UNIVERSIDAD DE CANTABRIA

*Facultad*

*de*

*Ciencias*

# Hardware Support for Multiprogramming in GPUs

(Soporte Hardware para Multiprogramación en GPUs)

Trabajo de Fin de Grado

para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Mehdi Nasef

Directores: José Luis Bosque, Borja Pérez

Septiembre-2023

# Contents

# Abstract

With every generation, GPUs incorporate an ever-increasing amount of computational resources, which are dedicated to accelerate data-parallel applications. Using those resources efficiently requires applications to expose high levels of parallelism. Even then, they may not be able to maximize the usage of each resource type (bandwidth, compute resources, memory, the different execution units of the same core, etc).

Not all applications can fully use a complete GPU, but when executing more than one application concurrently, they may compensate for each other's lacks. this is the idea behind multiprogramming. It is supported by current GPUs, but the techniques they use only tackle the lack of thread level parallelism (TLP) issue. Other methods like Warped-Slicer have been proposed to reduce resource underutilization by making applications share the resources of the same SM (Streaming Multiprocessor, which is a core of the GPU in Nvidia terminology). This is called Intra-SM-sharing. Warped-Slicer, which is the state of the art for thread block scheduling, tries to make an efficient partitioning of the resources between kernels (programs executed by the GPU) to maximize performance. However, its profiling strategy makes it agnostic to the interference between the kernels.

This dissertation proposes MIAS (Memory Interference Aware Scheduler), a thread block scheduling algorithm that tackles the issues Warped-Slicer has. It carries out a profiling phase where, unlike Warped-Slicer, it profiles kernels jointly, making them share the SMs during that phase. It profiles all the resource partitioning configurations (or thread block configurations) in parallel, in different SMs. It, then, chooses the best configuration using different metrics to estimate the interference between the profiled alternatives in the memory system, which they share. Evaluation results show that MIAS improves performance by 32% over sequential execution and 20% over Warped-Slicer.

# Resumen

En cada generación, las GPUs incorporan mayor cantidad de recursos computacionales, que se dedican a accelerar applicaciones paralelas en datos. Para utilizar dichos recursos eficientemente, las applicaciones tienen que exponer niveles altos de paralelismo. Aún cuando cumplen eso, es posible que no usen todos los recursos de los que despone la GPU (ancho de banda, memoria, unidades de ejecución de cada core, etc.).

No todas las aplicaciones pueden hacer un uso eficiente de una GPU, pero, si se ejecutan varias concurrentemente, puede que entre ellas compensen las deficiencias que cada una tiene. Esta es la idea detrás de la multiprogramación. Las GPUs actuales tienen soporte para ella, pero solo abordan la falta de paralelismo. Otros metodos, como Warped-Slicer, se han propuesto para reducir la infrautilización haciendo que las aplicaciones compartan los recursos de un mismo SM (Streaming Multiprocessor, que es el equivalente a un core de la GPU en la terminología de Nvidia). A esto se le denomina Intra-SM-Sharing. Warped-Slicer, que es el estado del arte en la planificacion de bloques de threads, intenta hacer un particionado eficiente de los recursos entre los kernels (programas ejecutados por la GPU) para maximizar el rendimiento. Sin embargo, su estrategia de profiling lo hace agnóstico a las interferencias entre los kernels.

Este trabajo propone MIAS (Memory Interference Aware Scheduler), un algoritmo de planificación de bloques de threads sin las limitaciones de Warped-Slicer. Para ello, MIAS lleva a cabo una fase de profiling donde, a diferencia de Warped-Slicer, realiza un profiling conjunto de los kernels, haciendo que compartan los SMs durante esta fase, y evaluando todas las configuraciones de particionado de recursos (o configuraciones de bloques de bloques de threads) en paralelo, en diferentes SMs. Después, escoge la mejor configuración usando diferentes metricas para estimar los efectos de las interferencias entre las alternativas que están en profiling. Los resultados de evaluacion muestran que MIAS mejora el rendimiento en un 32% respecto a la ejecución secuencial y en un 20% respecto a Warped-Slicer.

# Chapter 1

# Introduction

## 1.1 GPGPUs

CPUs have been the dominant architecture because of their ability to perform general purpose computations. Moreover, their performance has been improving with every generation, mainly due to a combination of advancements in chip manufacturing technologies, and architectural improvements.

However, those performance gains are slowing down. Performance no longer scales well with improvements to transistor technology, which itself is hitting hard limits, and considerable architectural improvements lead to smaller performance gains, usually for only a subset of applications. Regardless of that, there are always significant potential performance gains if optimizations for a specific kind of applications are made. Thus, domain specific architectures emerged. These are optimized to do a very specific kind of computations, very efficiently [1]. The most prominent of these architectures are Graphics Processing Units (GPUs), FPGAs[2], and Google's Tensor Processing Units (TPUs), which are only used for the kind of operations found in deep learning algorithms.

GPUs started as a domain specific architecture that could only do graphics. Early GPUs had very limited to no programmability, but they became more and more programmable over time to support a wider variety of graphics algorithms. In addition, those graphics applications are data-parallel, the main characteristic that GPUs exploit to boost performance [3, 4]. Therefore, they could be used to accelerate other data-parallel applications, which led manufacturers to add interfaces for general purpose programming [5, 6].

Current GPUs can perform a wide variety of computational tasks. In fact, they support a Turing-complete model, i.e. they can do any computation given enough time and memory [7, 8]. Performance-wise, however, they only excel at data-parallel applications. They perform poorly on other tasks [9, 10, 11, 12]. This is owing to their SIMT (Single Instruction Multiple Threads) execution model, which, in short, requires all executing threads to do the same operation on different data items to be efficient.

## 1.2 Multiprogramming

Since, GPUs have become quite versatile, and because of their excellence at data-parallel applications, programs from many areas are ported over to run on them. they are used for scientific, machine learning and graph applications, to name a few. Therefore, there is a lot of *heterogeneity* in the workloads that GPUs run. Different applications have different computational requirements, and stress the GPU in different ways, usually underutilizing its computational capacity. They may even have complementary behaviour.

Moreover, GPUs are incorporating more and more resources with every generation (more compute units, more memory bandwidth, etc) [3, 13, 14]. Using current GPUs to their full potential requires very big and massively parallelized workloads. However, This is not the case for many of the applications of GPUs. Furthermore, algorithms that are less suitable for GPUs are sometimes run on them in order to minimize CPU-GPU communication overheads [15].

To mitigate the inefficiencies mentioned previously, current GPUs use multiprogramming [16, 17, 18, 19, 20, 21, 22, 23, 14], which is the concurrent execution of different applications in the same GPU. This increases the occupancy of the GPU, and can balance the usage of resources. These two aspects increase performance and energy efficiency.

To support multiprogramming, GPUs have to simultaneously manage different workloads. To do that, a scheduling algorithm that decides which workload to schedule is needed. Current GPUs use the Left-over algorithm, which, as its name suggests, assigns resources to a kernel greedily until its requirements are fulfilled and only schedules a subsequent kernel if resources are remaining. However, this algorithm is far from optimal, leaving quite a margin for improvement. That is why research proposals like Warped-Slicer[18] have been made. Warped-Slicer makes different applications share and co-run on the same core of the GPU (in all cores). It tries to determine the best way to partition the resources of each core between them. For that, It carries out a profiling phase to profile the performance behaviour of the applications and use it to decide. It profiles each application separately (in deferent cores), which makes it agnostic about the interferences that may happen between them. This leads to suboptimal decisions. Therefore, there is still margin for improvement. This dissertation proposes MIAS (Memory Interference Aware Scheduler), the algorithm that tackles the issues that Warped-Slicer has.

## 1.3 Objectives

The main objective of this dissertation is to design a scheduling algorithm, MIAS, that can improve the performance of multiprogrammed workloads. The following smaller objectives are set as a roadmap to achieve the main one:

- Study currently proposed multiprogramming methods to gain a wider perspective on the subject and, especially, the state-of-the-art to be able to compare with it.
- Study GPGPU architecture in order to be able to understand its behaviour and derive conclusions about the performance of a certain workload run on GPUs.
- Gather a diverse benchmark set to perform small evaluations during development and for the final evaluation of the algorithm.
- Iteratively design the actual algorithm.
- Develop a simulated implementation of the algorithm, together with the baseline alternatives for evaluation and comparison.
- Evaluate the impact of the algorithm on performance and compare it with the baselines.

## 1.4 Work plan

Each of the small objectives mentioned in the previous section is considered as a task. They were carried out mostly simultaneously, except for final evaluation, which was done at the end of development. The design of the algorithm required knowing about the architecture of current GPUs and the current techniques of multiprogramming. Moreover, benchmarks had to be used to evaluate and test new ideas. To do so, the execution of the benchmarks had to be simulated, so significant time was dedicated to the development and execution of the simulations.

## 1.5 Document structure

In addition to this chapter, the dissertation has four more, with the following structure:

- Chapter 2: **Background.** Explains background concepts about GPGPUs used in the following chapters. It briefly covers the programming model and focuses on the architecture. It also introduces the baseline multiprogramming techniques to compare MIAS with.

- Chapter 3: **MIAS.** presents the design and the simulated implementation of MIAS, the proposed algorithm that improves upon the current state of the art in GPU multiprogramming. It also discusses some of the limitations that were faced during the design of the algorithm.

- Chapter 4: **Evaluation.** Discusses the evaluation of MIAS. It presents the evaluation methods and then the results. It compares MIAS with the baseline multiprogramming schemes, including the state-of-the-art. It also reasons about the insights behind the obtained results.

- Chapter 5: **Conclusions and future work.** Discusses conclusions about what has been accomplished so far, and presents future lines of work.

# Chapter 2

# Background

In this chapter, concepts and background related to GPGPUs, and specifically to thread block scheduling and GPU multiprogramming, will be introduced. First the programming model is explained and, then, microarchitecture concepts relevant to the topic of the dissertation are discussed.

## 2.1 CUDA programming model

CUDA [24] and OpenCL [25] are the most popular languages for general purpose programming on GPUs. There are alternatives such as SCYL or OneAPI, but they higher-level languages and focus on usability rather than performance [26, 27]. CUDA and OpenCL both share the same core ideas in their programming models. Unlike OpenCL, CUDA is exclusively used for Nvidia GPUs, and exposes all of their features. CUDA is chosen because the simulator used in this dissertation, called GPGPU-Sim[28], comes preconfigured for Nvidia GPUs, and it is much easier to use than the verbose OpenCL. However, the algorithm proposed in this dissertation should have the same effects regardless of the programming model.

The CUDA heterogeneous model defines two separate compute entities, host and device, each with its separate memory. This is called the host-device programming model.

### 2.1.1 Execution model

A CUDA program starts in the host, conventionally, as a sequential C/C++ program. Then, heavy data-parallel functions are offloaded to the GPU, as kernels. In this dissertation, host and device memory spaces are considered to be separate. Therefore, necessary data has to be sent in advance to the device, and retrieved afterwards.

### Kernels

In CUDA, computations that are run on the device are expressed as functions that are executed by thousands, or even millions, of threads in parallel, these functions are called *kernels*. Kernels are declared using the `__global__` keyword, as can be seen in code 2.1. The parameters of a kernel are the same for every thread. Inside the kernel, the `blockIdx`, `blockDim` and `threadIdx` variables are available. They are set by the runtime and help to identify each thread and the data it will work with.

Kernel execution is initiated by the host using the notation in code 2.2. This special call is called a *kernel launch*. It is similar to a C function call with extra parameters in between '<<<' and '>>>' which determine the configuration of the offload to the GPU. These parameters are the grid dimensions, thread block dimensions, and, optionally, the amount of shared memory assigned to each thread block (if not known at compile time) and the stream to which the kernel will be launched. They will be discussed in more detail in later sections.

```
__global__ void saxpy(int n, float a, float *x, float *y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) y[i] = a * x[i] + y[i];
}
```

Code 2.1: Example of a kernel in CUDA

```
//...
threads_per_block = 256;
num_blocks = 1000
saxpy<<<num_blocks, threads_per_block>>>(N, 2.0f, d_x, d_y);
//...
```

Code 2.2: Kernel launch example

### Thread organization

In a kernel launch, millions of threads, organized into a grid of blocks, are issued to the device. Each of those blocks is called a *thread block*, and is a group of threads that can communicate and synchronize with each other. All threads of a thread block are scheduled on the same core or SM (Streaming Multiprocessor), and they share its private resources, like shared memory. The maximum size of a thread block is usually at around 1024 to 2048 threads.

The blocks in a grid have a one, two or three-dimensional organization. Threads inside a block can also have similar organizations, independent of the grid dimensions. The multi-dimensional organization of threads makes indexing in applications that deal with matrices or volumes easier. Figure 2.1 illustrates a two-dimensional grid of thread blocks, having a 2×3 organization. However, neither the organization of thread blocks, nor the organization of threads within a block, are relevant to their scheduling, therefore, it is not covered in detail.

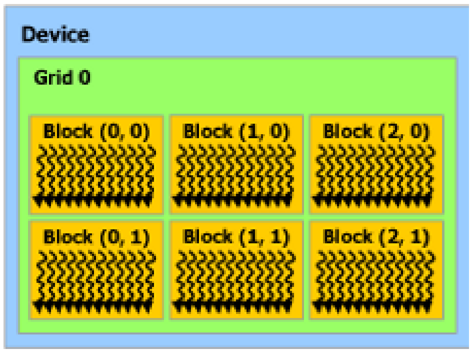


Figure 2.1: Illustration of a grid of thread blocks [15]

### The SIMT model

GPUs run threads in groups that execute instructions in lock-step, having the same PC, fetching and decoding one instruction per thread group each cycle. This way of execution is called SIMT (Single Instruction, Multiple Threads). In Nvidia GPUs, those thread groups are called *warps*, and they are 32 threads each.

CUDA Programmers should be aware of this Implicit SIMT model. It is one of the core ideas that make GPUs much more efficient than CPUs in data parallel applications. Control flow and memory access patterns should be adapted to it. Kernels that take advantage of it can be several times faster and more energy efficient [29].

### Streams

CUDA supports GPU multiprogramming via the *streams* API [15]. Stream behave as FIFO queues of operations. Usually, interdependent operations (kernels, memory transfers, etc.) are launched to the same stream. Independent operations can, and usually should, be launched to different streams to allow them to overlap execution or run in parallel.

### 2.1.2 Memory model

All Discrete GPUs have their own memory. For this reason, from the point of view of the programmer, CPU and GPU memory spaces are considered to be separate, which requires data transfers between both devices. However, this particular detail is not relevant to the topic of the dissertation. Consequently, the rest of the section focuses on the memory of the GPU.

In the device side, CUDA distinguishes between five memory spaces: global, local, constant, texture and shared memory [15]. The first four are all located on the off-chip DRAM and have similar latency behaviour. Therefore, in this dissertation, only global memory is considered when discussing the off-chip DRAM.

Global memory is the space where data is initially located when it gets transferred from the host. It has relatively low bandwidth and long latencies. As discussed in section 2.2.5, caches are used to reduce latencies and increase bandwidth. However, this is transparent to the programmer since the hardware manages them.

Shared memory is a scratchpad memory that is managed by the programmer. It is located in the SM and, when used correctly, delivers more bandwidth than the other memory spaces. Shared memory variables are declared using the keyword `__shared__`. These variables are only visible to and shared by threads in the same thread block, i.e., there is one instance of the variable per thread block, hence the name shared memory. Code 2.3 is an example of a kernel that uses shared memory.

Usually the use of shared memory implies thread synchronization to control the use of shared memory by the different threads of a thread block. This is usually carried out using barriers like the `__syncthreads()` call in code 2.3

```
// m, n and k are assumed to be multiples of BLOCK_SIZE
__global__ void matmul(int *d_a,int *d_b,int *d_result,int m,int n,
    int k) {
    // Declare shared memory variables
    __shared__ int tile_a[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ int tile_b[BLOCK_SIZE][BLOCK_SIZE];

    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
    int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    int tmp = 0;
    int idx;
    int chunks=n/BLOCK_SIZE;

    for (int sub = 0;sub < chunks; ++sub) {
        // Load data to shared memory
        idx = row * n + sub * BLOCK_SIZE + threadIdx.x;
        tile_a[threadIdx.y][threadIdx.x] = d_a[idx];
        idx = (sub * BLOCK_SIZE + threadIdx.y) * k + col;
        tile_b[threadIdx.y][threadIdx.x] = d_b[idx];
        // Make sure all data is loaded before computations
        __syncthreads();
        for (int i = 0; i < BLOCK_SIZE; ++i)
            tmp += tile_a[threadIdx.y][i] * tile_b[i][threadIdx.x];
        // Make sure all computations have finished before
        // overwriting shared data
        __syncthreads();
    }
    d_result[row * k + col] = tmp;
}
```

Code 2.3: Shared memory example: A simplified matrix multiplication

## 2.2 GPGPU Architecture

GPUs have a throughput oriented architecture. Their memory systems are designed to maximize bandwidth and their execution cores are optimized for operation throughput. In this section, the architecture will be explained in a top-down approach. First, an overview of the structure of a GPU is presented, and then each component will be explained in more or less detail depending on its relevance to this dissertation.

### 2.2.1 High level overview

Like multicore CPUs, GPUs have a set of compute units (cores), which in Nvidia and CUDA's terminology they are called Streaming Multiprocessor (SM), connected via a NoC (Network on Chip) to multiple memory partitions. A memory partition consists of a share of L2 cache and a memory controller. Each SM can accommodate a number of thread blocks depending on their resource requirements. A hardware thread block scheduler assigns thread blocks to the SMs, one block at a time, in round-robin order. This structure is illustrated in the diagram on the left in figure 2.2

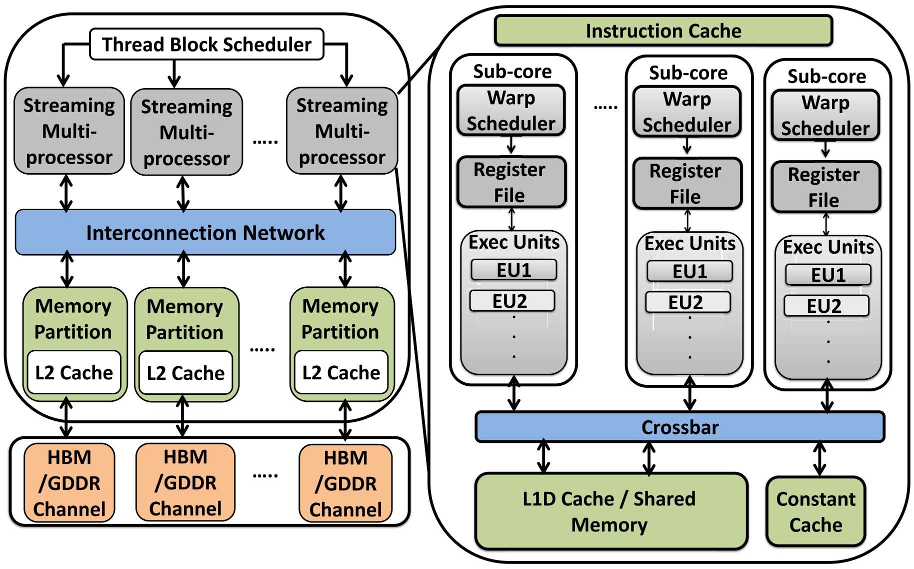


Figure 2.2: Illustration of the architecture of a GPU [28]

### 2.2.2 NoC

Current Nvidia GPUs use two crossbars to connect SMs with memory partitions, one for traffic from SMs to memory and the other for traffic in the other direction. This kind of network between the SMs and Memory partitions supply the SMs with the large memory bandwidth they need. Different SMs can communicate with different memory partitions in parallel. However, some conflicts can occur if two or more SMs communicate with the same memory partition at the same time or vice versa, in that case, communication is serialized.

### 2.2.3 Memory Partition

A memory partition has a portion of the L2 cache and one or more memory controllers that connect it to the off-chip DRAM. The memory controller, schedules DRAM accesses in a way that increases row buffer locality. Some algorithms to achieve that are FR-FCFS and others proposed for multi-application execution [30].

### 2.2.4 Streaming Multiprocessor

GPU SMs use two key Ideas to increase performance. SIMT execution for more efficiency and throughput, and aggressive multithreading to hide memory and execution unit latencies. Figure 2.2 depicts a breakdown of an SM on the right.

### SIMT model

In SIMT execution, only one instruction is fetched and decoded for each warp. Then, it is executed as a SIMD instruction on data items from all the scalar threads. This saves energy and area and allows more of it to be dedicated to execution units, since the control logic is shared between all threads of a warp.

There is an issue with this model, the CUDA programming model does not restrict threads of the same warp to always execute the same instruction. They can follow different control paths in an if-else statement, for instance. This issue is called *thread divergence*. One way it is solved is by using predication masks, like in vector processors[1]. A thread's lane is active only when its corresponding bit is set. More advanced ways are used to support complex control flow[7] in GPUs.

### Pipeline

Compared to the standard CPUs found in most computers, The pipeline of an SM is simpler, no out of order execution is used. The pipeline has the following 6 stages:

1. **Instruction fetch:** Like in CPUs, this stage brings instructions from memory.
2. **Instruction decode:** Fetched instructions are decoded and placed into the instruction buffer ready to be scheduled for execution.
3. **Warp scheduling:** The warp scheduler, then, selects a ready warp and issues from its instructions to the corresponding execution pipelines. Current GPUs use the GTO (Greedy Then Oldest) algorithm[31].
4. **Operand collection:** Operands of issued warp instructions are read from the massive register file[32, 33, 34].
5. **Instruction execution:** At this stage, instructions are executed. Different instructions can have different execution latencies. Execution units can be further segmented, increasing the depth of the pipeline.
6. **Write back:** The last step is writing the result back to the register file.

### Memory pipeline

Memory instructions are executed, with each thread specifying its own address. A warp instruction can produce up to 32 memory accesses. The memory pipeline has a coalescer which joins consecutive accesses into one access before sending it to the L1 cache. If all threads access consecutive and aligned data items, just one access to the L1 cache is made. If accesses are not coalesced, the memory pipeline may stall and the memory system may get saturated with fewer memory instructions.

**Sub-cores**

In contemporary GPUs, an SM is made up of simpler cores, each executing a portion of warps assigned to the SM, with its separate instruction buffer, warp scheduler, register file and execution units. They, however, share caches (Instruction and data), shared memory and can synchronize with each other. Usually, the number of warp-schedulers is used instead of the number of sub-cores, since every sub-core has one warp-scheduler.

### 2.2.5 Memory hierarchy

Apart from the use of multithreading to hide memory latencies, GPUs employ caches as well. Usually, they have two cache levels, and the DRAM as the lowest level in the memory hierarchy. Those levels are explained next:

**L1 cache.** It is the highest level in the memory hierarchy and the fastest (After the register file, which is sometimes considered a level in the memory hierarchy). Each SM has its own private L1 cache. No cache coherency is maintained, i.e. different SMs can have different data in the same variable (same memory address). An SM has multiple L1 caches (instruction cache, constant cache, etc). However, this dissertation only focuses on the data cache.

**L2 cache.** It is bigger, but slower, than the L1 cache. It is shared by all SMs and connected to them via the NoC. It is distributed across memory partitions.

**DRAM.** It is the lowest level and the slowest. Its latencies are variable because of refreshes and the use of the row buffer (a memory buffer to which a row of the DRAM matrix is brought before smaller items are read).

To hide memory latencies, GPUs have to be able to continue issuing instructions even if a miss happens. GPU caches allow a number of misses to be outstanding at the same time. This is achieved by the MSHRs (Miss Status Holding Register) table, which is indexed and tagged by parts of the address, like caches are. It holds information to determine which accesses correspond to which memory requests. It also allows for access merging: If two or more accesses are to the same cache line, then only one memory request is generated.

### 2.2.6 Thread block scheduler

The thread block scheduler, assigns thread blocks from kernels to SMs. First, it computes the maximum number of thread blocks that can fit in each SM, which depends on the resources that the kernel to be scheduled needs. Then, it starts issuing thread blocks, one to an SM each cycle, using a round-robin policy. When doing so, the resources required by the thread block are allocated. These include: hardware threads/warps, registers, shared memory, and hardware thread blocks (per thread block resources). The number of thread blocks assigned to an SM can be limited by any of these resources. In fact, usually, more thread blocks than the SMs can accommodate are launched. In that case, the remaining thread blocks are issued as resources become available when previous thread blocks finish.

Thread scheduling in GPUs is done in two levels. The first is thread block scheduling and the second is warp scheduling. When the thread block scheduler issues a thread block, it assigns its warps (logical ones) to hardware warps. The warp scheduler schedules the execution of the warps and issues their instructions to be executed in the execution units (ALUs, FPUs, etc). Warp scheduling happens in the issue stage of the pipeline in every SM (section 2.2.4). In contrast, thread block scheduling happens outside the SMs and there is just one thread block scheduler in all the GPU.

In the context of GPU multiprogramming, there are many ways in which thread blocks from different kernels can be scheduled. This can have a significant impact on performance. Next, in section 2.3 some of these methods will be explained.

## 2.3 Related work

This dissertation proposes an algorithm that tries to maximize throughput by efficient multiprogramming. GPU multiprogramming is the concurrent execution of different kernels in a GPU. It aims to use the resources of the GPU more efficiently. Many scheduling algorithms for multiprogramming have been proposed, some of them are explained next.

Left-Over is the algorithm implemented in current GPUs. it runs kernels in the order they were launched. thread blocks from one kernel are assigned to SMs until they run out. Then, it starts issuing thread blocks from the next, if there are enough remaining resources or as they become available. It is meant to mitigate the lack of thread level parallelism (TLP) in kernels, but it does not co-run them if they have enough TLP except briefly at the end of execution of each kernel.

Spatial Multitasking[16] makes kernels share resources at the SM granularity. Each kernel is assigned a set of SMs where only thread blocks from it are run. This mitigates the lack of TLP in kernels. Moreover, it can give performance gains if the co-run kernels have different memory behaviour.

Other methods use Intra-SM sharing, which is the concurrent execution of thread blocks from different kernels in the Same SM. Most of them try to partition resources among the kernels in a way that achieves better performance and/or fairness. Some of them approach that at the thread block scheduling level [17, 18, 19, 20], while others try to solve it at the warp scheduling level or both levels [21, 22, 23].

Warped-Slicer [18] is the state of the art for Intra-SM sharing. It uses information about how performance scales with increasing number of thread blocks of each kernel to predict the configuration of thread blocks (the number of thread blocks of each kernel to run concurrently) that minimizes performance loss for both kernels when they co-run.

First, it carries out a profiling phase to get the performance scalability information. All thread block numbers from 1 to the maximum possible on an SM are scheduled on different SMs for each kernel. For instance, if kernel $K_0$ has a maximum of 3 thread blocks and kernel $K_1$ can fit 4 thread blocks, then $SM_0$, $SM_1$ and $SM_2$ run 1, 2, and 3 thread blocks respectively from kernel $K_0$, while $SM_3$, $SM_4$, $SM_5$ and $SM_6$ run 1, 2, 3 and 4 thread blocks from kernel $K_1$. Figure 2.3 illustrates the profiling strategy, but with both kernels having a maximum of 4 thread blocks per SM. Once the profiling phase is over, the performance (IPC) achieved by each SM is gathered and used to predict the best configuration.

An algorithm similar to the water filling one [35] is used to determine the best configuration. It starts by reserving resources for one thread block of every kernel. To minimize the performance loss suffered by each one, it tries to compensate the kernel with the lowest performance, by reserving resources for more of its thread blocks to level it up with the rest. It does so iteratively until resources run out. The algorithm falls back to Spatial Multitasking if it predicts that kernels will lose too much performance as a result of intra-SM sharing. Figure 2.3 illustrates the previously described steps.

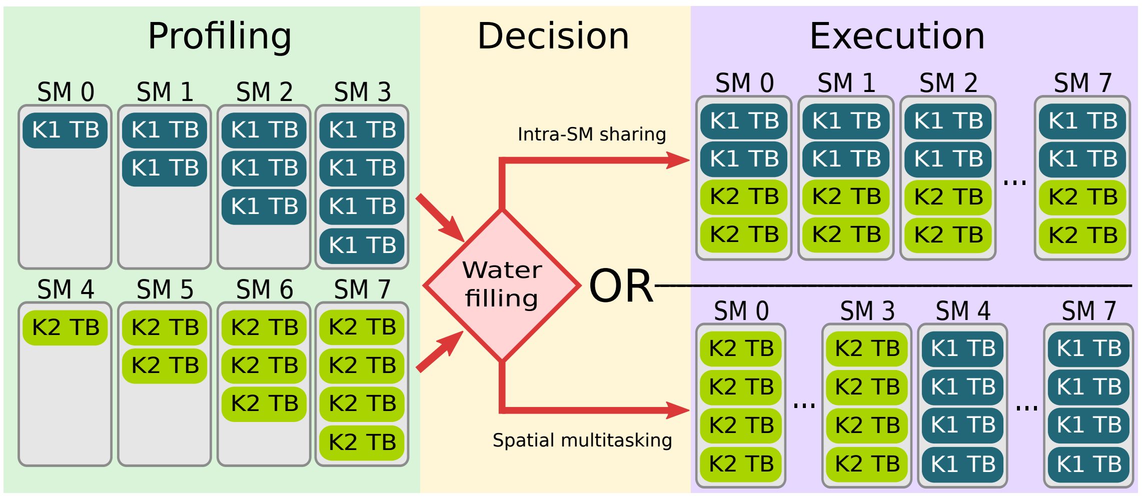


Figure 2.3: Illustration of the Warped-Slicer algorithm.

# Chapter 3

# MIAS

This chapter will explain MIAS (Memory Interference Aware Scheduler), the proposed algorithm for thread block scheduling. It starts with a motivation. Then, the algorithm is explained. Afterwards, the implementation in GPGPU-Sim[28] simulator is covered.

## 3.1 Motivation

As of the writing of this dissertation, The state of the art for thread block scheduling is Warped-Slicer[18]. Its profiling strategy, explained in section 2.3, has the following two main issues:

**Kernels profiled separately** which makes the profiling process agnostic to the impact of interferences between kernels on performance, especially due to contention in the L1 cache.

**Lower occupancy during profiling** this considerably lowers the stress put on the memory system during profiling, leading to very inaccurate performance measurements for memory intensive kernels. Moreover, computational resources are also underutilized during profiling, which makes it (the profiling phase) more costly performance-wise.

Figure 3.1 shows the performance of Warped-Slicer for some example kernel combinations. The y-axis represents the speedup (a metric explained in section 4.1.4). It has four bars, each indicating the speedup that corresponds to the kernel combination indicated by the x-axis. Below every kernel combination label, the type of its kernels is

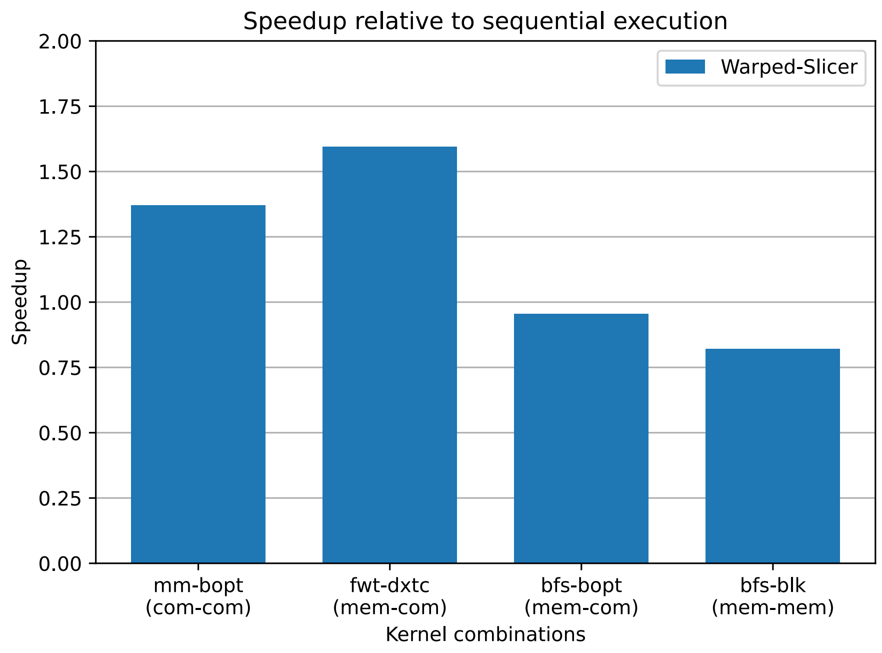


Figure 3.1: Examples of the performance of Warped-Slicer

indicated. “com” and “mem” respectively stand for compute intensive and memory intensive. The kernels are Matrix Multiplication [15] (mm), Binomial Options [15] (bopt), Fast Walsh Transform [15] (fwt), DXTC [15] (dxtc), Breadth First Search [36] (bfs) and Black Scholes [15] (blk).

These results will be further explained in section 4.2. They are used as examples of the behaviour of Warped-Slicer here. Warped-Slicer performs well when all combined kernels are compute intensive (mm-bopt combination in figure 3.1). Only the separate profiling issue affects this kind of combinations. But still, it achieves results that are better than running the kernels sequentially.

Warped-Slicer does not perform as well when memory intensive kernels are involved. The low stress on memory during profiling makes the performance measurements of this kind of kernels very inaccurate. Although a technique is used to adjust the measured values and take into account memory behaviour, experimentation results showed that it does not make much of a difference in the baseline GPU used in this dissertation (section 4.1.1). For some memory-compute combinations like fwt-dxtc it achieves great speedups of up to 1.6. However, for others, like bfs-bopt, its performance is less than sequential execution. The results are worse for memory-memory combinations. All the speedups of Warped-Slicer are less than 1 when dealing with that type of combinations in the experimentation done for this dissertation.

Although Warped-Slicer outperforms the other thread block scheduling policies covered in section 2.3. It still leaves a considerable margin for improvement. GPUs are growing in complexity with each generation of architecture [3, 13], which makes their performance less and less predictable. Hence, MIAS is proposed, an algorithm that creates profiling conditions that are as close to normal post-profiling execution as possible. This can make accurate performance evaluations to better decide thread block configurations.

## 3.2 Algorithm design

This section explains the design of MIAS. An overview of the algorithm is given, then, its steps are explained in depth.

The main objective of MIAS is to maximize the throughput of the GPU when multiple kernels run on the same device. It tries to select the best performing thread block configuration (How many thread blocks of each kernel) to schedule on the SMs. It is expected to achieve results close to the Ideal case, where the performance of each thread block configuration is known beforehand.

MIAS has three main steps: *profiling*, *decision* and *broadcasting*. Figure 3.2 illustrates these steps, which are also explained next:

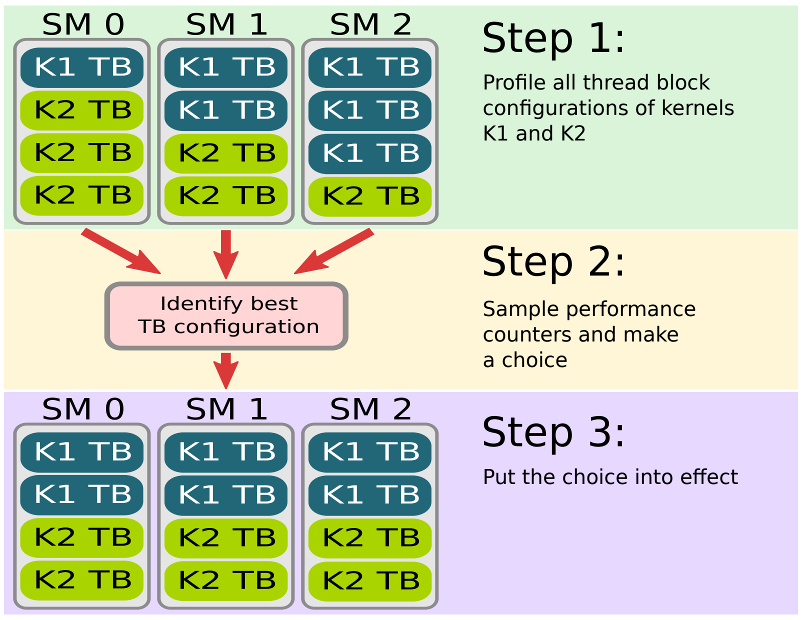

SM 0
SM 1
SM 2
K1 TB
K2 TB
Step 1:
Profile all thread block configurations of kernels K1 and K2
Identify best TB configuration
Step 2:
Sample performance counters and make a choice
Step 3:
Put the choice into effect

Figure 3.2: Illustration of the steps of MIAS.

1. **Profiling:** The strategy used in MIAS makes kernels execute inside the SMs, exactly how they do in post-profiling execution. All the possible thread block config-

urations (Ways of partitioning SM resources between kernels) to be scheduled are generated, and issued to different SMs. They are let to run for enough time to allow performance to stabilize. Hardware performance counters are used to collect and measure certain metrics in each SM.

2. **Decision:** Once the profiling period is over, the values collected in the performance counters are used to determine the best performing thread block configuration. Estimations of the effect of memory interference are made, and the measured performance is adjusted accordingly.

3. **Broadcasting:** The chosen thread block configuration is broadcast to all SMs. Thus, the post-profiling execution starts. It continues until either a new kernel is launched or one of the kernels finish execution. In both cases, rescheduling is done by repeating the tree steps.

### 3.2.1 Profiling

In a multiprogramming scenario with intra-SM sharing, kernels can interfere with each other, either constructively or destructively. The constructive interference happens when kernels are more intensive in different resources. Combining them in good proportions leads to a more balanced workload, which increases the throughput. The destructive interferences happen when both or one of the kernels uses a certain resource too intensively, causing contention or saturation. Contention mainly occurs in the L1 data cache when kernels start to evict each other's data due to conflict misses. Saturation takes place mostly in the memory pipeline when too many memory requests are generated, causing it to stall.

Unlike Warped-Slicer, which profiles kernels separately, the profiling strategy used in MIAS makes kernels share the SM so that the interferences between them take place, and affect performance as they would do in post-profiling execution. All candidate thread block configurations are run in parallel, using different SMs, to reduce the time it would take to finish profiling. Step 1 in figure 3.2 illustrates this strategy.

For the generation of thread block configurations, the algorithm 1 is proposed. It has the following two main steps:

1. **Recursive assignation of resources to kernels**. At each call, it incrementally reserves resources for thread blocks of a kernel and makes a call to partition the rest

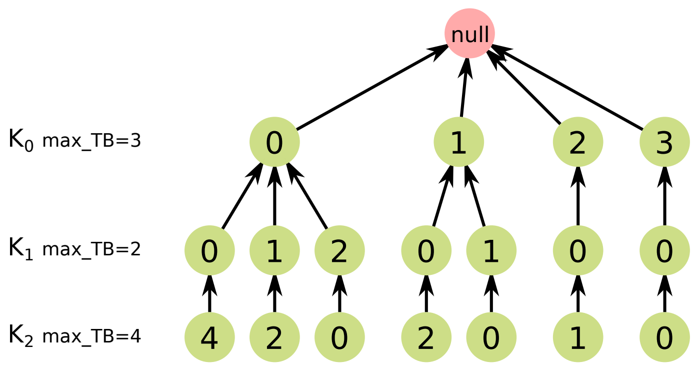


Figure 3.3: An example of the tree produced during the generation of thread block configurations.

between the remaining kernels. This step generates a tree, where each path from a leaf to the root corresponds to a thread block configuration. Figure 3.3 illustrates the structure of the tree with an example with three kernels, each one can fit the number of thread blocks indicated by `max_TB` in one SM. The first leaf in the left represents the (0, 0, 4) configuration (0 TBs of kernel $K_0$, 0 TBs of kernel $K_1$ and 4 TBs of kernel $K_2$). Each node points to its parent to allow for an easy extraction of the configurations.

2. **Extraction of thread block configurations from the tree**. The paths from each leaf to the root correspond to the configurations. Pointer chasing is carried out to extract them. It also checks for and removes incomplete configurations, i.e. the ones that leave enough resources for more thread blocks from one of the kernels. This can occur if the kernels are limited by different resources (section 2.2.6). later reserved thread blocks leave room for previously assigned ones because they do not consume the resources that limit them.

Single kernel thread block configurations (where only thread blocks of one kernel are executed in the SM) are not considered when two or more kernels are available for execution. The algorithm always does intra-SM sharing. The configurations returned by algorithm 1 are filtered to remove the single kernel ones.

The amount of cycles dedicated to profiling should be as short as possible, but enough for the performance to stabilize. An experiment has been carried out on the benchmarks to determine an adequate length for the profiling phase. The execution of the benchmarks (covered in section 4.1.2) was simulated, with periodic measurements of the IPC (Instructions Per Cycle). Results are plotted in figure 3.4. The IPC of most kernels seems to stabilize at around 45000 to 50000 cycles. Therefore, the length of 45000 cycles was

**Algorithm 1** The algorithm to generate all possible thread block configurations

```
function GENERATE_TB_CONFIGURATIONS($kernels$)
    $L \leftarrow \emptyset$                                    ▷ The leaves of the configuration tree.

    ▷ A recursive procedure to recursively generate the configuration tree.            ◁
    procedure GENERATE_SUBTREE($parent, remaining\_resources, k$)
        $max\_TBs \leftarrow$ TBs of $kernels[k]$ to fill $remaining\_resources$

        if $k = |kernels| - 1$ then
            ▷ Assign remainig resources to the remaining kernel.                        ◁
            $L \leftarrow L \cup \{Node(parent, max\_TBs)\}$
            return

        for $nTBs \leftarrow 0$ to $max\_TBs$ do
            $node \leftarrow Node(parent, nTBs)$
            $r \leftarrow$ Resources for $nTBs$ of $kernels[k]$
            GENERATE_SUBTREE($node, remaining\_resources - r, k+1$)

    $resources \leftarrow$ The resources of the SM

    ▷ After this, L contains all the leaves of the configuration tree.                  ◁
    GENERATE_SUBTREE($null, resources, 0$)

    $configurations \leftarrow \emptyset$
    for $node \in L$ do

        $k \leftarrow |kernels| - 1$
        $config \leftarrow$ A list with size $|kernels|$
        $resources \leftarrow$ The resources of the SM   ▷ Used to check for incomplete configs

        ▷ Chase the path to the root to get the configuration.                          ◁
        while $node \neq null$ do
            $config[k] \leftarrow node.nTBs$
            $r \leftarrow$ Resources for $node.nTBs$ of $kernels[k]$
            $resources \leftarrow resources - r$
            $k \leftarrow k - 1$
            $node \leftarrow node.parent$

        ▷ Check if the configuration fills the SM as much as possible.                  ◁
        $complete \leftarrow true$
        for $k \leftarrow 0$ to $|kernels|$ do
            if $resources$ are enough for one more TB of kernel $kernels[k]$ then
                $complete \leftarrow false$
                exit for loop

        if complete then
            $configurations \leftarrow configurations \cup \{config\}$

    return $configurations$
```

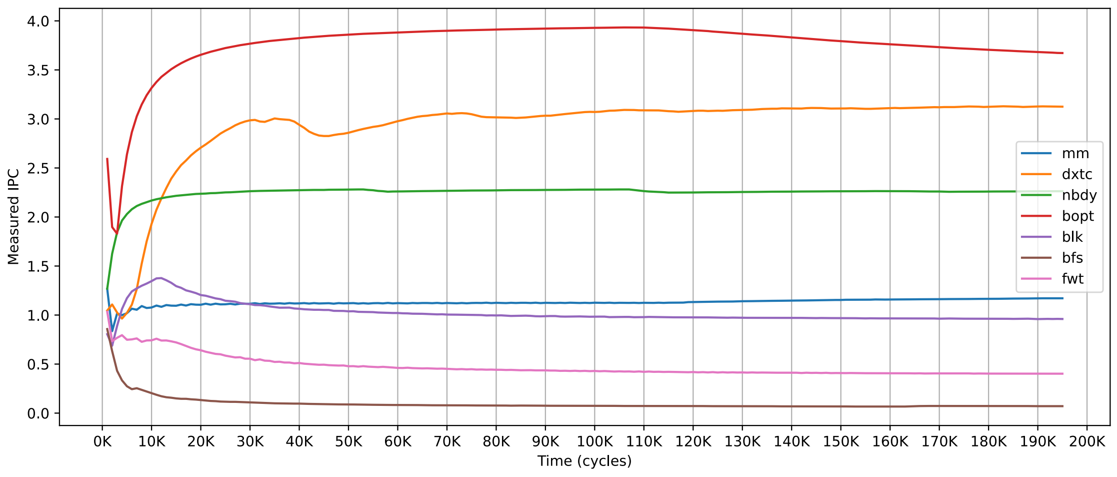


Figure 3.4: The IPC measured as execution progresses. The x-axis indicates the time in cycles. The y-axis indicates the IPC. Each curve corresponds to the evolution of the IPC of the corresponding benchmark over time.

chosen for the profiling phase of the experiments of this dissertation. Nevertheless, if other applications are used, their stabilisation threshold should be measured. This is a configurable parameter of the algorithm. It takes that much for the performance to stabilize due to the following reasons:

- Caches have to be warmed up (filled with data).
- The memory system can handle thousands of outstanding accesses which have to be made before latencies stabilize
- Compute intensive kernels that rely on shared memory start by fetching data to it from global memory, which produces memory intensive behaviour.

This strategy takes into account every thing that can happen inside the SMs. However, The memory system outside (NoC, L2 cache, Memory controllers and DRAM) is shared by all the profiled configurations. This lets them interfere in each other's performance during profiling, which produces inaccuracies in the measurements. The next section (3.2.2) discusses proposed techniques to mitigate this issue.

### 3.2.2 Decision

Once the profiling phase is over, hardware counters for number of cycles, instructions executed, stalls, L1 cache misses, accesses, etc... are read and used to decide what thread block configuration will give the greatest IPC. The IPC is affected by three factors:

1. **Control flow and thread synchronization**, which prevent some warps from issuing instructions until, for instance, others get to a barrier.

2. **Instruction diversity**, which allows instructions to overlap more as they execute on different execution units.

3. **Memory latency**, which causes warps to wait for a long time while data is fetched from memory.

Three methods are proposed to mitigate the issue mentioned at the end of the previous section (3.2.1), apart from the trivial method of choosing the configuration with the greatest IPC. They differ in the metric they use to choose the best thread block configuration. They are all original ideas of this dissertation. Therefore, there are four variants of MIAS. Each one uses a different method at this step. They will all be evaluated in the next chapter and the best performing variant is kept. Next, these variants are explained.

### The IPC metric

The IPC, Instructions per Cycle, is the performance metric we are trying to maximize. It is computed as follows:

$$IPC = \frac{I}{C} \tag{3.1}$$

Where $I$ is the number of executed instructions and $C$ is the number of cycles. It probably is the variant with the simplest hardware implementation.

### The IPM*IPC metric

Memory accesses that miss in the L1 cache are sent to the memory system outside the SM, and are the cause for the variability of memory latency. The SM continues to execute until all instructions from all active warps are waiting for data from memory. The more instructions can be executed after a miss, the more of its latency is hidden. One metric that can approximate this property is the $IPM_{L1}$ (Instructions per L1 miss). The proposed metric in this case combines it with the $IPC$, which has the information about synchronization behaviour and instruction diversity. This metric is computed as follows:

$$S = IPM_{L1} \cdot IPC \tag{3.2}$$

L1 misses are the accesses responsible for long stalls in the SM due to memory latency. However, it is not enough to know how many misses happen on average per instruction (The inverse of IPM). If cache misses happen sporadically, The SM can cover them by multithreading and may never stall. But, if the same amount of misses takes place in a burst, they will probably cause the SM to stall for a long time. The next metric is meant to tackle this issue.

### The Factor metric

The number of outstanding memory accesses increases memory latency [21]. If kernels with different memory behaviour are co-run, each thread block configuration will have different behaviour. When run together during profiling, memory experiences a stress that could be approximated as the average of all the configurations. When the chosen configuration is replicated on all SMs, the stress on the memory system can increase or decrease, leading to different performance with respect to the profiling phase.

One way to account for that, is by measuring the stress one SM is putting on memory compared to the overall, and use it to predict the change that will happen to memory stall cycles once that SM's configuration is broadcast. The difference in memory stress can be represented by a factor:

$$f = \frac{\text{Memory stress produced by } SM_i}{\text{Memory stress produced by all SMs}} \tag{3.3}$$

If this factor is greater than 1.0, it means that the thread block configuration on $SM_i$ will increase the memory load in the post-profiling phase. The opposite happens if the factor is less than 1.0.

The cycles lost due to memory latencies can be multiplied by this factor to increase or decrease them accordingly. Then, the IPC can be computed using:

$$IPC_{SM_i} = \frac{I_{SM_i}}{C_{SM_i} + M_{SM_i} \cdot (f_{SM_i} - 1)} \tag{3.4}$$

Where $I$ is the number of instructions executed, $C$ the total cycles, and $M$ the memory stall cycles, of the profiling phase, in each $SM_i$. The intuition behind this formula is to compute the IPC, adjusting the number of cycles to what it would be under normal execution. The adjustment is done by increasing or decreasing the number of cycles lost due to memory latencies by an amount determined by the factor $f$, hence, the term $M \cdot (f-1)$. $(f-1)$ is used instead of $f$ because the formula 3.4 is derived from the next formula where memory stall cycles are treated separately (extracted from the total and then $f$ is applied):

$$IPC_{SM_i} = \frac{I_{SM_i}}{(C_{SM_i} - M_{SM_i}) + M_{SM_i} \cdot f_{SM_i}} \tag{3.5}$$

Now, what should be addressed is how to quantify the memory stress from the SM's perspective. In order to measure the accesses responsible for memory stalls, every time the SM stalls because of memory, the number of outstanding accesses gets accumulated in a register. The accumulated number of outstanding accesses is an indicator of how much stress the SM puts on memory. The factor can now be calculated as follows:

$$f_{SM_i} = \frac{A_{SM_i} \cdot |SM|}{\sum_{j=0}^{|SM|-1} A_{SM_j}} \tag{3.6}$$

Where $A_{SM_i}$ is the accumulated outstanding accesses on $SM_i$ and $|SM|$ is the number of SMs of the GPU. This is the formal version of expression 3.3. The numerator represents the stress that the configuration executing in $SM_i$ would put if it was broadcast to all SMs. The denominator is the sum of the stress put by every SM on memory during the profiling phase. The resulting factor indicates how the stress would change post-profiling.

### The Linear metric

The method described in the previous section overestimates the memory stall cycles for thread block configurations that put a load greater than the average of all SMs, and underestimates for their counterparts. To get a better approximation, the following linear regression of memory stall cycles as a function of accumulated outstanding accesses was used:

$$M = coef \cdot A + b \tag{3.7}$$

Where $coef$ is the coefficient of the linear regression and $b$ is the bias term. $A$ is the accumulated outstanding accesses.

The variation in stall cycles is approximated using the derivative of the function (i.e. only the $coef$ parameter), because the reference point (current memory stall cycles and current accumulated outstanding accesses) already provides us with a bias term that is better than the average one (the one that comes with the function). The change in memory stall cycles is computed as follows:

$$\Delta M_{SM_i} = coef \cdot \left( A_{SM_i} \cdot |SM| - \sum_{j=0}^{|SM|-1} A_{SM_j} \right) \tag{3.8}$$

The term $|SM| \cdot A_{SM_i}$ approximates the stress that the configuration executing in $SM_i$ would put on memory when it is broadcast. The term $\sum_{j=0}^{|SM|-1} A_{SM_j}$ is an estimation of the stress that was put on memory during profiling. The difference between these two, is the estimated change that will happen to memory stress. When it is multiplied by the coefficient $coef$ (the derivative of the linear regression), it estimates the change that will happen to the stall cycles induced by memory latency.

With this model, the IPC that each thread block configuration would have post-profiling is estimated, and used to choose the best performing one. The IPC is computed as follows:

$$IPC_{SM_i} = \frac{I_{SM_i}}{C_{SM_i} + \Delta M_{SM_i}} \tag{3.9}$$

This equation is similar to equation 3.4. The estimated change to the memory stall cycles is applied by adding it to the total cycles, and the IPC is computed using that.

### 3.2.3 Broadcasting

This step is the simplest out of the three. Once the previous step is done, the thread block configuration that is estimated to perform the best is issued to be applied on all SMs.

### 3.2.4 The complexity of the memory system and latency prediction

The methods described in section 3.2.2 try to predict memory latency variations using mostly the rate at which memory requests are generated by an SM. However, between the SMs and DRAM, there are components that exploit locality and access patterns to cut latencies:

**NoC and memory partitions:** The network on a chip can have variable latencies depending on access pattern and the way addresses are mapped to memory partitions. The former depends on the hardware implementation. The first is a property of kernels or TB configurations running on the SMs. NoC latency increases with the number of conflicts.

**L2 cache:** accesses with greater locality can have their memory latencies considerably reduced by the L2 cache. However, it is shared by all the SMs and co-running kernels interfere a lot in it as they can evict each other's data.

**Row buffer:** DRAM's data is organized into a matrix. Entire rows have to be read to the row buffer and, then, smaller data items can be read from the latter. Accesses that are ready in the row buffer have reduced latency.

Some techniques have been developed to evaluate the locality of programs and even GPU kernels[37, 38]. However, they all need traces of accessed memory addresses, which is too expensive to implement in hardware. In summary, the methods proposed for the decision step can be improved by incorporating locality and access pattern information (of L2, row buffer, etc. L1 is excluded) in their estimations. However, this is too complex and costly to implement in GPU architectures.

## 3.3 Implementation in GPGPU-Sim

To evaluate the impact of the proposed algorithm on performance, it was implemented in the GPGPU-Sim simulator[28]. In this section, the simulator is first introduced, and then the implementation is discussed.

### 3.3.1 GPGPU-Sim

GPGPU-Sim[28][1] is a cycle accurate simulator for GPGPUs, one of the most widely used in the scientific literature. It models the architecture described in section 2.2. It can run both CUDA and OpenCL kernels. Most features of the architecture are configurable and don't need any changes to the source code. Its configuration is specified in the `gpgpusim.config` file with option-value pairs like `"-gpgpu_scheduler gto"` to set the warp scheduling algorithm to GTO, for instance. It comes with configurations that were fine-tuned, using the Accel-Sim framework[28], to quite accurately simulate a handful of real GPUs. However, to implement new techniques, code modifications may be needed. This is the case of this dissertation.

GPGPU-Sim intercepts CUDA calls, such us `cudaMemCpy()` and kernel launches, from running programs and simulates them. Programs have to be dynamically linked to

[1]https://github.com/gpgpu-sim/gpgpu-sim_distribution.git

the CUDA runtime, and the shell environment has to be set up to redirect the calls to the simulator. Configuration files must be placed in the working directory of the simulated program.

### 3.3.2 Pre-existing thread block scheduling implementation

Thread block scheduling was not configurable for the most part. Left Over (section 2.3) was the only algorithm implemented, since it is what current GPUs use. The code for it was spread across two classes: `gpgpu_sim` which is the class containing all the simulated components of the GPU and the `shader_core_ctx` which simulates an SM. Every simulated cycle, the `gpgpu_sim::cycle()` method, i.e. the `cycle()` method of the `gpgpu_sim` class is called to simulate a GPU cycle. It, then, calls the `gpgpu_sim::issue_block2core()` which tries to issue one block to one SM iterating over them in round-robin order. For each one, it calls its `shader_core_ctx::issue_block2core()` until one of them accepts a block, when the SM has enough free resources. The former calls `gpgpu_sim::select_kernel()` which selects and returns a kernel to issue the thread block from.

This implementation is pretty much hardcoded for the Leftover policy. Moreover, more advanced algorithms would require processing information related to thread block scheduling, distributed across multiple classes, leading to unnecessary complexity.

### 3.3.3 A self-contained and extendable implementation

After familiarizing with the Simulator and gaining more insight into its inner workings, a self-contained and extendable implementation was made. A class called `tb_scheduler` (thread block scheduler) contains all the variables that are related to thread block scheduling, common to all the SMs. Also, it implements common functionality to all thread block scheduling algorithms. It has a list of `intra_sm_slicer` objects. Each tracks the thread block scheduling state on its corresponding SM, and manages resource allocation on it.

Two lists of kernel entries are used in the `tb_scheduler` class: One for kernels launched and issued to the hardware thread block scheduler, and one for kernels that are scheduled, or to be scheduled, on the SMs. Another list is used to hold thread block configurations. Each thread block configuration is a vector of integers the same size as the scheduled kernels list. The position *i* of that vector corresponds to kernel *i* in the scheduled kernels list.

Every thread block configuration is scheduled in an SM. If there are more SMs than configurations, they can be scheduled on multiple SMs. It also handles the profiling logic, which is common to all algorithms that use it. It profiles whatever is executing on the SMs, which is determined by downstream implementations, for a configurable amount of cycles.

The Leftover, Spatial-Multitasking, Warped-Slicer (section 2.3) and MIAS thread block scheduling algorithms were implemented. They were found to only differ in the following three operations:

`update_scheduled_kernels()`: which decides which kernels from the list of launched kernels are going to be scheduled on the SMs. It puts them on the scheduled kernels list (`m_scheduled_kernels`). The default implementation passes all the launched kernels to the scheduled list.

`generate_configs()`: Generates the thread block configurations. It does not have a default implementation.

`on_done_profiling()`: It is used by algorithms that need profiling to decide what to do once the profiling is over. The default implementation does nothing.

Most algorithms only redefine one or two of these operations:

**Spatial multitasking:** It redefines the `generate_configs()` operation. It generates one configuration per kernel, where the corresponding position is set to the maximum number of thread blocks of that kernel, and the others are set to 0. This way, it dedicates a set of SMs to each kernel.

**Leftover scheduler:** It extends the spatial multitasking scheduler and redefines the `update_scheduled_kernels()` function to schedule only one kernel at a time in the order they were launched. Once there are no more thread blocks to issue from the current kernel, thread block scheduling entries are updated and thread blocks from the next kernel are issued to the leftover resources.

**MIAS:** This one overrides the `generate_configs()` function to generate all configurations for profiling and, once it is over, choose the best configuration. The `on_done_profiling()` function is used to make the decision and reschedule.

**Warped-Slicer:** This one overrides the same functions as MIAS, but with different procedures to implement its algorithm.

As can be seen, with this infrastructure, implementing different thread block scheduling algorithms requires only adding their logic by reimplementing some of the described functions, making implementing new algorithms quite easy. This is a contribution to the simulator that may benefit other researchers. Therefore, its incorporation to the upstream GPGPU-Sim git repository will be requested.

### 3.3.4 The phase to phase transition issue

Thread blocks cannot be pre-empted until they finish execution. Therefore, before profiling starts on an SM, the thread block scheduler has to wait until the SM is partitioned according to the desired configuration, i.e. the extra thread blocks from the first kernel finish and get substituted by thread blocks from the newly launched one. The same happens when transitioning from the profiling to post-profiling execution. These transitions are inevitable and add to the time that the GPU is not running the optimal configuration. They can last from a few cycles to millions, depending on the time it takes the thread blocks that have to be substituted to finish execution.

This is an implementation related issue and not part of the algorithm. If thread block pre-emption is used, the implementation will no longer have this overhead.

# Chapter 4

# Evaluation

In this chapter, the performance impact of MIAS (Memory Interference Aware Scheduler) is discussed. It is compared with sequential execution and Warped-Slicer. The variants of MIAS are also compared with each other. First, the methodology and resources used to perform the evaluation are covered. Then, the results are explained.

## 4.1 Methodology

This section explains the setup of the experiments to carry out the evaluation. It starts with the architecture of the baseline GPU. Then, the benchmarks and, afterwards, the metrics used for evaluation is discussed.

### 4.1.1 Baseline GPU configuration

The GPGPU-Sim[28] simulator has been configured to model an Nvidia GeForce RTX2060, which has a Turing GPU architecture[39]. It is used as the baseline and modified to implement the thread block scheduling algorithms, as explained in section 3.3. It has 30 SMs, each having 4 Greedy-Then-Oldest (GTO) warp schedulers. There are 1024 hardware threads in each SM organized into 32 warps. An SM can accommodate a maximum of 32 thread blocks, has 65536 registers, 64KB of shared memory and a 64KB, 512-block, fully-associative L1 cache. The GPU has a 3MB, 16-way associative L2 cache, shared by all SMs and distributed across memory channels. Both caches have blocks that are 128 bytes. There are 12 FR-FCFS Memory controllers that are connected to GDDR5 DRAM chips. It has 12GB of DRAM. Table 4.1 summarizes the configuration.

| | |
|---|---|
| SMs | 30, SIMD width=32, 1365MHz |
| Per-SM warp schedulers | 4 Greedy-Then-Oldest schedulers |
| Per-SM resources | max 1024 threads, max 32 thread blocks, 65536 Registers, 64KB Shared Memory |
| Per-SM L1 Data Cache | 64KB, 512 blocks, fully-associative, 256MSHR |
| L2 Cache | 3MB, block size=128B, 16-way |
| Memory Model | 12 MCs, FR-FCFS, 3500MHz, GDDR5, 12GB |

Table 4.1: The configuration of the baseline GPU.

### 4.1.2 Benchmarks

Most of GPGPU literature classify applications as *Compute intensive* or *Memory intensive* [17, 21]. These categories are defined as follows:

**Compute intensive** Their performance is not bounded by memory bandwidth, since they do not use much. Therefore, it increases with more TLP (Thread Level Parallelism).

**Memory intensive** Their performance is limited by memory bandwidth, and saturates very quickly as their TLP is increased. It can even start to degrade due to increased memory latencies and, possibly, L2 cache contention as well.

To be able to rigorously evaluate MIAS, the benchmark set has to be as diverse as possible. It has three memory intensive kernels and 4 compute intensive ones. They are from the CUDA SDK[15] and Rodinia[36] benchmarks. Next, the peculiarities of each benchmark are discussed:

**Breadth First Search[36] (bfs)** It is a graph exploration algorithm. These kinds of algorithms tend to be very Memory Intensive, since the main operation they perform is edge chasing. The implementation of the algorithm for GPGPUs does a lot of uncoalesced memory accesses. As illustrated in figure 4.1, its threads spend most of their time waiting for data from memory (RAW dependencies).

**Fast Walsh Transform[15] (fwt)** It is also a Memory Intensive kernel that does very little computations per data item. It, however, makes coalesced memory accesses.

**Black Scholes[15] (blk)** It is a Memory Intensive kernel, but it performs a considerable amount of computations per data item and makes coalesced memory accesses.

**Matrix Multiplication[15] (mm)** It is a Compute Intensive kernel. Its memory latency induced stalls are very reduced due to the use of shared memory. It, however, loses a substantial amount of cycles to compute stalls. It could probably be owing to the lack of diversity in compute instructions, which leads to using one of the pipelines a lot more than the others.

**N-body[15] (nbdy)** It is mostly a Compute Intensive kernel, but it has quite a bit of RAW stalls that are caused by memory latencies.

**Binomial Options[15] (bopt)** It is a Compute Intensive kernel that has little memory induced stalls. It loses a considerable amount of cycles due to thread synchronization (idle or control stall in figure 4.1).

**DXTC[15] (dxtc)** It is a very Compute Intensive texture compression algorithm. The implementation for GPGPU achieves high IPC values as it does not produce too much memory traffic.

Figure 4.1 depicts their stall behaviour. Each bar corresponds to one benchmark and, in its entirety, represents the percentage of cycles when a warp-scheduler stalls (cannot issue an instruction). Each one is further broken down according to the reason the stall was produced: **RAW stall** (Read After Write) are caused mostly by long memory latencies. **Idle or control stall** mean that either there are no valid instructions in the instruction buffer or the warps are waiting, probably due to synchronization. **Memory pipeline stall** and **Compute pipeline stall** are caused when instructions are issued to the corresponding pipeline faster than it can accept them.

### 4.1.3 Benchmarking infrastructure

The evaluation process encountered different kinds of issues. Next, these are discussed together with the solutions that were come up with.

GPGPU-Sim is compiled as a shared library that is dynamically linked to programs, substituting the CUDA runtime. However, GPU programs execute in different processes that share the code of the simulator library but not the mutable data. Multi-kernel simulation requires the kernels to be launched by the same process.

Some benchmarks don't allow to change workload size, as they are part of a bigger algorithm. Others, allow that but not directly because they read their data from a file like an image for instance. A way to precisely adjust the size of the workloads of the kernels was needed to control the time each kernel executes for.

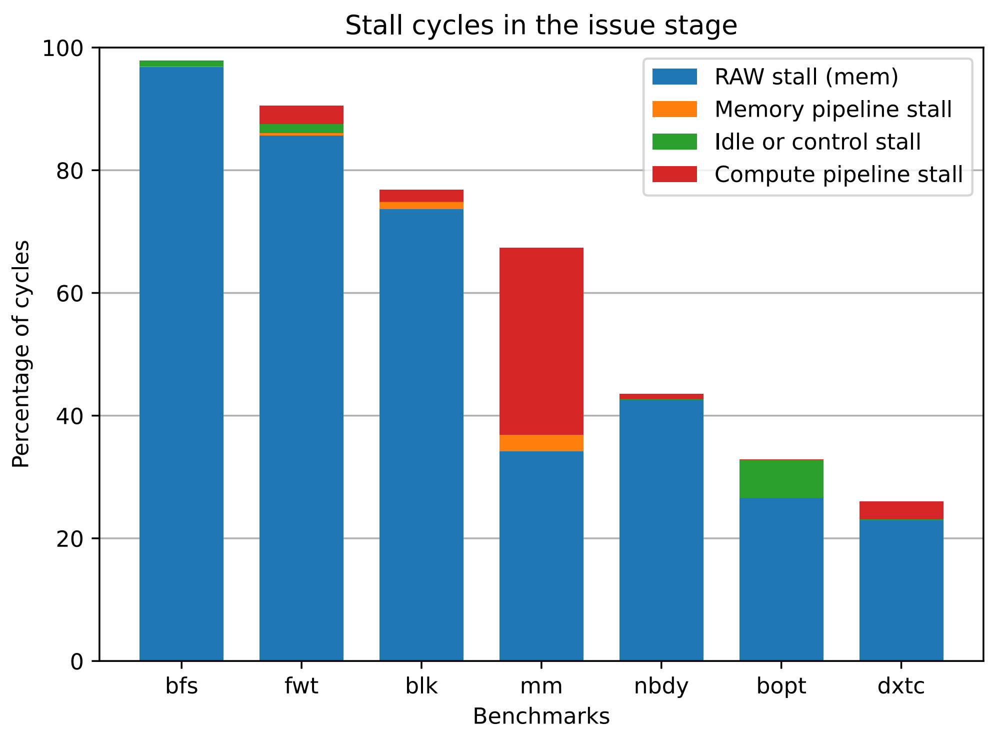


Figure 4.1: The composition of the percentage of stall cycles of the benchmarks.

Moreover, some of the selected benchmarks use features of CUDA that are not implemented in the simulator, like Cooperative Thread Groups[15]. Therefore, benchmarks need to be adapted by using low level synchronization primitives like `__synchthreads()` instead.

To overcome all of these issues, a program that launches the kernels in parallel with the specified number of thread blocks is developed. Each kernel is extracted from its original program and wrapped in a library with four functions:

**<kernel_name>_init()** takes as parameter the number of thread blocks, allocates and initializes the variables needed by the kernel.

**<kernel_name>_launch()** launches the kernels to the stream that it receives as parameter.

**<kernel_name>_check_results()** checks the results of the kernel to make sure they are correct.

**<kernel_name>_uninit()** frees up the memory that was allocated for kernel execution.

The main program stores pointers to these functions together with the name of the kernel they belong to in a data structure. It receives the names of the kernels to launch, with the number of thread blocks of each as command line arguments. It looks up the function pointers in the data structure and initializes all the kernels. Once they are all ready to go, it launches all of them to different streams. When they finish, it checks the results and frees up memory. This way, kernels are launched by the same process and, thus, they share the same simulated GPU.

The number of simulations that had to be made was quite big (sometimes hundreds of simulations), so scripts were written to automatically generate their configuration and run them. For that, Python was chosen over Bash because it provides more flexibility and ease to carry out the tasks of configuration generation. Although, executing commands is more verbose in Python.

Simulating computer architecture is a heavy process. It can take hours to simulate a millisecond of a GPU. Moreover, simulations of all the pairs of kernels with each alternative (MIAS, Warped-Slicer, etc.) have to be carried out, adding up to a total of 112 executions. A cluster, named Triton that belongs to the ATC group in the University of Cantabria, was used to perform the heavy simulations. They are launched to be executed in parallel by three, 20-core, 40-thread nodes. This reduced the simulation time, which would take several days in a normal computer, to less than a day.

During multi-kernel execution, it is more likely that one of the kernels finishes earlier than the other. That is why simulations are stopped before neither of the kernels finish execution. Only the part where both kernels are running is relevant to the evaluation. Moreover, Execution time is not used as the performance metric (section 4.1.4) and having predictable simulation time is important for running the simulations in Triton. Therefore, a fixed execution time of 5 million cycles is simulated in each experiment. Even if the multi-kernel execution continues beyond the 5 million cycles, the behaviour would not change until one of the kernels finish execution.

### 4.1.4 Metrics

To evaluate the impact of MIAS on performance and compare it to the other alternatives, the speedup metric is used. It measures the relative performance improvement of an alternative relative to another. It is calculated as folows:

$$S_{A,B} = \frac{IPC_A}{IPC_B} \tag{4.1}$$

Where $S_{A,B}$ is the speedup of alternative $A$ relative to $B$. $IPC_A$ and $IPC_B$ are, respectively, the IPC of $A$ and the IPC of $B$. The performance is usually considered as the inverse of the time it takes to finish execution. However, in the case of this dissertation, the IPC is used because MIAS aims to maximize it. Moreover, time cannot be used because the simulations are cut before the programs finish execution since they are run for a fixed amount of cycles. The next section (4.2) uses the speedup of variants of MIAS and Warped-Slicer relative to sequential execution.

For the performance of sequential execution of kernels, the average IPC of the kernels executed separately is used. No distinction between the pure sequential execution and Left-Over is made, because the transition from the execution of the first kernel to the second would last for less than a hundred thousand out of the five million simulated cycles. Those are the cycles where the GPU may have low occupancy in pure sequential execution, or kernel execution may overlap in Left-Over. The impact on performance would not be significant because of Amdahl's law[40]. This would simplify the evaluation process since fewer simulations are required, while results would not differ from actual Left over or pure sequential simulations.

Moreover, the percentage of cycles when the SM is stalled is used to study the behaviour of simultaneous kernel execution. The stalls have different causes, and they are distinguished as explained in sections 4.1.2.

## 4.2 Results

In this section, evaluation results will be discussed. First, the charts used are explained. Then, the achieved performance is discussed.

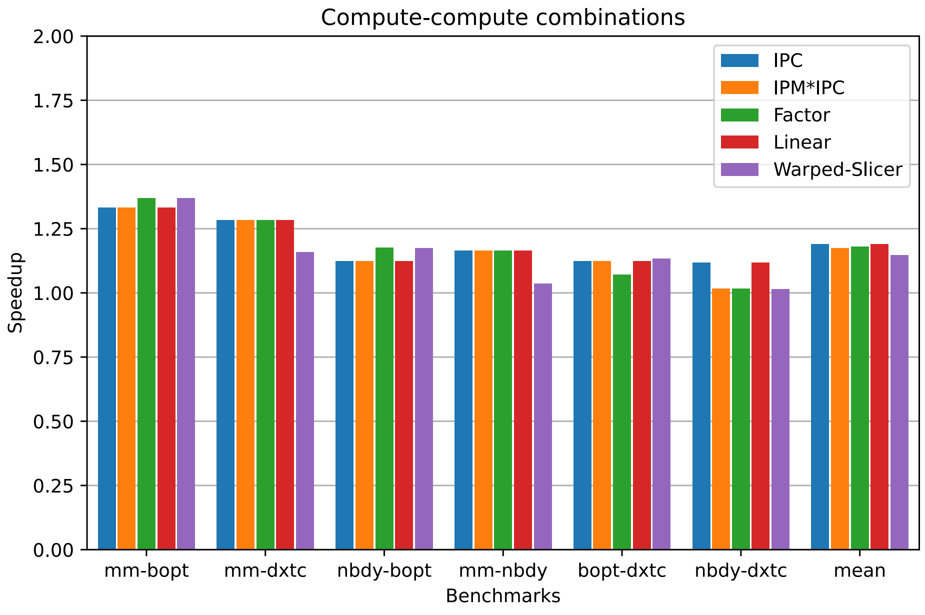
To visualize the results of the evaluation, the charts in figure 4.2 were made. The combinations of kernels were separated into three groups: Compute-compute combinations where both kernels are compute intensive (figure 4.2a), memory-memory combinations where both kernels are memory intensive (figure 4.2b) and memory-compute combinations where one of the kernels is memory intensive and the other is compute intensive (figure 4.2c). Every group has its own chart. All three charts have the same layout: The y-axis represents the speedup relative to the sequential execution, and the x-axis indicates the pair of benchmarks which the group of bars belong to. Each bar in a group indicates the speedup achieved by running the pair of kernels using the algorithm specified by the colour according to the legend. The first four bars each belong to the variant of MIAS indicated by the legend. The purple bar is dedicated to Warped-Slicer for comparison. The last group of bars are the mean speedup achieved by each algorithm for the type of combinations of kernels.

Figure 4.3 has four charts with the same layout as the one in figure 4.1. They are used to explain the results by having an insight into what happens in the SM when two kernels are executed together. Each chart has three bars which represent the percentage of stall cycles, broken down according to what caused them. The first two are for each kernel executed individually, while the one in the right depicts what happens when they are executed together.

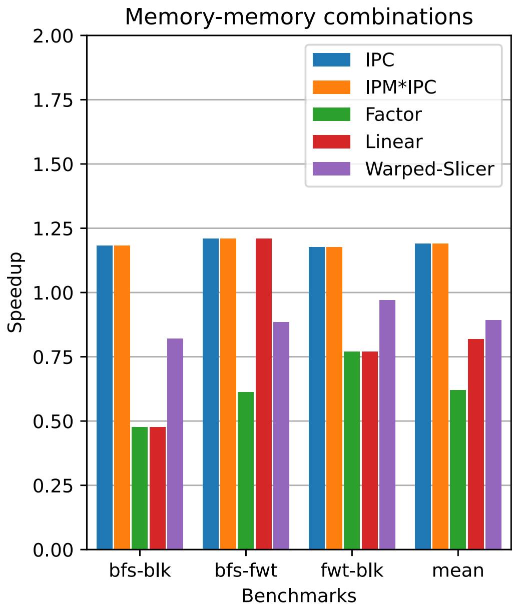
### 4.2.1 Compute-compute combinations

For compute-compute combinations, all five algorithms perform well, with the IPC and Linear variants of MIAS being in the lead with a mean speedup of 1.19. The strong influence of memory behaviour on the IPM*IPC and Factor variants make them choose incorrect thread block configurations because they overestimate the impact on performance. Because Warped-Slicer profiles the kernels separately, it does not take into account the interactions between them, thus, it achieved the lowest speedup out of the five algorithms.

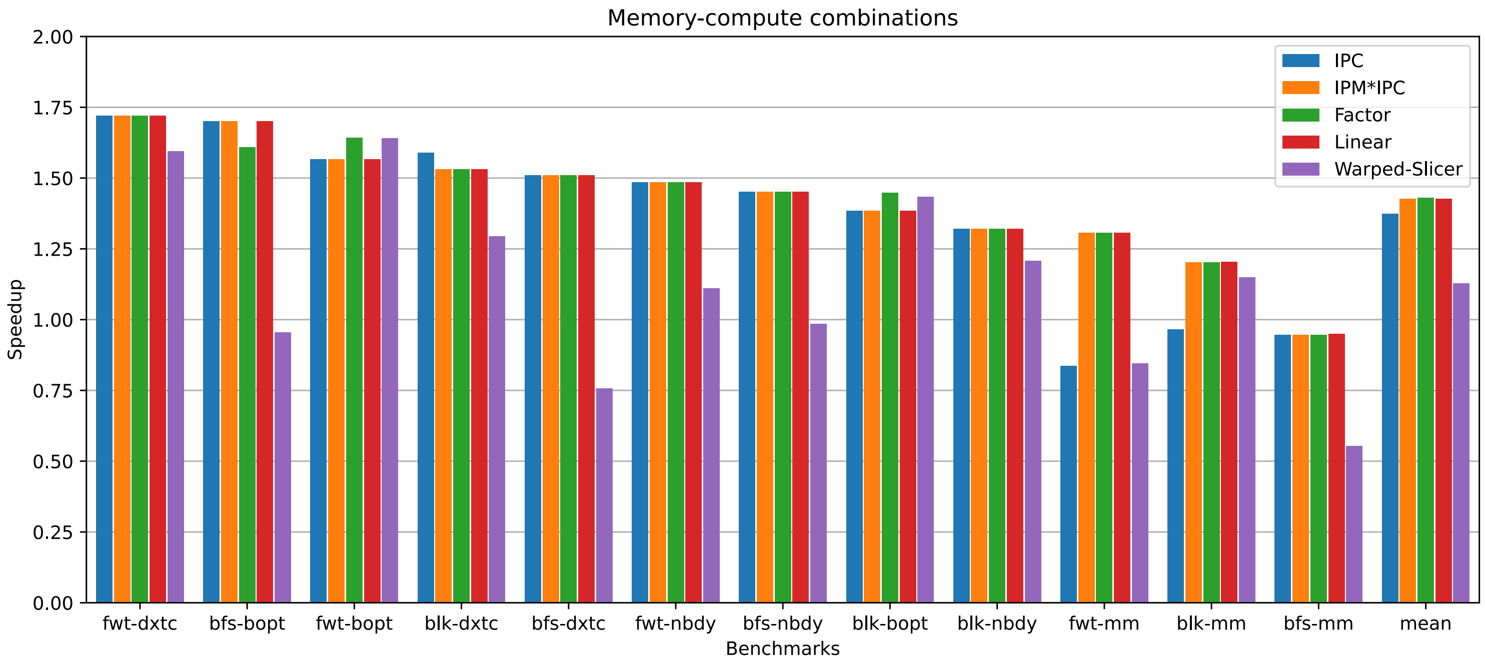
The performance gains for this group of combinations comes mainly from the diversification of the instructions being executed. This leads to better distribution of workload across all execution units in the SM, which in turn reduces the stalls caused by overutilization of one of the execution pipelines. Moreover, most compute kernels use thread synchronization to control shared memory access, which causes some warps to become temporarily unavailable for execution and, thus, the SM stays idle for quite a bit of time. Combining a kernel with this problem with others gives the SM work to do during synchronization bursts. Figure 4.3a illustrates these two phenomena. For example, Matrix Multiplication (mm) has a lot of compute pipeline stalls (first bar), while Binomial Options (bopt) has a lot of Idle or control stalls due to synchronization (second bar). When the two kernels are combined, both kinds of stalls are reduced because each kernel compensates for the other's deficiencies (third bar). The rest of compute-compute combinations have similar behaviour.



(a)



(b)



(c)

Figure 4.2: The speedup achieved by each algorithm relative to sequential execution of each pair of benchmarks. Per combination group mean values are plotted alongside.

### 4.2.2 Memory-memory combinations

When two memory intensive kernels are combined, there is probably not a significant difference in the stress put on memory by the different thread block configurations. This explains why the IPC and IPM*IPC variants of MIAS have the best performance, achieving a mean speedup of 1.19. Because the memory system is already very saturated, small changes in the stress put on it do not produce as much of an impact as predicted by the Factor and Linear variants. That is why they fail to make the right choice. Warped-Slicer significantly reduces the number of thread blocks that are simultaneously executing during profiling, which reduces the stress on memory compared to post-profiling execution. This leads to an inadequate choice.

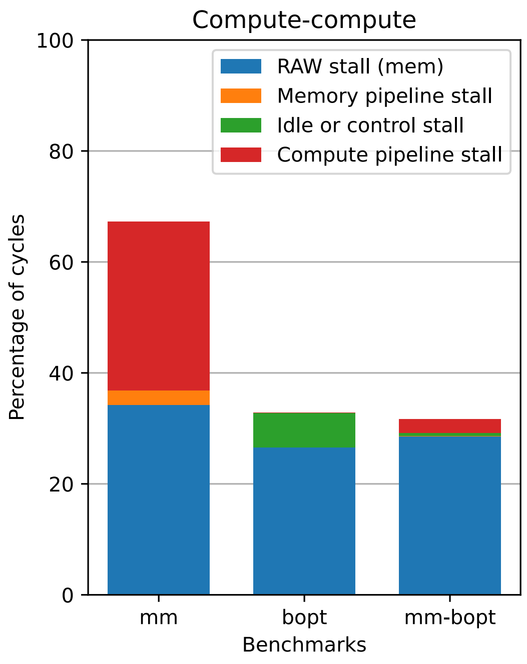
Figure 4.3b depicts what happens when two memory intensive kernels are combined. Most of the gains come from the diversifications of the small compute workload of the two kernels, like what happens in compute-compute combinations.

### 4.2.3 Memory-compute combinations

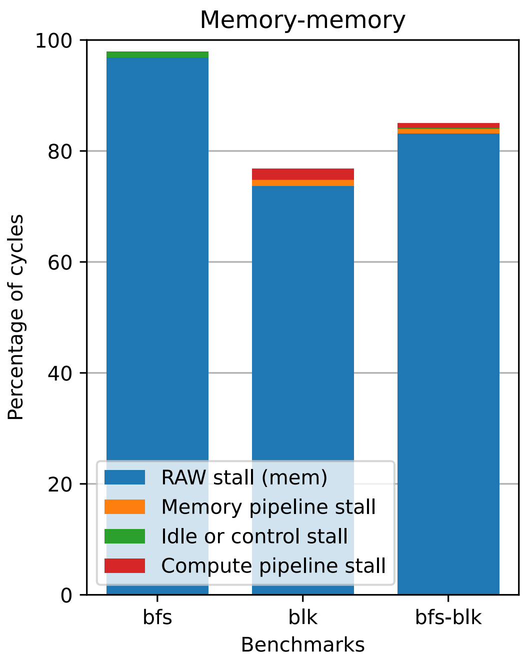
Combining Memory intensive kernels with compute intensive ones gives the best results, as can be seen in figure 4.2c. The IPM*IPC, Factor and Linear variants of MIAS are the best performing for this type of combinations, achieving a mean speedup of 1.43 with a 28% improvement over Warped-Slicer. Some combinations got a speedup of more than 1.7 which indicates a great performance gain. The more extreme the kernels get in being compute intensive or memory intensive, the better are the gains because they complement each other.

Predicting the impact of memory on the final IPC is very important for this type of combinations, because their thread block configurations have very different memory behaviours. This is specially important when combining compute intensive kernels that have pipeline stalls like Matrix Multiplication (mm) with memory intensive ones like Fast Walsh Transform (fwt) and Black Scholes (blk). That is what led to the IPC*IPM, Factor and Linear variants outperforming the IPC variant.

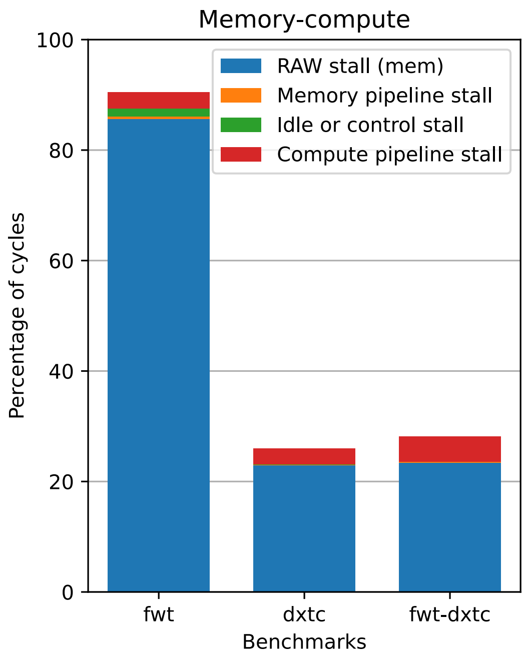
Figure 4.3c shows what happens when a memory intensive kernel is combined with a compute intensive one. The SM has more compute workload to cover the long memory latencies of the memory intensive kernel. The combined execution of fwt and dxtc gets a RAW stall percentage that is the same as when dxtc is executed alone. Both the SMs and memory system are kept busy, which increases overall throughput of the GPU.



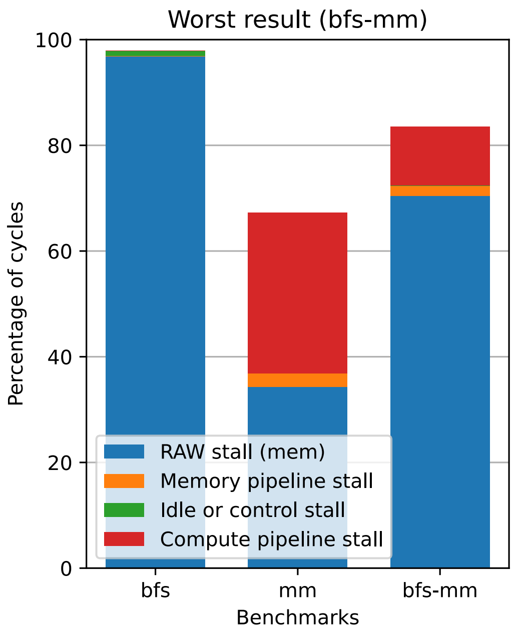
(a)



(b)



(c)



(d)

Figure 4.3: Comparisons between separate and joint executions of stall cycles composition of the best performing combination of each group of figure 4.2

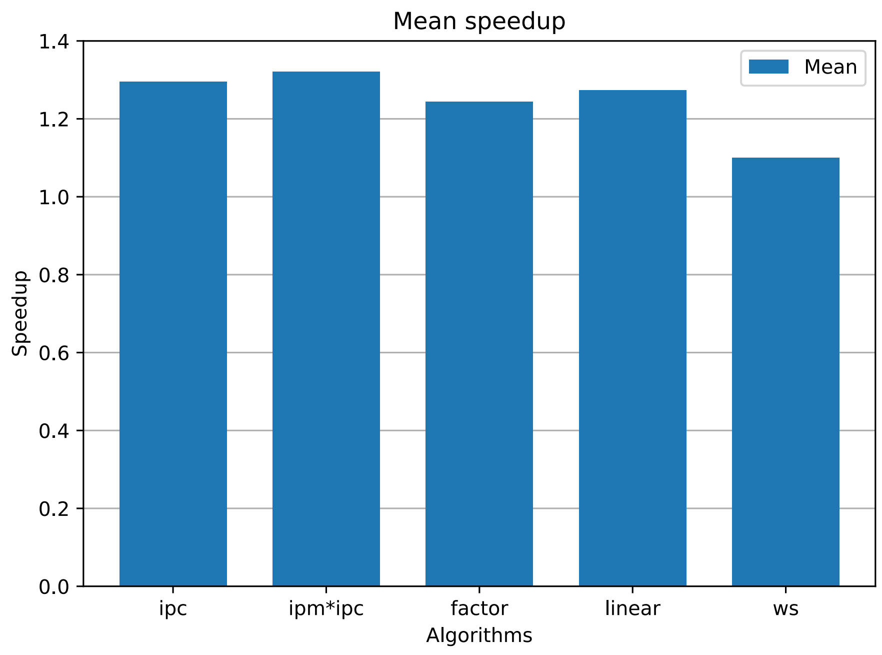


Figure 4.4: The global mean speedups of each algorithm

Warped-Slicer does not perform well for this type of combinations, getting speedups that are less than 1 in many of them. On one hand, profiling kernels separately does not allow it to take into account the interactions inside the SM. On the Other hand, it reduces the workload during profiling which lowers memory latencies, leading to significant differences between the pre and post profiling behaviours.

bfs-mm is the only combination in which successful multi-kernel execution is not achieved under any of the evaluated algorithms. Figure 4.3d shows that memory induced stalls were not reduced enough. BFS makes a lot of uncoalesced memory accesses, which increase memory latency even if its thread blocks are significantly reduced (only one out of the four run under MIAS). This keeps latencies high, which slows down mm too much. Empirical experimentation revealed that all possible thread block configurations (1-3, 2-2 and 3-1) have a speedup that is less than 1. In fact, MIAS chooses the best thread block configuration, which is 1-3 (1 thread block of bfs and 3 thread blocks of mm). This means that those two kernels are not good for multi-kernel execution, and should rather be executed sequentially or under Spatial Multitasking[16].

### 4.2.4 Overall performance

Figure 4.4 depicts the global mean speedup acheived by each algorithm (The variants of MIAS and Warpped-Slicer) relative to sequential execution. It is a bar chart whose y-axis represents the speedup, and the x-axis indicates the algorithm that corresponds to each bar. the bars indicate the global mean speedups.

The IPM*IPC variant of MIAS achieved a mean speedup of 1.32 which is the greatest. The IPC variant follows it at 1.3 and, then, the Linear variant at 1.27. The Factor variant achieved the lowest mean speedup out of MIAS variants at 1.24. Therefore, the IPM*IPC metric is the definitive metric for the decision step of MIAS. Warped-Slicer has a global mean speedup of 1.1.

These results show that MIAS has been quite a success. It improves overall performance by 32% over sequential execution and by 20% over Warped-Slicer. Moreover, it gets performance gains for all types of kernel combinations and excels when combining different types of kernels, achieving a speedup of up to 1.72 and a mean speedup of 1.42.

# Chapter 5

# Conclusions and future work

## 5.1 Conclusions

GPUs have become an integral part of most current computing systems, ranging from small ones, like mobile phones, to supercomputers. They accelerate computationally demanding tasks and make them cheaper and more accessible. They are still the dominant platform for deep learning, in which they enable more advancements as their performance and capacity grows.

GPGPUs make a good compromise between being specific and general purpose architectures, which makes them a compelling platform for many diverse applications. More data-parallel applications are expected to get ported over to GPUs, with different computational requirements. Moreover, GPUs are still growing at much faster rate than CPUs do. A lot more compute and memory resources are added with every generation. Therefore, multiprogramming is necessary for an overall efficient usage of these devices. Scheduling algorithms that can maximize performance by efficiently partitioning the resources of the GPUs are needed.

Left-over, the currently used algorithm for thread block scheduling for multiprogrammed workloads on GPUs leaves quite a margin for improvement. It is only meant to increase the occupancy of the GPU, but does not co-run the applications except maybe briefly. Warped-Slicer is currently the best research proposal that makes applications share resources at the finest-grain possible, by intra-SM sharing (sharing resources of the same Streaming Multiprocessor or core). It tries to make efficient resource partitioning. However, it profiles kernels separately, which does not allow for interference detection. Therefore, it still leaves room for improvement.

This dissertation proposes MIAS (Memory Interference Aware Scheduler). a thread block scheduling algorithm that improves upon the state of the art. It has a rigorous profiling strategy that creates the different resource partitioning configurations and conditions that the different kernels may encounter if they co-run. Unlike Warped-Slicer, the kernels are profiled jointly, which takes into account their interferences. Moreover, the different configurations are profiled in parallel in different SMs, significantly reducing the overhead of profiling. The impact of accesses to the memory system outside the SMs, which the profiled configurations share, is estimated and taken into account when deciding which one is the best configuration. The dissertation proposes four alternative metrics to decide the best configuration. For homogeneous kernel combinations (of only compute intensive kernels or only memory intensive ones), the IPC metric is enough to predict the best thread block configuration. However, when the combined kernels are of different types, the proposed memory estimation methods achieve better results. Because the factor and linear metrics do not perform well for combinations with only memory intensive kernels, the IPM*IPC metric is the one that achieves the best overall results. Thus, it is chosen as the metric for the final version of MIAS.

Evaluation results show that MIAS have been quite a success. It improves overall performance by 32% over the Left-over or the sequential approach (They have very similar results on big workloads), and by 20% over the Warped-Slicer approach, which is the state of the art. For homogeneous kernel combinations, MIAS improves performance by 18% over sequential execution. It performs much better for heterogeneous combinations, improving performance by a mean of 43%.

## 5.2 Future lines of work

This dissertation has achieved considerable advancements in the multiprogramming realm. Nevertheless, there are still some issues and lines of work to explore. They are discussed next:

- **Locality estimation for better decisions:** The methods used to decide which thread block configuration is the best, in this dissertation, do not take into account the locality behaviour, i.e. how close memory accesses are in both space (spatial locality) and time (temporal locality). An estimation of the locality of the workloads would improve the decision step of MIAS.

- **A kernel scheduler on top of the thread block scheduling level:** In current GPUs, kernels are issued to the hardware thread block scheduler in FIFO order, regardless of what type they are. However, from the results of this dissertation and other multiprogramming literature[18, 21], it is clear that executing different types of kernels concurrently yields greater performance. A kernel scheduler, that can determine the types of kernels and try to schedule complementary ones together, would considerably improve performance.

- **MIAS with a kernel-aware warp-scheduler:** In this dissertation, the default GTO scheduler was used. It is agnostic about the origin of the warps (to which kernel they belong). Putting together MIAS with a kernel-aware warp-scheduler might improve performance. The warp-scheduling algorithm might be a new one that takes into account what MIAS decides or an already proposed one[21, 22, 23]

- **MIAS with kernel-aware memory request scheduling algorithms:** This is similar to the previous line of work. It consists in using a kernel-aware memory request scheduling algorithm[30] together with MIAS. An ultimate implementation of a GPU with all four, previously mentioned, schedulers (kernel, thread block, warp, and memory request schedulers) supporting multiprogramming, would also be a good option to explore.

# Bibliography

[1] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055.

[2] Maria Angelica Davila Guzman et al. “Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL”. In: *J. Supercomput.* 75.3 (2019), pp. 1732–1746. DOI: 10.1007/s11227-019-02768-y. URL: https://doi.org/10.1007/s11227-019-02768-y.

[3] William J. Dally, Stephen W. Keckler, and David B. Kirk. “Evolution of the Graphics Processing Unit (GPU)”. In: *IEEE Micro* 41.6 (2021), pp. 42–51. DOI: 10.1109/MM.2021.3113475.

[4] Peter N. Glaskowsky. *NVIDIA’s Fermi: The First Complete GPU Computing Architecture*. Tech. rep. NVIDIA Corporation, 2009. URL: https://www.nvidia.com/content/pdf/fermi_white_papers/p.glaskowsky_nvidia%27s_fermi-the_first_complete_gpu_architecture.pdf.

[5] Pablo Toharia et al. “Scalable shot boundary detection”. In: *J. Supercomput.* 64.1 (2013), pp. 89–99. DOI: 10.1007/s11227-012-0784-8. URL: https://doi.org/10.1007/s11227-012-0784-8.

[6] Emilio Castillo et al. “Financial applications on multi-CPU and multi-GPU architectures”. In: *J. Supercomput.* 71.2 (2015), pp. 729–739. DOI: 10.1007/s11227-014-1316-5. URL: https://doi.org/10.1007/s11227-014-1316-5.

[7] Tor M. Aamodt, Wilson Wai Lun Fung, and Timothy G. Rogers. *General-Purpose Graphics Processor Architectures*. Morgan &amp; Claypool Publishers, 2018. ISBN: 1627059237.

[8] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321455363.

[9] Borja Pérez et al. “Sigmoid: An auto-tuned load balancing algorithm for heterogeneous systems”. In: *J. Parallel Distributed Comput.* 157 (2021), pp. 30–42. DOI: 10.1016/j.jpdc.2021.06.003. URL: https://doi.org/10.1016/j.jpdc.2021.06.003.

[10] Raúl Nozal, José Luis Bosque, and Ramón Beivide. “EngineCL: Usability and Performance in Heterogeneous Computing”. In: *Future Gener. Comput. Syst.* 107 (2020), pp. 522–537. DOI: 10.1016/j.future.2020.02.016. URL: https://doi.org/10.1016/j.future.2020.02.016.

[11] B. Pérez, J. L. Bosque, and R. Beivide. “Simplifying Programming and Load Balancing of Data Parallel Applications on Heterogeneous Systems”. In: *9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*. GPGPU ’16. ACM, 2016, pp. 42–51. ISBN: 978-1-4503-4195-0.

[12] Borja Pérez et al. “Auto-tuned OpenCL kernel co-execution in OmpSs for heterogeneous systems”. In: *J. Parallel Distributed Comput.* 125 (2019), pp. 45–57. DOI: 10.1016/j.jpdc.2018.11.001. URL: https://doi.org/10.1016/j.jpdc.2018.11.001.

[13] Nvidia. *NVIDIA A100 Tensor Core GPU Architecture*. Tech. rep. NVIDIA Corporation, 2020. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[14] Nvidia. *NVIDIA’s Next Generation CUDA Compute Architecture: Fermi*. Tech. rep. NVIDIA Corporation, 2009. URL: https://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.

[15] *NVIDIA CUDA Programming guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Online; accessed 12-April-2023]. 2023.

[16] Jacob T. Adriaens et al. “The case for GPGPU spatial multitasking”. In: *IEEE International Symposium on High-Performance Comp Architecture*. 2012, pp. 1–12. DOI: 10.1109/HPCA.2012.6168946.

[17] Zhenning Wang et al. “Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing”. In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016, pp. 358–369. DOI: `10.1109/HPCA.2016.7446078`.

[18] Qiumin Xu et al. “Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming”. In: *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA ’16. Seoul, Republic of Korea: IEEE Press, 2016, 230–242. ISBN: 9781467389471. DOI: `10.1109/ISCA.2016.29`. URL: `https://doi.org/10.1109/ISCA.2016.29`.

[19] Zhen Lin et al. “Coordinated CTA Combination and Bandwidth Partitioning for GPU Concurrent Kernel Execution”. In: *ACM Trans. Archit. Code Optim.* 16.3 (2019). ISSN: 1544-3566. DOI: `10.1145/3326124`. URL: `https://doi.org/10.1145/3326124`.

[20] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. “Dynamic Resource Management for Efficient Utilization of Multitasking GPUs”. In: *SIGPLAN Not.* 52.4 (2017), 527–540. ISSN: 0362-1340. DOI: `10.1145/3093336.3037707`. URL: `https://doi.org/10.1145/3093336.3037707`.

[21] Hongwen Dai et al. “Accelerate GPU Concurrent Kernel Execution by Mitigating Memory Pipeline Stalls”. In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2018, pp. 208–220. DOI: `10.1109/HPCA.2018.00027`.

[22] Haonan Wang et al. “Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management”. In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2018, pp. 247–258. DOI: `10.1109/HPCA.2018.00030`.

[23] Chen Zhao et al. “Fair and cache blocking aware warp scheduling for concurrent kernel execution on GPU”. In: *Future Generation Computer Systems* 112 (2020), pp. 1093–1105. ISSN: 0167-739X. DOI: `https://doi.org/10.1016/j.future.2020.05.023`. URL: `https://www.sciencedirect.com/science/article/pii/S0167739X19306545`.

[24] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 4st. Morgan Kaufmann, 2022. ISBN: 0123814723, 9780123814722.

[25] D. R. Kaeli et al. *Heterogeneous Computing with OpenCL 2.0*. 1st. Morgan Kaufmann Publishers Inc., 2015. ISBN: 0128014148, 9780128014141.

[26] Raúl Nozal and Jose Luis Bosque. "Exploiting co-execution with oneAPI: heterogeneity from a modern perspective". In: *European Conference on Parallel Processing*. 2021, pp. 501–516.

[27] Raúl Nozal and Jose Luis Bosque. "Straightforward Heterogeneous Computing with the oneAPI Coexecutor Runtime". In: *Electronics* 10.19 (2021), p. 2386.

[28] Mahmoud Khairy et al. "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 473–486. DOI: `10.1109/ISCA45697.2020.00047`.

[29] B. Pérez et al. "Energy efficiency of load balancing for data-parallel applications in heterogeneous systems". In: *The Journal of Supercomputing* 73.1 (2017), pp. 330–342. ISSN: 1573-0484.

[30] Adwait Jog et al. "Anatomy of GPU Memory System for Multi-Application Execution". In: *Proceedings of the 2015 International Symposium on Memory Systems*. MEMSYS '15. Washington DC, DC, USA: Association for Computing Machinery, 2015, 223–234. ISBN: 9781450336048. DOI: `10.1145/2818950.2818979`. URL: `https://doi.org/10.1145/2818950.2818979`.

[31] Adwait Jog et al. "Orchestrated Scheduling and Prefetching for GPGPUs". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. Tel-Aviv, Israel: Association for Computing Machinery, 2013, 332–343. ISBN: 9781450320795.

[32] Brett W. Coon et al. "Register file allocation". U.S. pat. 7634621. Dec. 15, 2009.

[33] John Erik Lindholm et al. "Simulating multiported memories using lower port count memories". U.S. pat. 7339592. Mar. 4, 2008.

[34] Samuel Liu et al. "Operand collector architecture". U.S. pat. 7339592. Mar. 4, 2008.

[35] D.P. Palomar and J.R. Fonollosa. "Practical algorithms for a family of waterfilling solutions". In: *IEEE Transactions on Signal Processing* 53.2 (2005), pp. 686–695. DOI: `10.1109/TSP.2004.840816`.

[36] Shuai Che et al. “Rodinia: A benchmark suite for heterogeneous computing”. In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 2009, pp. 44–54. DOI: `10.1109/IISWC.2009.5306797`.

[37] Mohsen Kiani and Amir Rajabzadeh. “Analyzing data locality in GPU kernels using memory footprint analysis”. In: *Simulation Modelling Practice and Theory* 91 (2019), pp. 102–122. ISSN: 1569-190X. DOI: `https://doi.org/10.1016/j.simpat.2018.12.003`. URL: `https://www.sciencedirect.com/science/article/pii/S1569190X18301849`.

[38] Chen Ding et al. “Performance Metrics and Models for Shared Cache”. In: *Journal of Computer Science and Technology* 29 (2014), pp. 692–712. ISSN: 1860-4749. DOI: `https://doi.org/10.1007/s11390-014-1460-7`.

[39] Nvidia. *Nvidia Turing GPU Architecture*. Tech. rep. NVIDIA Corporation, 2018. URL: `https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`.

[40] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, N.J., Apr. 18–20), AFIPS Press, Reston, Va., 1967, pp. 483–485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California”. In: *IEEE Solid-State Circuits Society Newsletter* 12.3 (2007), pp. 19–20. DOI: `10.1109/N-SSC.2007.4785615`.