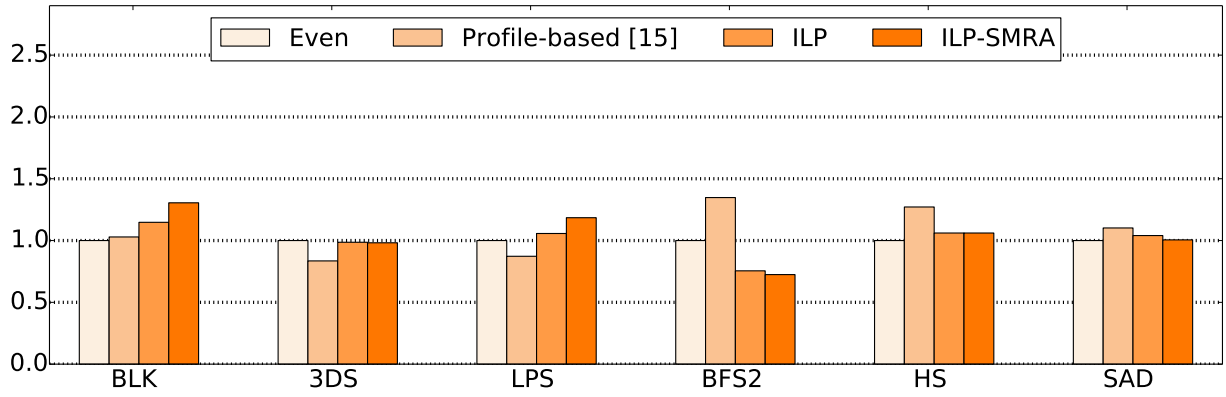


Figure 4.7: throughput with class MC dense work queue

4.1.5 Queue with high class C distribution

With the queue being dominated by class C applications, the ILP approach showed approximately same average throughput as the even approach while the Profiling-based

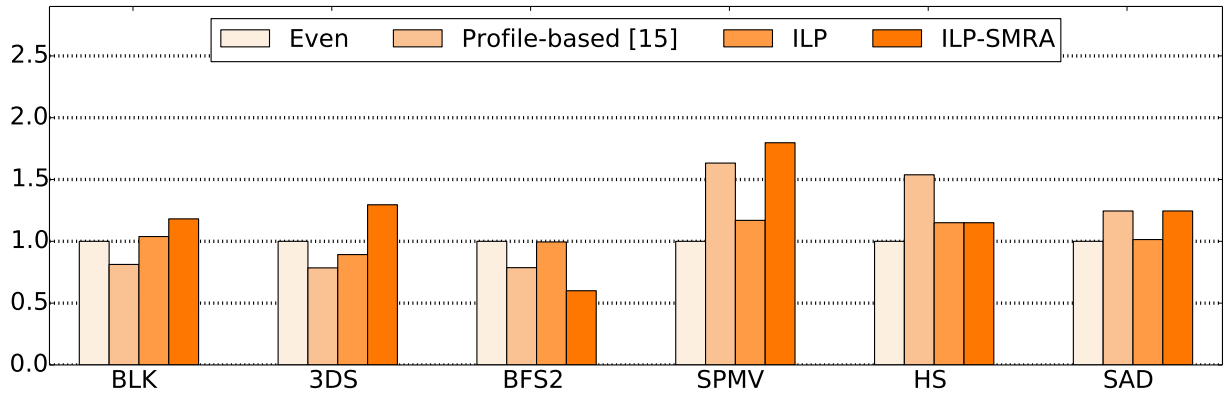


Figure 4.8: throughput with class c dense work queue

approach performed 9% better. Our ILP+SMRA on the other hand has performed 29% better than Even approach scenario on average. It also achieved an average 6% better throughput than the profiling-based method.

4.2 THREE APPLICATION EXECUTION

We then executed three applications concurrently and the throughput results are shown in Figure 4.9. With three application execution our ILP method achieves double the throughput than Even approach and 45% more than FCFS.

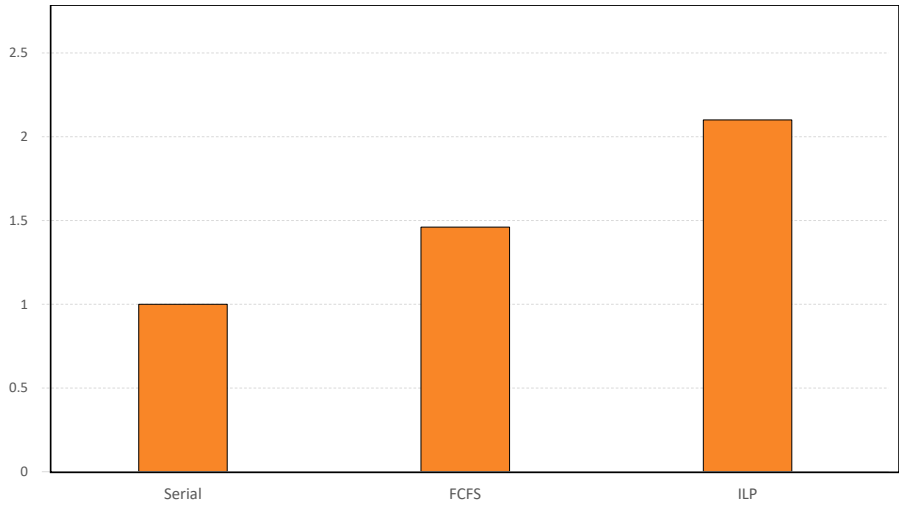
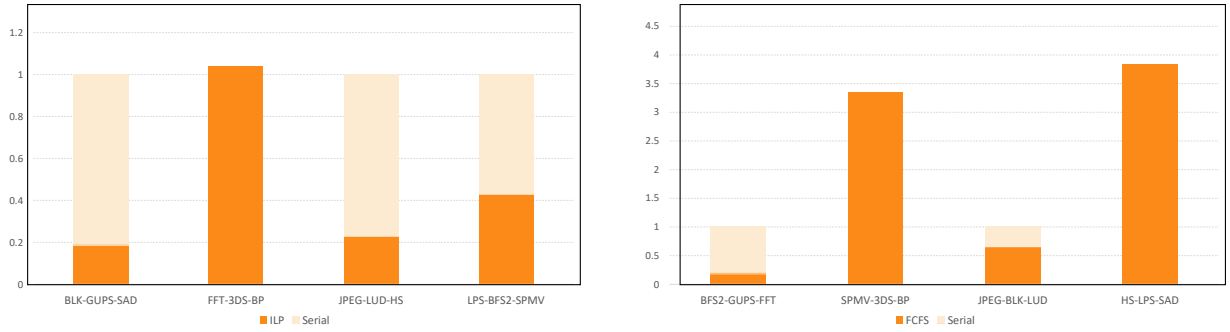


Figure 4.9: Throughput Comparison of three application execution When applications are selected using ILP and FCFS compared to their Even approach time



(a) ILP (b) FCFS

Figure 4.10: Cycles taken by each group of three applications When applications are selected using (a) ILP (b) FCFS compared to their Even approach time

The Figure 4.10(a) shows that 3 of 4 formed groups finish in less than 40% of their Even approach time while the Figure 4.10(b) shows that only 1 group of applications finished within 40% of their Even approach time.

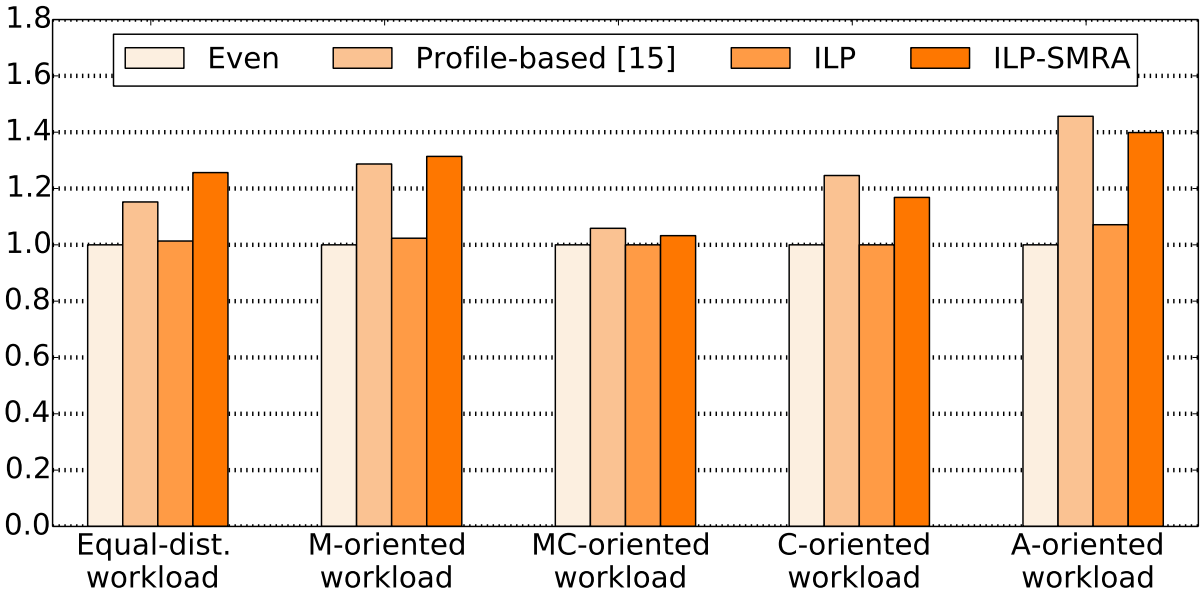


Figure 4.11: Concurrent execution of three applications

Simulation results for three concurrent applications with different queue distributions are presented in Figure 4.11. Figure 4.12 present the average device throughput for different distributions of queue. The *Even* approach is considered as the baseline for our comparison. *ILP-SMRA* increases throughput by an average of 23%, compared to the

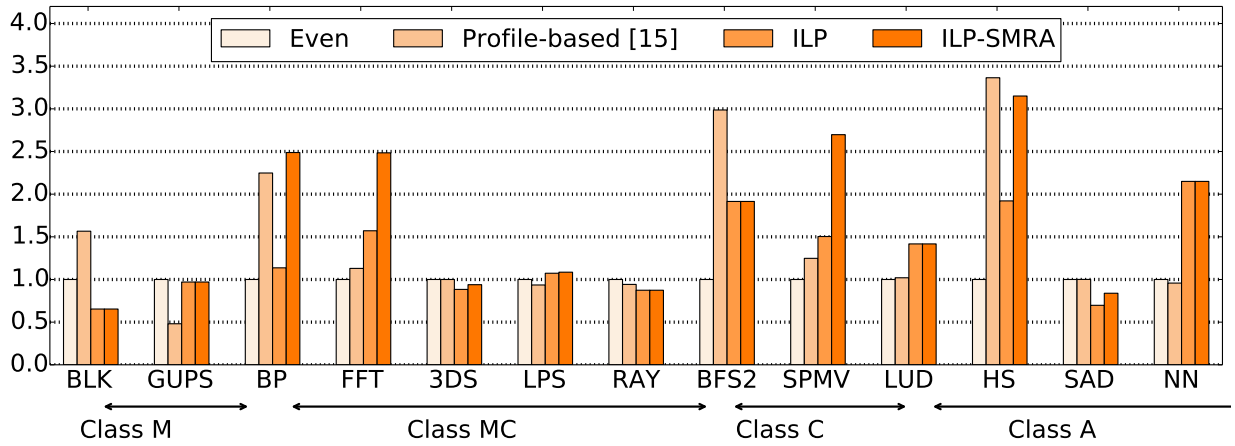


Figure 4.12: Average device throughput of different distributions of queue under Concurrent execution of three applications

Even method, achieving the best gain of 40% in the A-oriented workload. Even in the case of three simultaneous executing applications, *ILP-SMRA* reduces contention due to the best matching of the classes and reallocates SMs at run-time based on the score of each application further increasing the throughput of the GPU. Regarding the *Profile-based* method [17], it achieves on average 23% better throughput compared to the *Even* and performs similarly with the *ILP-SMRA*. However, as aforementioned, it requires extensive off-line profiling in order to find the best configuration which does not make it scalable and adaptive to new incoming applications.Figure 4.12 depicts the comparison for 3 concurrent applications. And in this case, *ILP* achieved on average a gain of 28% while *ILP-SMRA* increased the average IPC gain by 67% on average.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this thesis, a methodology for efficient concurrent execution of multiple applications on GPUs by minimizing the interference in shared resources was presented. Specifically, the proposed methodology focuses on the maximization of GPU's throughput by (i) performing application classification; (ii) analyzing the per-class interference and slowdown; (iii) finding the best matching between classes; and (iv) it employs an efficient kernel-to-SM policy that reduces the destructive effects of applications' interference. Experimental results showed that the proposed approach increases the throughput of the system for two concurrent applications by an average of 36% compared to other optimization techniques [17], while for three concurrent applications the proposed approach achieved an average gain of 23%.

5.1 FUTURE WORK

5.1.1 Dynamic Warps

In general consecutive threads are grouped to form warps. In case of branching with in a warp, threads that take one branch are allowed to execute ant the rest are halted. This leads to under utilization of resources available on a SM. Instead the trend of branching in different warps cab be monitored and threads can be regrouped to form new warps in which all threads take same branch.

5.1.2 Heterogeneous Systems

Mobile computing devices have similar architecture. Chip manufacturers like INTEL and AMD have already integrated GPU and CPU on a single chip, where CPU and GPU share same Last Level Cache. Traditional GPUs have their own memory controllers and Last Level Cache. Work we presented in this thesis is on traditional GPUs, where GPU

is placed on PCI extension as a co-processor. Our work can be extended to support this architecture.

APPENDICES

APPENDIX A

Example for Methodology Presented in Section 3.2.3:

For two Application execution we divide SMs into 2 groups ($N_C = 2$). As mentioned in Section 3.2.1 we have four classes of applications.(i.e., $N_T = 4$). Using the Formula 3.2 we get $N_p = 10$. We assume our queue length N_q to be 14. So, $L = 7$. So, total number of patterns possible are 10 $(p_1, p_2, \cdots, p_{10})$.

We then calculate $e_1, e_2, \cdots, e_{10}$ using slowdown values presented in Figure 3.4. By substituting all the values in Equation 3.3 we get the below equation.

$$f = max\{0.0072L_1 + 0.0110L_2 + 0.0146L_3 + 0.03584L_4 + 0.0204L_5 + 0.0202L_6 + 0.0698L_7 + 0.0178L_8 + 0.0412L_9 + 0.166L_{10}\} \quad (5.1)$$

The possible patterns are as shown below

M - M	M - MC	M - C
M - A	MC - MC	MC - C
MC - A	C - C	C - A
	A - A	

$$\begin{bmatrix} P_1 & P_2 & \cdots & P_{N_P} \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 2 \end{bmatrix} \quad (5.2)$$

We have 2 class M (N_q^1=2) , 5 class MC (N_q^2=5), 2 class C (N_q^3=2) and 5 class A (N_q^4=5) applications in our queue.

$$\begin{bmatrix} N_q^1 \\ N_q^2 \\ N_q^3 \\ N_q^{N_T} \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ 2 \\ 5 \end{bmatrix} \tag{5.3}$$

From Equation 3.6 we get

$$\begin{bmatrix} 2 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} L_1 \\ L_2 \\ \vdots \\ L_{10} \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ 2 \\ 5 \end{bmatrix} \tag{5.4}$$

Solving **??** we get the following inequalities.

$$\begin{aligned} 2L_1 + L_2 + L_3 + L_4 &\leq 2 \\ L_2 + 2L_5 + L_6 + L_7 &\leq 5 \\ L_3 + L_6 + 2L_8 + L_9 &\leq 2 \\ L_4 + L_7 + L_9 + 2L_{10} &\leq 5 \end{aligned} \tag{5.5}$$

From Equation 3.7 we get

$$L_1 + L_2 + \cdots + L_{10} = L = 7 \tag{5.6}$$

Solving Equation 5.1 using ILP subject to constraints in Equations 5.4 and 5.6 gives

$$\begin{bmatrix} L_1 \\ L_2 \\ L_3 \\ L_4 \\ L_5 \\ L_6 \\ L_7 \\ L_8 \\ L_9 \\ L_{10} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 2 \\ 0 \\ 2 \\ 0 \\ 1 \\ 0 \\ 0 \\ 2 \end{bmatrix} \tag{5.7}$$

So, the final solution set contains 2 pairs of p_3, 2 pairs of p_5 2 pairs of p_{10} and 1 pair of p_7 patterns.

REFERENCES

[1] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on.* Ieee, 2009, pp. 44–54.

[2] NVIDIA, "GPU accelarated computing, http://www.nvidia.com/object/what-is-gpu-computing.html."

[3] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 2011.

[4] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on.* IEEE, 2009, pp. 163–174.

[5] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi gf100 gpu architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50–59, 2011.

[6] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of gpu memory system for multi-application execution," in *Proceedings of the 2015 International Symposium on Memory Systems.* ACM, 2015.

[7] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving gpgpu concurrency with elastic kernels," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 407–418.

[8] P. Aguilera, K. Morrow, and N. S. Kim, "Qos-aware dynamic resource allocation for spatial-multitasking gpus," in *Design Automation Conference (ASP-DAC), 2014*

19th Asia and South Pacific. IEEE, 2014.

[9] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Application-aware memory system for fair and efficient execution of concurrent gpgpu applications," in *Proceedings of workshop on general purpose processing using GPUs*. ACM, 2014, p. 1.

[10] C. Zhang, H. Tabkhi, and G. Schirner, "Studying inter-warp divergence aware execution on gpus," *IEEE Computer Architecture Letters*, vol. 15, no. 2, pp. 117–120, 2016.

[11] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram, "Virtual thread: Maximizing thread-level parallelism beyond gpu scheduling limit," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 609–621.

[12] T. Bradley, "Hyper-q example," *NVidia Corporation. Whitepaper v1. 0*, 2012.

[13] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel: Fine-grained sharing of gpus," *IEEE Computer Architecture Letters*, vol. 15, no. 2, pp. 113–116, 2016.

[14] L. Wang, M. Huang, and T. El-Ghazawi, "Exploiting concurrent kernel execution on graphic processing units," in *High performance computing and simulation (HPCS), 2011 international conference on*. IEEE, 2011.

[15] F. Wende, T. Steinke, and F. Cordes, "Multi-threaded kernel offloading to gpgpu using hyper-q on kepler architecture," *ZIB-Rep. 14-19 June 2014*, 2014.

[16] C. Nvidia, "programming guide 4.0 (2012)," *URL: http://developer. download. nvidia. com/compute/DevZone/docs/html/C/doc/CUDA_C_ Programming_Guide. pdf*.

[17] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for gpgpu spatial multitasking," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 2012.

[18] P. Aguilera, K. Morrow, and N. S. Kim, "Fair share: Allocation of gpu resources for both performance and fairness," in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*. IEEE, 2014, pp. 440–447.

[19] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian, "Managing dram latency divergence in irregular gpgpu applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 128–139.

[20] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 850–855.

[21] S. Rixner, "Memory controller optimizations for web servers," in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*. IEEE, 2004, pp. 355–366.

[22] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2. ACM, 2000, pp. 128–138.

[23] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 72–83.

VITA

Graduate School
Southern Illinois University

Srinivasa Reddy Punyala

srinivasareddy.punyala@siu.edu

Jawaharlal Nehru Technological University Hyderabad
Bachelor of Technology, JNTUH, 2015

Thesis Title:

Throughput Optimization and Resource Allocation On GPUs Under Multi-Aapplication Execution

Major Professor: Dr. I. Anagnostopoulos