Priority-Based Memory Access Scheduling for CPU-GPU Workloads

BY

FABIO GHIOZZI
Laurea, Politecnico di Torino, Turin, Italy, 2013

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2015

Chicago, Illinois

Defense Committee:

Zhichun Zhu, Chair and Advisor
Wenjing Rao
Gianpiero Cabodi, Politecnico di Torino

To my wonderful Family,

I wouldn't be here, writing the final lines of my Master Thesis, without your unimaginable support. Your creativity, your astonishing efforts and your perseverance have always been, and always will be, the source of my inspiration and the true cause of all my achievements.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my committee members for the opportunity to develop this project as my Master Thesis and for the provided guidance that made this research work possible.

I would like to thank Professor Zhichun Zhu for introducing me to this interesting topic, for the continuous support and the remarkable patience she demonstrated over all these months. Without her help this Master Thesis would not have been possible.

I sincerely thank Professor Gianpiero Cabodi for his support and his helpful suggestions for my Master Thesis, and for his invaluable teachings over these years spent at Politecnico di Torino.

My gratitude goes to Professor Wenjing Rao, for her interest and appreciated support to this Research work.

Finally, I would like to thank the whole UIC Electrical and Computer Engineering Department for their contribution and support in the development of this Master Thesis.

FG

TABLE OF CONTENTS

CHAPTER PAGE

1 INTRODUCTION 1

2 BACKGROUND AND FUNDAMENTALS 5
2.1 The Memory Hierarchy 6
2.1.1 The Main Memory 7
2.1.2 Main Memory Operations 9
2.1.3 Main Memory Timing Parameters 11
2.1.4 The Memory Controller 11
2.1.5 The Row Buffer Management 13
2.2 Memory Accesses in CPU Systems 16
2.2.1 The Memory Bus 17
2.2.2 Fairness 17
2.3 Memory Accesses in GPU Systems 19
2.3.1 Tiling 20
2.4 CPU-GPU Heterogeneous Systems 22
2.5 The Simulation Environment 26
2.5.1 MARSSx86 Simulator 27
2.5.2 DRAMSim2 27
2.5.3 DRAMSim2 Components 28
2.5.4 DRAMSim2 Modifications Overview 30

3 SCHEDULING ALGORITHMS 32
3.1 Overview of Previous Approaches 33
3.1.1 Priority Assignments According to Frames per Second 33
3.1.2 Priority Assignments According to Dynamic Behavior Analysis 34
3.2 The Reference Algorithm: Staged Memory Scheduling 36
3.2.1 The Visibility Problem 38
3.2.2 Open Problems 39
3.3 Staged Memory Scheduling Implementation 41

4 THE PRIORITY-BASED SCHEDULING ALGORITHM 44
4.0.1 Assumptions 45
4.1 Priority-Based Algorithm: Architectural Design 48
4.1.1 The Queuing System 48
4.1.2 The Relocation Operation 50
4.1.3 The Priority Assignment Policy 52
4.1.4 The Reordering Management 54

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

4.1.5 The Scheduling Algorithm 55
4.2 Priority Queues Algorithm: Implementation 59
4.2.1 Implementation in the DRAMSim2 Environment 59

5 SIMULATION RESULTS AND ANALYSIS 63
5.1 Metrics 64
5.2 Test Cases 68
5.3 Two-Core Machine Results 70
5.4 Four-Core Machine Results 73
5.4.1 One-Channel Results 74
5.4.2 Two-Channel Results 77
5.5 Result Summary 80

6 CONCLUSIONS AND FUTURE WORK 82

CITED LITERATURE 84

VITA 86

LIST OF TABLES

TABLE **PAGE**

I EXAMPLE OF CACHE FEATURES THROUGH DIFFERENT LEVELS. 8

II TIMING PARAMETERS IN DRAM MEMORY ACCESSES. 12

III TEST CASES PERFORMED ON THE TWO CORES MACHINE CONFIGURATION 69

IV TEST CASES PERFORMED ON THE FOUR CORES MACHINE CONFIGURATION 70

LIST OF FIGURES

FIGURE **PAGE**

1 CPU-GPU heterogeneous architectures 2

2 The memory hierarchy 7

3 Memory components scheme 9

4 Transistor level DRAM architecture 10

5 The memory scheme 14

6 Tiling-based rendering 20

7 MARSSx86 architecture 26

8 DRAMSim2 classes and relationships 28

9 SMS batch creation scheme 41

10 Queuing system in the priority based scheduling algorithm 49

11 The relocation operation 51

12 Queues precedence order in the priority-based scheduling algorithm. . . 55

13 IPC, two cores machine configuration 71

14 System and CPU speedup, two cores machine configuration 73

15 IPC, four cores machine configuration, one channel 74

16 System and CPU speedup, four cores machine configuration, one channel 75

17 GPU respect of deadline, four cores machine configuration, one channel 76

18 IPC, four cores machine configuration, two channels 78

19 System and CPU speedup, four cores machine configuration, two channels 79

LIST OF FIGURES (continued)

FIGURE **PAGE**

20 GPU respect of deadline, four cores machine configuration, two channels 80

LIST OF ABBREVIATIONS

DRAM	Dynamic Random Access Memory
FPS	Frames Per Second
SMS	Staged Memory Scheduling
PB	Priority-Based
EP	Expected Progress
GPUT	GPU Timeout
QoS	Quality of Service
IPC	Instructions per Cycle
CPUIPC	CPU Instructions per Cycle
GPUIPC	GPU Instructions per Cycle
WS	Weighted Speedup
NIR	Number of Instructions Ratio

SUMMARY

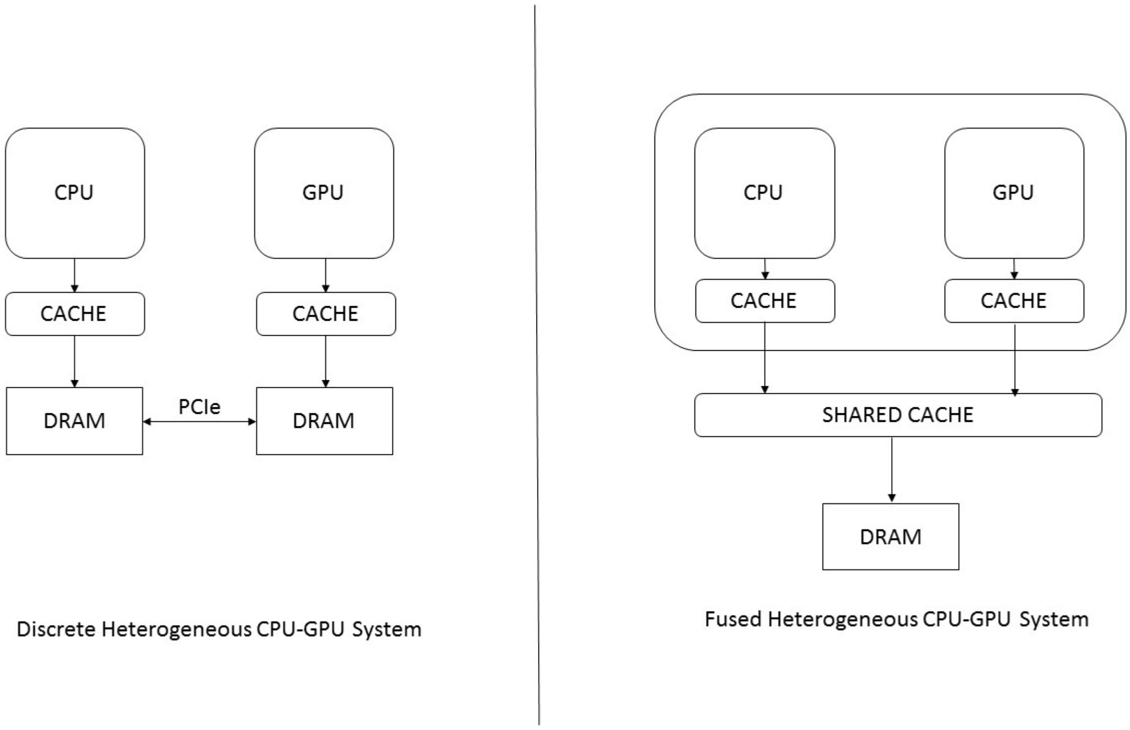
This thesis work is developed to design and implement a brand new Memory Access Scheduling Algorithm for CPU-GPU heterogeneous architectures. The ultimate goal is to introduce a new strategy that would improve the overall performance in real world scenarios, exploiting diverse memory patterns. Due to the complexity of having two modules executing very different workloads, many considerations have to be taken into account when dealing with these systems. For this reason, a considerable part of this work introduces many concepts, previous researches and simulation tests to provide all the necessary knowledge to understand the fundamentals behind the concept of the Priority-Based Scheduling Algorithm. The main idea behind this new strategy is to develop an Algorithm that improves the performance of the whole system in fixed conditions defined by previous works and architectural analysis‘. From this basis, the scheduling mechanism has been built developing dedicated architectural features to support a logic that would reflect the environment of interest. Introducing Priority-Based queuing structures and implementing operations able to provide Fairness and different latency values according to the issuing modules, it was possible to achieve a logic able to improve performance in an environment defined by the previously assumed conditions. The design of this new strategy is presented and described starting from a high level architectural point of view, to the implementation layer. Finally, the simulation results are presented and discussed, highlighting the advantages and trade-offs of this new scheduling algorithm.

CHAPTER 1

INTRODUCTION

In the latest years, the Computer Architectures Research Area has been populated by many different concepts and commercial products based on CPU and GPU heterogeneous systems. These architectures are characterized by the fact that both modules work together to improve the overall performance, according to the running applications, and are already diffused in commercial environments. These systems can be split in two main categories: Discrete and Fused architectures (Figure 1).

In the first case, the two processing units are developed on two different physical modules and provided with private and dedicated memory hierarchies. In this first scenario, shared data is processed using particular protocols that control the communication between the two DRAM memories and, from a lower level point of view, the architecture is made of two independent but interconnected memory systems. Unfortunately, in these environments data has to be sent from the CPU module to the GPU, processed and sent back, and the whole operation introduces a considerable amount of overhead on memory instructions and delay. Modern approaches to this problem brought to the development of Fused Architectures, where both the CPU and GPU are placed on the same die. Fused Architectures have already been developed and distributed on the market, demonstrating good achievements in terms of timing performance and power consumption. The concept is to have the CPU and GPU modules on the same die and served by one single off-chip DRAM Memory Hierarchy and Controller. The major benefit of



Discrete Heterogeneous CPU-GPU System

Fused Heterogeneous CPU-GPU System

Figure 1: CPU-GPU heterogeneous architectures

this architecture is the reduction of the overhead generated by the communication between the memory modules dedicated to the two distinct computation units, as designed for the Discrete counterpart. Developing an architecture based on one single main memory hierarchy, theoretically, would completely remove the timing overheads. In a real world scenario, however, while the improvements still justify the research and development of Fused Architectures, new challenges have surfaced on a variety of different aspects. While many of these issues like frequencies management and low-level microelectronics design are attributable to the Electrical

Research area, major attention has been focused on the Scheduling of the instructions coming from the two distinct modules. Usually, when dealing with these two different computation units, CPU and GPU, several new parameters have to be taken into account. Different memory accesses patterns, applications, requirements and constrains are introduced and have to be considered not only to reduce the overhead, but also to serve the particular purpose of each computation unit. Additionally, in certain cases, those differences can be exploited to improve the performance of the whole system. In summary, the introduction of a complex architecture composed by units with different behaviors sharing a single resource has to be dealt with extreme attention of each module characteristics and requires a detailed strategy to process all the requests coming from the whole heterogeneous computation environment. Due to this large amount of different constraint and application-related peculiarities, many different approaches have been taken under consideration to design a scheduling algorithm for both Fused and Discrete Heterogeneous Architectures. In many cases, strategies have been developed to evaluate a dynamic algorithm according to the priority degree desired for one of the two modules. The goal of this Research is to introduce a brand new Priority-Based Memory Access Scheduling Algorithm for CPU-GPU Workloads that takes into account both the different purposes of the two modules and provide an efficient scheme, based on assumptions built from their expected behavior, to evaluate and assign coherent priority degrees to their instruction flows. The whole research stands on a set of performance analysis' and behavioral considerations discussed and presented on different papers published in Literature. For this reason, the goal of the second chapter of this work is to collect and explain, from an architectural point of view, the major characteristics and features of all the modules of the system under study. Subsequently, the same modules are analyzed considering a CPU-GPU heterogeneous system. The aim of this part is to describe in detail how all the features of each module behave in a shared environment and how it is possible to control all the requirements and application-specific patterns in order to optimize the considered metrics. These considerations are then taken as assumptions in the third and fourth chapters, where two algorithms are discussed and explained. The first one, published in 2012, is the Staged Memory Scheduling: Achieving High Performance and Scalability by Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh and Onur Mutlu (1), presented here as the reference algorithm that will be used for the result comparisons. The second algorithm is the Priority-Based Memory Access Scheduling for CPU-GPU Workloads, a brand new algorithm built from the assumptions described in chapter II. The two algorithms' objective is similar: to exploit the CPU and GPU features in order to improve the overall behavior of the considered system. However, while the reference algorithm's high level scheduling across modules is developed over probability considerations independent from the application loaded in the computation cores, the Priority-Based algorithm picks up instructions from the different modules according to the status of the system and the running applications. Our is to provide a scheduling algorithm able to improve performances according to analysis on specific but common workloads. Finally, in Chapter 5, the two algorithm are compared through a series of simulation results analysis' where the different advantages and disadvantages of the two approaches are introduced and discussed. Finally, we summarize our work in Chapter 6.

CHAPTER 2

BACKGROUND AND FUNDAMENTALS

The goal of the following sections is to provide a background of all the concepts and architectures useful to understand the design features presented in the following chapter, where the studied Scheduling Algorithms will be discussed in detail. In this chapter all the actors that will play a part in the explanation of the algorithms will be described, without introducing details on the whole computer architectures topic that would be not useful for the purpose of this work.

In the first section of this chapter, the main features of the memory system are discussed in order to present the main modules, operations and metrics that will have a role in the design and the implementation of the algorithms. Subsequently, an overview on the dynamics and mechanisms of the memory accesses and the main memory operations requested by the CPU and GPU modules are introduced to state the behaviors that will be taken into account in the considered Scheduling Algorithms. Finally, the simulation environment is represented to provide a clear understanding on how the tests and the simulations had been performed and how the simulation environment had been modified to output the information used for the comparisons.

2.1 The Memory Hierarchy

The memory system considered in our machine can be schematized as a hierarchy of different storage components having different purposes and features. This is a well-known and powerful concept, essential in the computing architectures definitions (Figure 2).

This hierarchical view of the memory system finds its origin from the different types of memories present on the market that can be used, applying different strategies, to improve performances and maintain a boundary on costs. The main idea of Memory Hierarchy can be conceptualized using a pyramidal scheme of the system where the height of each layer represents the speed of the memory, the width the capacity. At the top of the pyramid it is possible to find the CPU registers, very small and fast memories that can be accessed by the CPU in a negligible amount of time to store data during execution. CPU registers can be used, for example, during the execution of a program to store the value of operands, or the offset of a jump instruction. The second layer is occupied by cache memories, small storage devices, faster than the main memory, loaded with data accessed during execution by the CPU and that are likely to be accessed or manipulated in the future. They‘re introduced as a faster extension of the main memory, and provide a fast access to data that otherwise would require a slow main memory request at each need. Usually cache memory is represented in its own sub-hierarchy made of multiple levels, where each level has different features in terms of performance and size (Table I). Every level can be made of multiple chips, each one private to its core, or shared where more than one core has access to the stored data. Access policies and coherence protocols have to be set accordingly to the visibility of the cache. Data stored in the cache memories is loaded

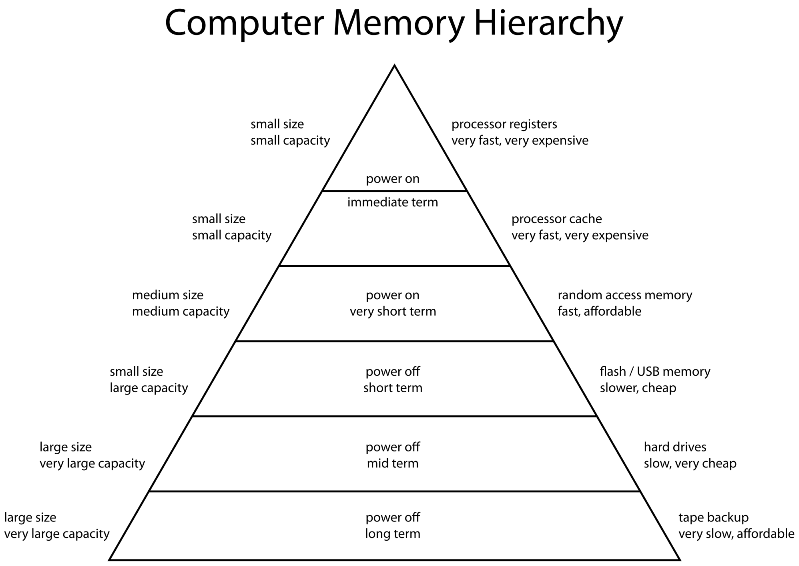


Figure 2: The memory hierarchy

from the Main Memory according to policies aiming to maximize the hit ratio of the memory requests operations, an important metric that estimates the number of accesses performed on a fast cache with respect to the number of accesses on the slow Main Memory.

2.1.1 The Main Memory

The Main Memory is the physical volatile layer of memory responsible to store all the data necessary for the execution of the processes running on the computation cores. It can store parts of the programs extracted from the secondary memory and make it accessible, through the previously described layers, to the CPU, propagating memory commands. Usually this layer of the memory hierarchy is implemented with dynamic random access memories (DRAM), where each bit of information is stored using the charge of a single capacitor. Due to the scalability

TABLE I: EXAMPLE OF CACHE FEATURES THROUGH DIFFERENT LEVELS.

Level	Description	Size	Speed
Level 1	Instruction and Data Cache	128 Kib	700 GiB/sec
Level 2 (Also Shared)	Instruction and Data Cache	1 MiB	200 GiB/sec
Level 3	Shared Cache	6 MiB	100 GiB/sec
Level 4	Shared Cache	128 MiB	40 GiB/sec

of microelectronics technology, the number of transistors packed in a small area can lead to a considerable amount of storage (GB). The main memory is organized and described in a sub-hierarchy where each layer is an aggregate of lower level components (Figure 3).

The highest level components are the channels, and each one of them includes a set of ranks which are collections of different memory chips served by one unique chip select. Down in the hierarchy, each rank is made of different banks (usually 8) split in rows (64K). Each memory cell is addressed by the rank, bank, row and column (2). The DRAM is organized as a bidimensional array of transistors placed in a matrix of variable size (Figure 4). The Read operation is used to obtain a cell of data addressed by the row and column indexes and is performed by reading and rewriting the whole row of the addressed cell because of the loss of charge that requires the data to be reloaded after the read. After that, the required bit is then selected and provided. The Write operation is similar: the bit lines are charged, the selected transistors closed, and the charge stored. Due to the implementation of the stored bit as a capacitor, the charge is not kept for a high amount of time due to the leak of charges. For this reason, a Refresh operation is

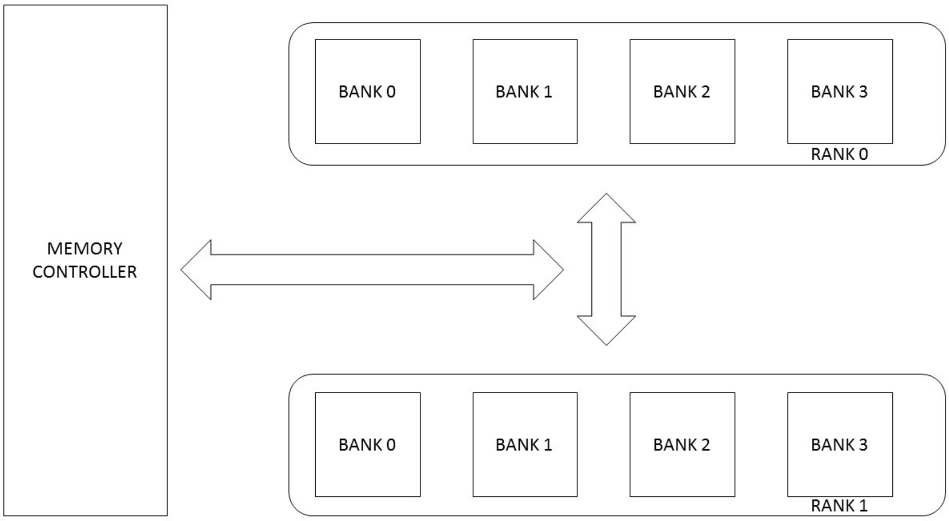


Figure 3: Memory components scheme

required and every period of time in terms of milliseconds, a dummy Read and Write operation is required to restore the charge (3).

2.1.2 Main Memory Operations

The following is a list of the main operations performed in the memory system and implemented in the memory controller design.

Read

1. At the beginning of the Read scenario, the addressing signals, RAS* and CAS* can be considered active, and the bit lines are precharged to Vdd/2.
2. The desired Row Address signal is activated and RAS* goes to GND. The bit lines are disconnected from the Vdd/2 voltage and are floating, maintaining the charge.

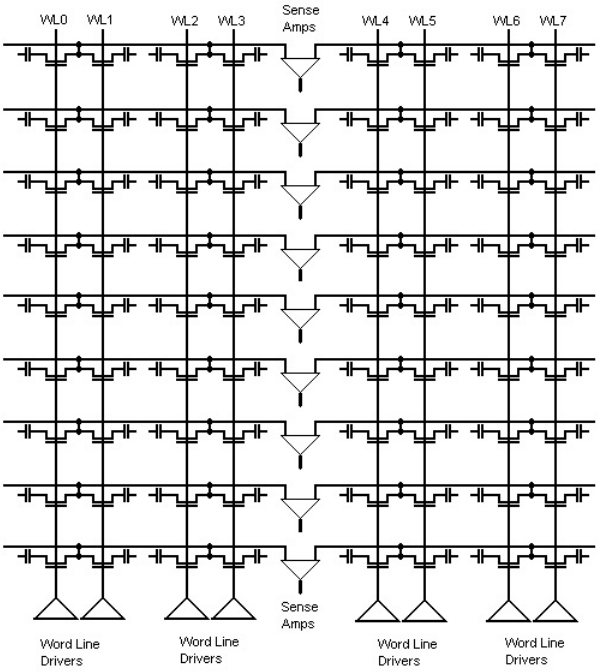


Figure 4: Transistor level DRAM architecture

3. The signal address is applied to the row driver and the stored signal is propagated to the sense amplifier.
4. The differential voltage between the stored signal and the reference is amplified and the single bits are brought to VDD or GND.
5. CAS is brought to GND and the column data is connected to the sense amplifier.
6. All data of the row is brought, through sense-amplifying, to GND or VDD.

Write

1. At the beginning of the Write scenario, the addressing signals, RAS* and CAS* can be considered active, and the bit lines are precharged to Vdd/2.
2. The desired Row Address signal is activated and RAS* goes to GND. The bit lines are disconnected from the Vdd/2 voltage and are floating, maintaining the charge.
3. Datum is applied to the line.
4. CAS* brought to GND and the write driver drives the sample amplifier.
5. Both RAS* and CAS* are brought to the active state again.

Refresh

The Refresh operation is performed as a dummy Read, and each row is refreshed at a time. It is an overhead operation because cannot be performed at the same time of a Read and a Write operation.

2.1.3 Main Memory Timing Parameters

The following Table (TABLE II) represents a list of the main timing parameters used to evaluate the performances of a memory system (4).

2.1.4 The Memory Controller

The memory controller is the module in charge of the management and addressing of the memory requests coming from the computation units in the system. It can be both integrated into the Computation Unit (CPU, GPU) or off-chip and shared among the different sources of the commands. The memory controller manages the configurations signals directed to the Main Memory for the Read and Write operations. Additionally, it periodically triggers the Refresh

TABLE II: TIMING PARAMETERS IN DRAM MEMORY ACCESSES.

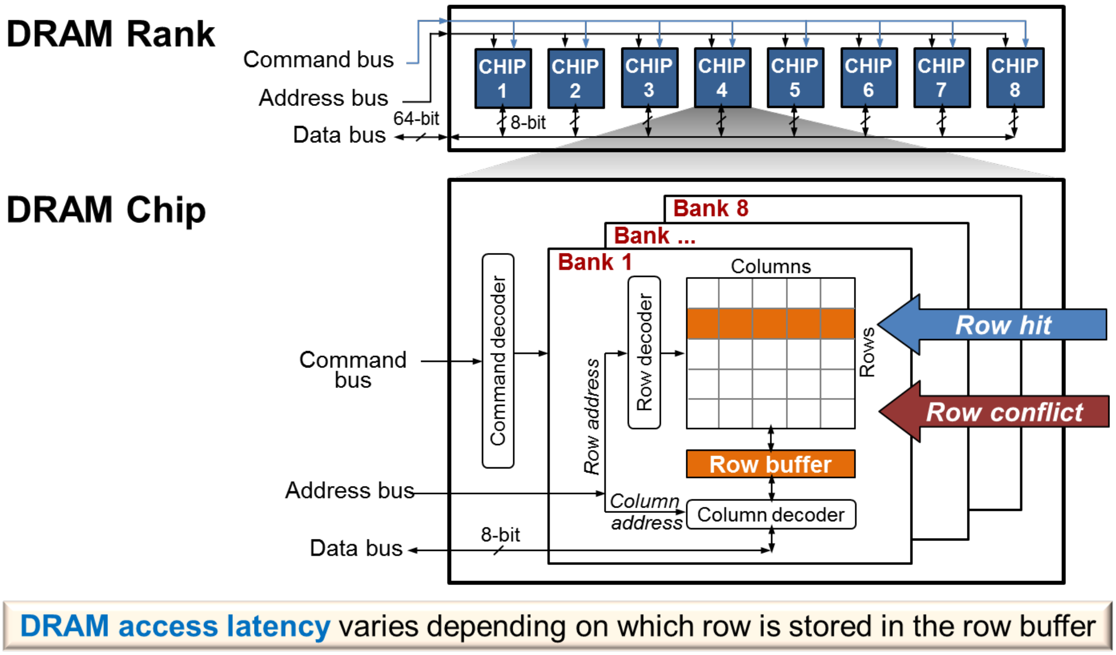
Name	Description
t_RP	Time for Row Precharge: time required to charge the sense amplifiers and the bank activation.
t_RCD	Time between Ras and Cas activation: number of clock cycles between the opening of the row and the column activation.
t_RAS	RAS Active Strobe: time to activate a row of a bank.
t_RC	Time for Row Cycling: time between two consecutive accesses to the same bank. Defined as the sum of t_RAS + t_RP
t_RRD	Time between successive activations on different banks.
t_CLK	Clock Cycle Time
t_WTR	Time for Write To Read: Minimum time between end of a WRITE and a READ command.
t_WR	time for Write to Row: time interval between end of WRITE and PRECHARGE command.

commands to perform the data recovery. Furthermore one of the main tasks of the module is to convert the high level addresses it receives from the instructions executed by the CPU into the proper set of control signals required by the Main Memory to perform its operations. This conversion has to take into account the mapping from a physical encoded address to a set of selection signals to the correct rank, bank, row and column of the data we want to access. Once the address has been converted and the control signals has been injected into the main memory, the memory controller receives the data through a Data Bus. The Data Bus Width can be variable and spans from 8 bits implemented in early systems to 512 bits to serve video cards and high bandwidth computation units. The Memory Controller can also consists of error detection logic that, through matching operations, can acknowledge the presence of an error and restart the memory request operation in order to submit the correct information. Due to the increasing of complexity in terms of parallel programming, and the necessity to have a shared memory across multiple modules and multiple threads, Memory Controllers have become more and more complex to support new challenges on Fairness and Data Coherence.

Furthermore, many new ways to optimize the memory throughput have been developed to improve the concurrency exploiting both software and architectural solutions. One example is the impressive amount of research performed in the topic of Scheduling Algorithms, but the set of optimization spans among almost all the layers of design, from low-level Software to Microelectronics. The development of more complex architectures is led by numerous different improvements in the design of DRAM memories. As an example, the implementation of Double Data Rate memories, which allows the transmission of data both in the rising and falling edge of a clock cycle, requests additional logic to the memory controllers, which can be also dependent from the number of channels of the main memory since they have to deal with all the channels of the whole memory system.

2.1.5 The Row Buffer Management

In order to improve the timing performances during accesses to the memory, not only traditional caches have been introduced in the memory systems, but also some mechanisms to exploit the temporal and spatial locality of multiple accesses. One of the most common is the row buffer, a fast-access memory that stores the last accessed rows to improve the timing performances, due to the fact that in terms of probability it is common that during normal execution the CPU performs more than one access to the same row (5) (Figure 5). For this reason the larger the hit ratio of the row buffer, the better the timing performances. This subject has been at the center of the optimization mechanisms for the memory system architectures. The row buffer misses happen when a sequence of instructions is trying to access different pages in the same bank. As a response to these architectural choices, to improve different



DRAM access latency varies depending on which row is stored in the row buffer

Figure 5: The memory scheme

scenarios of the memory utilization, several different policies have been developed to support the management of the row buffer, which can be chosen according to the different applications and needs.

1. Open Page Policy

   This strategy is used to exploit locality and it is based on the idea of triggering one Precharge operation only when the row buffer access is missing. On row buffer hits, it presents a minimal latency because no overhead operation is performed, but in a scenario where a high number of misses are happening, it can introduce too much overhead. From the power consumption point of view this policy is the most expansive because the sense amplifiers are kept open for the whole time.

2. Close Page Policy

   It starts the Precharge operation, periodically, at every memory access. Differently from the open page case, here the performances are improved in scenarios where many buffer misses are predicted, but since the Precharge operation is performed more frequently, this policy can introduce too much overhead in scenarios where the hit rate is high. From the power consumption point of view, this is estimated to be 3x lower than in the Open Page Policy (6).

3. Adaptive Page Policy

   Since the two previous strategies' effectiveness depend on the row buffer hit ratio, a dynamical policy that switches from one to the other has been implemented. The central idea is to have a trade-off between the advantages and disadvantages of both the strategies. It works comparing the current hit ratio of the memory session with a threshold value and, according to this check the policy is kept or swapped with the inactive one.

2.2 Memory Accesses in CPU Systems

In the latest years, the design and the implementation of efficient memory systems and memory management strategies in CPU-based architectures has been at the center of the Architectural Research. The introduction and then the massive development of multicore technologies has been one of the main advancements in the field of Processor's Design. While the benefits and the new opportunities coming from these devices has brought a whole set of new challenges in the development of multithreaded applications, new problems, especially from the memory management design, have surfaced. The development of computation systems where different cores have access to the same physical memory, and are able to work on the same data set, has introduced new challenges in terms of coherency and data protection. Additionally, the memory bus shared among all the cores can be the bottleneck in an execution scenario due to the insufficient bandwidth it can provide with respect of the required data exchange generated by the cores (7). Finally, having a single resource shared among different sources could introduce problems in terms of fairness due to the fact that, according to different needs and different workloads, the distribution of the memory accesses could be trickier then expected. In the following sections, some of the main problems of memory management in multicore environment are introduced and discussed along with possible solutions. Many of the issues here presented, even if introduced in a CPU-only environment, can play also an important role in a CPU-GPU environment, where there are still multiple modules running for the control of a shared resource (Figure 6).

2.2.1 The Memory Bus

The Memory Bus represents, in modern architectures, one of the most well-known delay factors and can be a bottleneck in the whole CPUs memory system. The main problem is technological: the bus speeds, in fact, do not increase at the same rate as the CPU speeds. Additionally, the more CPUs are implemented in the system, the more stressed the bus is and can influence the overall performance. Cache Snooping itself, a Cache Coherency method described in the previous sections, can increment the utilization of the bus and decrease the performances. In order to reduce the effect of this boundary on the performances, many approaches have been introduced to optimize the bus usage. Again, different strategies have been designed on different layers in many research works and have been tested extensively to determine the best trade-offs and results (10).

2.2.2 Fairness

In architectures composed of several computation cores, it is important to design and build a scheme that grants several Fairness controls over the overall flow of memory requests coming from the different sources. It is necessary, for example, that the time in which the bus is dedicated to one source does not inject excessive stall time in the other cores' execution. For this reason, many research works have been published to control how a memory resource is used and addressed by all the modules, suggesting many Scheduling Algorithms able to improve the performances in different scenarios. The main goal would be to provide a Scheduling Algorithm that would assure to each thread the opportunity to perform the same progress over their own execution flow. While previous work on Scheduling Algorithms was focused on the timing performances and power consumption, the pursuit of a scheme able to optimize this metric has been subject of research over the latest years. In many research works on the topic (11) (12), the framework is similar: assigning different priority degrees at each thread based on application-aware considerations at the beginning of the executing scenario (static approach), or at run-time according to traffic analysis performed on the fly. Several concepts used to manage the fairness across different CPU cores have been introduced and implemented also in CPU-GPU architectures, in environment where the applications running on each module are not considered.

2.3 Memory Accesses in GPU Systems

In recent years, the architectures of GPGPUs have been changed dramatically in order to improve the overall system performance according to the new architectural concept risen and developed for the CPU. The general trend is a change from a very specialized and pipelined module to something more similar to a general purpose programmable processor. One of the most relevant differences between a GPU and a CPU is the memory hierarchy and how the module is interfacing the memory. Due to the application specific purposes of GPU, which usually requires a large amount of data that is going to be processed in parallel, many key features present in the CPU architectures here have a secondary role. For the same reason, the GPU is provided with a larger memory bus and special hardware features (texturing hardware, for example) used to improve the bandwidth of the memory bus and speedup certain applications. This is necessary because of the high level of multiprogramming that can be found in a GPU application, where thousands of threads can run in parallel.

In the following sections, some of the frameworks of GPU parallel computing are presented in order to represent some application specific behavior and its relationship with the memory management and the issuing of memory requests. The aim is to present examples of how a GPU manages the stored data in order to provide an idea of some behaviors that had been considered, and will be explained in the next chapters, while designing the Priority Based Scheduling Algorithm.

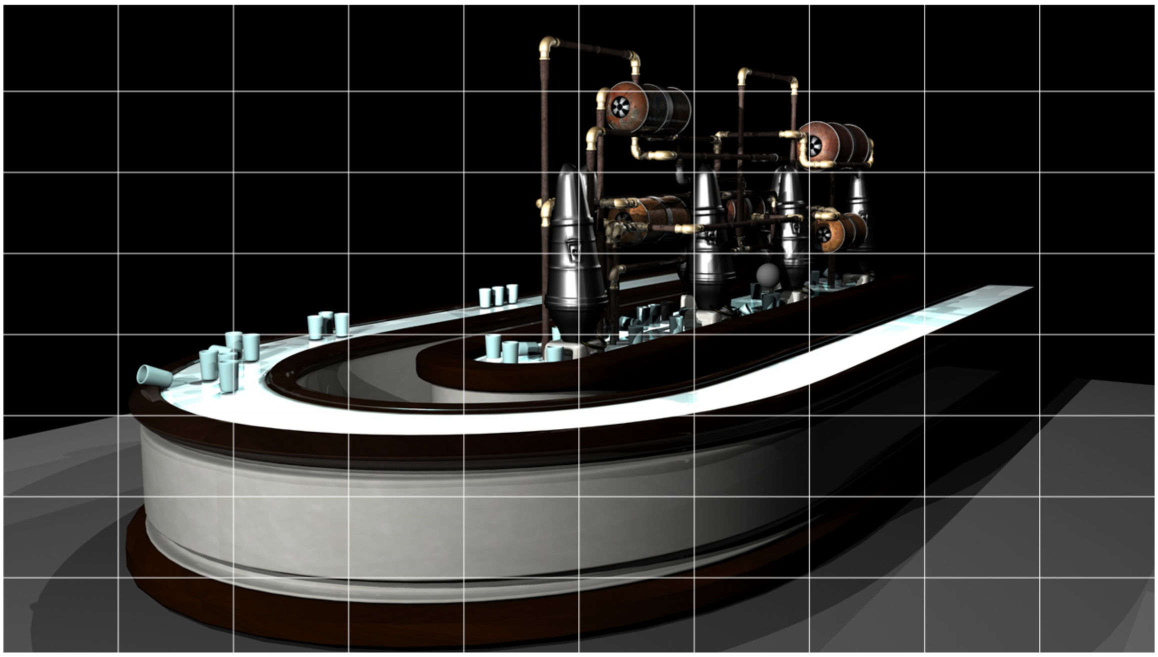


Figure 6: Tiling-based rendering

2.3.1 Tiling

Tiling is a common framework used in the development of GPU related applications. The concept is to subdivide the whole workload into subsets of independent data, and apply a Divide and Conquer methodology (Figure 7). This approach is very common in many Computer Graphics applications and almost every real-time rendering application like videogames and dynamic physical effects run-time simulations. The static images generated from the projection of the scene into an image plane using ray casting techniques, are divided in a grid of smaller images. The rendering process is then decomposed in many smaller sized rendering processes that can exploit the parallel computer architecture offered by the GPU. In fact, the geometry data of each tile is assigned to one GPU core. Since the dataset of each tile is independent from the one of the other tiles, no additional synchronization protocols are required because every computation can be performed independently. Then, when each core has processed the input data and produced the pixel information, this is written in the main memory. The main advantage of this approach is that the process is not sensitive to the latency. Due to the large amount of data processed independently, in case of momentary unavailability of a data set, a GPU core can start processing another tile and return back to the original later. On the other hand, due to the fact that the problem is subdivided in a lot of subsets processed in parallel, the whole operation requires a large amount of bandwidth. These concepts, and the different constrains in terms of latency and bandwidth, will play an important role in the design of the Priority-Based Scheduling Algorithm as explained in the following chapters. Tiling Based Rendering has been developed further and is now considered one of the main frameworks in rendering technologies. Ultimately, it has been established as one of the main techniques not just in high level environments, but also in Embedded Systems platforms (13).

2.4 CPU-GPU Heterogeneous Systems

As seen in the previous sections, the communication between CPU and GPU can represent a considerable factor in terms of overhead because of the latency generated by the data transfer and by the synchronization operations. Among others, this is one of the reasons behind the development, by the major processor's manufacturers, of Fused Architectures, where both the computation units are placed on the same die, decreasing overheads and improving the performances in terms of timing and power consumption. In order to solve the PCIe problem described in the previous section, having the modules on the same chip gave the opportunity to designers to replace the inefficient PCIe bus with an unified North Bridge. Then, depending on the manufacturer, several different architectural choices have been implemented to improve the modules' communication. An example could be represented by AMD's Radeon Memory Bus (14), that improves GPU access to local memory providing high bandwidth, and the Fusion Compute Link, which is part of the implementation of the Cache Coherency System shared among CPU and GPU. Obviously, due to the many differences in terms of requirements of the two modules, some trade-offs have to be introduced in order to have an architecture that is flexible for all the needs. The memory itself, has to be chosen accordingly: on one hand, a DDR3 DRAM memory would be the best choice for the CPU, because it would provide additional capacity with respect to the GDDR5 memories, paired with GPU modules to offer an improved bandwidth. In a standard architecture, it would be possible to pair one memory chip (DDR or GDDR) with its own computation core improving the performances of each of them according to the chosen technology. But in a Fused Architecture, where the physical memory is shared, a trade-off has to be implemented in order to guarantee capacity without limiting the bandwidth. Recent studies (15) have proved that both approaches have its own advantages and disadvantages and, in the commercial environment, both have been adopted with different results.

Another parameter that has to be taken into account is the resource allocation to support both the sequential flow of execution of the CPU and the parallel computation of the GPU module. The two different purposes of the modules represent an additional complexity due to the fact that, from a microelectronics and architectural point of view, in order to improve one behavior, the other has to be partially neglected. One high level solution would be to improve the scheduling methodology, which is the topic of this research and will be extensively discussed in the following sections. But from a low level and electrical point of view the problem can be formulated in terms of resources. If a fixed amount of transistors on the shared die is considered, how many of them have to be allocated to latency sensitive units and how many to the parallel processing counterpart? Again, the solution is to accept a trade-off between prioritizing one module with respect to the other and different approaches and architectural choices have been developed and implemented by the main Fused Architectures manufacturers.

The system under study can be simplified and represented as a model made of two modules and one single shared Memory Hierarchy system which processes all the requests coming from the modules through a single and shared Memory Controller. The first of the two sources, represents a CPU defined here as a set of computation cores loaded with a General Purpose application (benchmark). The second module represents a GPU, defined as a collection of computation cores loaded with a Graphics application (benchmark). From studies already published in Literature, and the considerations presented in the previous sections, it is possible to determine the different constrains, requirements and sensitive parameters of the two modules. While these differences are introducing more complexity on the presented environment, on the other hand can be exploited to build an application-aware Scheduling Algorithm, which from these same differences can produce benefits in the timing performance of the system. Due to the different kind of applications loaded into the modules, and based on the previous analysis on each module criticalities, different parameters play different roles in each module and each application.

In the general purpose CPU applications, usually the majority of the computation has been already performed before the issue of the memory requests. This analysis (16) has been extensively exploited, as a feature, in many previous Scheduling Algorithms. For this reason it would be expected to have strict latency-related constrains due to the fixed deadlines imposed to commit each instruction. Due to this characteristic, in many Scheduling Algorithms, by default, the CPU is assigned with the highest priority.

In a graphics application run by a GPU module, on the other hand, the latency-related requirements are more relaxed due to the high amount of parallelization granted by the GPU architecture and exploited in numerous ways in each application. If the module is waiting for stored data, it can still work on a different workload and the latency can result to have a minimum impact on the whole performance. Furthermore, usually the different sets of data on which the GPU is working in parallel, are independent.As presented in the previous sections, Tiling-Based rendering, pixel manipulations, and many other applications can provide a high degree of parallelism due to the fact that the same operation is performed to a huge amount of different and independent input data. According to this model, while the GPU system can tolerate a higher latency with respect to the CPU one, it appears more sensible to the bandwidth of the memory controller. In fact, since the GPU has to work on large data sets, the exchange of data between the memory and the module is consistent and usually much heavier than the one issued by the general purpose CPU. From this overview, in an environment such as the one under study, both the constrains play an important role and the different nature of them can be exploited to optimize the overall performance in terms of fairness and efficiency, using priority assignments able to guarantee the respect of CPU deadlines and bandwidth reservations to support the GPU module. Furthermore, in numerous published papers, it has been highlighted the fact that often, in a CPU-GPU heterogeneous system, the CPU computation-intensive phase does happen at the end of the memory access phase (17). On the other hand, from many application-related analysis, it is possible to observe that also the number of memory accesses issued by a GPU is not constant but discontinuous, where periods of intensive memory exchanges are followed by a lighter traffic on the memory controller. In an environment as the one presented, this feature can be further analyzed and used to schedule the accesses of a module during the computation-intensive phase of the other one, in order to maximize the throughput of the memory exploiting the different overhead phases of the CPU and the GPU.

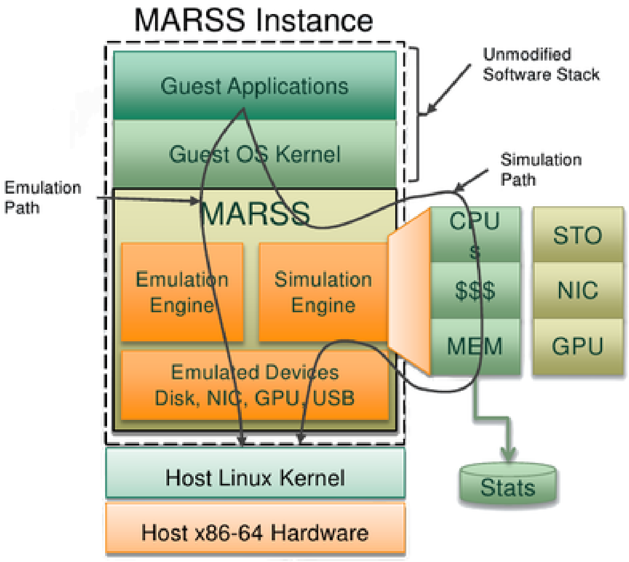


Figure 7: MARSSx86 architecture

2.5 The Simulation Environment

In order to perform the simulations needed to run the algorithm and analyze the results with different configurations and on a wide set of workloads, a customized version of the MARSSx68 (18) and DRAMSim (19) simulator has been adopted. This original project combines the MARSSx86 simulator, developed by the State University of New York, and DRAMSim2 a cycle-accurate memory simulator developed by the University of Maryland. The two platforms have been merged to provide an accurate and complete simulation environment useful for very detailed memory analysis.

2.5.1 MARSSx86 Simulator

Marssx86 is a cycle-accurate full system simulator used for x86-64 computer architectures built over the PTLSim simulator but improving the environment with additional features and better performances in terms of simulation time (Figure 8). Additionally, it provides a simple but powerful way to declare and implement different fully-customizable computer configurations. The original version of MARSSx86 includes a Memory Hierarchy system, not described here because substituted by the one implemented in DRAMSim2. Due to the emulation support provided by Qemu, it is possible to exploit several features of the virtualization to set numerous simulation settings like the number of instructions to be executed or the number of iterations. Additionally, a wide set of different coherent caches and interconnections is included in order to simulate and customize a whole system. It is possible to build a machine using YAML files and describe each component and interconnection, then use the written configuration inside the emulation environment to execute one of the many supported sets of benchmarks. Since the main focus of this research in terms of machine configuration was represented by the memory, the whole structure of the MARSSx86 has not been modified except, as described below, for the interface with DRAMSim2.

2.5.2 DRAMSim2

DRAMSim2, which was initially designed as a trace-based simulator, in combination with MARSSx86 is part of a whole simulation environment and provides a detailed, cycle-accurate analysis of the main memory in terms of time and power performance. Every single part of the simulator is highly configurable and, for this reason, DRAMSim2 could be a flexible platform

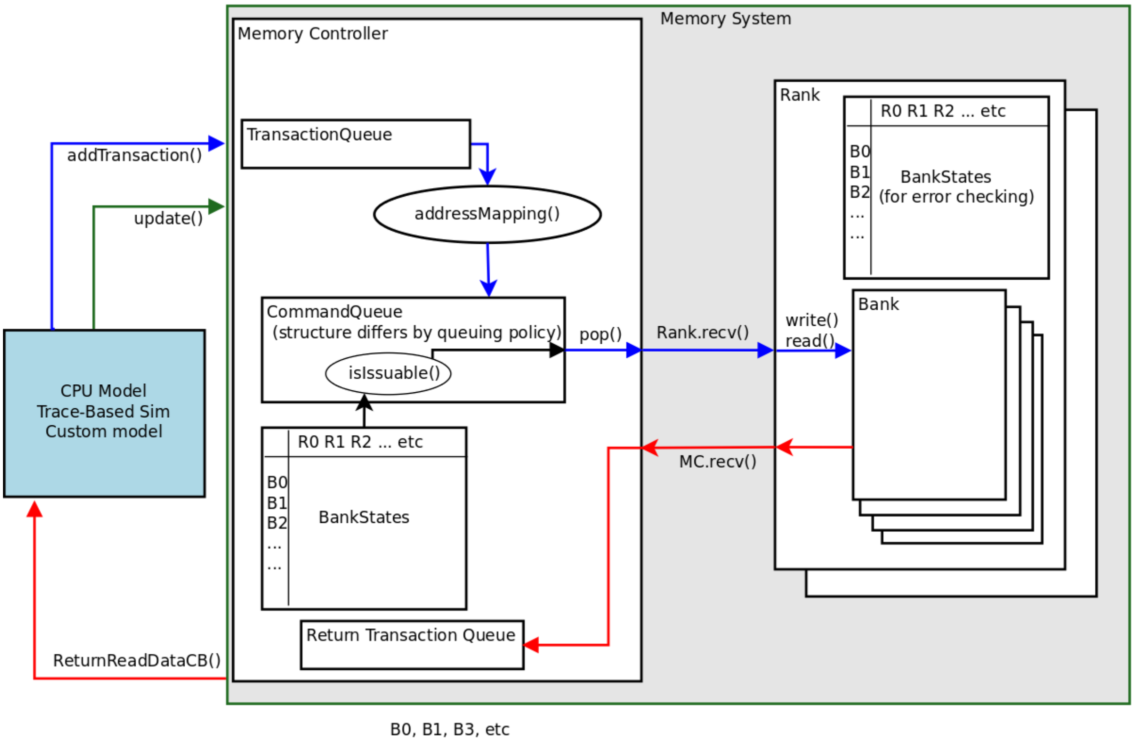


Figure 8: DRAMSim2 classes and relationships

for the design and testing of memory components. DRAMSim2 is designed following a modular approach, where each part of the system provides support functions for the higher level modules (Figure 9).

2.5.3 DRAMSim2 Components

The Memory System is the most external component of the whole platform. This is the interface between DRAMSim2 and MARSSx86 and is the only object directly accessed by the CPU simulator. Its function is to be a wrapper around the other components and to trigger the update operations for the Memory Controller and Rank components, which performs all the operations that have to be done at each clock cycle. The Memory System receives from the CPU simulator the Transaction objects that are propagated to the Memory Controller and managed by the same component.

The Memory Controller is the core of the Memory Hierarchy system: it is in charge of the translation of CPU memory Transactions into BusPacket objects and it is where these objects are managed and enqueued in the correct structures managed by the Command Queue object. Furthermore, the Memory Controller is in charge of managing the BankState objects, databases used to store the current states of each bank of each rank. The main task of this component, in summary, is to provide all the services needed to let the communication between lower level modules happen.

The BankStates object provides the data to the CommandQueue object and is updated by the same element. It is a database that stores the current status of each bank of each rank and additional information like countdowns and number of accesses. It is queried by the Command Queue and the Memory Controller, in order to establish if a Bus Packet is issuable or not. During the instructions' execution, a bank can be in the Idle, Precharging, Rowactive, Refreshing or PowerDown state. According to the current state, some operations can be issued and executed while others are forbidden, in order to ensure coherency with the electrical modules.

The CommandQueue object is where the original queue of memory commands is placed and where the majority of the Scheduling logic is executed. This object is basically in charge of three functions: enqueue the requests, process them, and provide them when a pop operation is issued by the Memory Controller. The enqueue function is called when a new issued packet is injected in the system from the CPU and through the Memory Controller: in the original implementation the queue is chosen according to the rank and the bank of the packet. The CommandQueue is also in charge of the Scheduling of the memory accesses: in the original implementation it is very simple: a round robin visit across all the ranks and all the banks. Finally, the CommandQueue has to provide a BusPacket to the higher levels of the memory hierarchy through the memory bus. According to the BankState information, which is updated each time a Packet is transferred from the queues to the Memory Controller, a packet is picked up and served or, if a Refresh or Precharge operation has to be issued, it is built in order to trigger one of these two operations in the next clock cycles.

The physical implementation of the memory is represented by the Rank and Bank objects, which are accessed every time a command is processed and a Read or Write operation is executed. The number of modules is defined by the system configuration file, which sets the number of instances for each component and builds in this way the whole physical memory implementation.

2.5.4 DRAMSim2 Modifications Overview

In order to build the full implementation of the algorithm, the entire DRAM simulator has been modified to propagate some useful data from the CPU simulator to DRAMSim2, where the Scheduling Algorithm is applied. The DRAMSim2 environment is built following a modular approach, so it is possible to edit just some components of the whole system with a minor impact on the other parts. For this reason, while the Command Queue has been almost fully rewritten in order to implement the new Scheduling algorithm, the other parts have sustained lighter modifications. Several edits on the Bus Packet object have been applied to the code in order to provide DRAMSim2 with all the necessary information coming from the CPU. First of all, both the core ID and thread ID had to be propagated to give the memory controller the opportunity to distinguish memory accesses coming from the CPU from memory accesses issued by the GPU. Furthermore, in order to implement a control over the waiting time of each single packet and avoid starvation, each packet has been marked with the time when it was issued by MARSSx86. Finally, when running a benchmark with numerous accesses to the same physical address, since both the "per rank per bank" queuing system and the priority queuing systems are implemented, it was necessary to have an identification element for each packet to avoid the accidental elimination of a packet representing a memory command destined to the same rank, bank, row, column of the one which has to be erased from one of the queuing systems. For this reason, each time a packet is built from a Transaction, a unique Packet ID is assigned. The Memory Controller object has been modified just to perform the coherent conversion between MARSSx86 issued Transactions and the Bus Packets. Additionally, since it is in charge of providing to the user the results data, it has been modified to produce information that reflects the new memory controller architecture, and the output now represents the data according to the new multi-queue Command Queue system.

CHAPTER 3

SCHEDULING ALGORITHMS

As introduced and explained in the previous chapter, many different approaches have been developed to improve the overall performance of CPU-GPU memory systems. All these strategies are spanning a large amount of layers, from low level electrical concepts, to high-level software related optimization. One of the most important concepts considered in the pursuit of new improvements is the Scheduling Algorithm for Memory Accesses. This subject plays an important role in many of the problems described above. First of all, it is one of the main tools when dealing with Fairness related problems, where it is necessary to establish a fixed or dynamic access order among multiple sources. A huge amount of research activities have been spent in the development of Scheduling Algorithm for multi-threaded CPU architectures and GPU architectures. While the topic of Scheduling in CPU-GPU architectures can not be considered recent, the proposed solutions on the topic are less numerous than in the previous two examples. In the following sections, two of the many previous approaches are introduced and described in order to provide a set of the different strategies adopted to deal with this subject over the years and in recent Research projects. Then in the following part one particular solution, the Staged Memory Scheduling Algorithm, is taken into account and described in detail as the reference that will be used in the last chapter where the simulation results are presented and compared.

3.1 Overview of Previous Approaches

In the development of a Scheduling Algorithm for CPU-GPU architectures, different approaches have been explored in order to improve the overall system performance or specific parameters like Fairness or the throughput of a subset of cores. Many strategies, while improving a set of metrics, can decrease the performance of other parameters and for this reason usually a trade-off is necessary across all the aspects of the system. As described in the previous chapter, in a CPU-GPU architecture, usually the goal is to provide a minimum latency to the CPU while guaranteeing enough bandwidth for the GPU module. This fact is particularly true if we consider GPU applications like in the Computer Graphics development, where large sets of data are processed in parallel. But from a higher level point of view, the GPU has still to guarantee some kind of reliability in terms of high level metrics. As example, during the execution of a real-time rendering in a videogame or another real time image-processing operation, it is possible to measure the overall performance in terms of Frame Per Seconds (FPS).

3.1.1 Priority Assignments According to Frames per Second

The FPS value is related to the ability of the human eye to perceive a flow of images as a continuous stream without acknowledging the sequence of static images. Usually, the threshold between a fluid video streaming and the perception of the sequence of images is around 18 FPS, but in modern multimedia applications a stable rate of at least 25 FPS is required. This constrain has been exploited, in CPU-GPU architectures, to determine a strategy for the priority degrees assignment across the different computation cores. This is a high level metric that provides a fixed constrain to the Scheduling Algorithm, that can dynamically determine the priority assignment based on a prediction on whether the GPU will be able to provide the desired frame rate or not. An approach similar to the one mentioned here has been developed in the paper ”A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC” by Min Kyu Jeong, Mattan Erez, Chander Sudanthi, Nigel Paver (20). In this paper, the scheduling algorithm performs an analysis on the GPU flow of execution to determine whether the module will be able to provide a desired rate in terms of FPS and adjusts the priority degree accordingly. Due to considerations explained in the previous section, the algorithm tries, by default, to prioritize the CPU according to the fact that usually it provides an improvement of the timing performances. It defines, for each frame, an Expected Progress (EP) value that is constantly compared to a fixed value. If EP is lower then the threshold, the GPU's priority is risen up and its issued memory accesses commands are managed as they were coming from the CPU, assigning the same priority degree. Otherwise, if the GPU execution is respecting the boundaries, the CPU is provided with the highest priority. One interesting feature of this approach to the problem is that no low level architectural optimization is applied. The QoS Aware Algorithm performs just an analysis on current performance without trying to tweak any memory feature like space and time locality to achieve an improvements on the performance.

3.1.2 Priority Assignments According to Dynamic Behavior Analysis

The concept of performing a run-time analysis on certain characteristics of the memory systems in order to act on the Scheduling process, that was present in the previous FPS based example, can be found in lower level approaches to the Scheduling problem. The idea is to focus the monitoring phase to other features, closer to the Architectural layer of abstraction. One example is the strategy proposed in the paper ”Phase Aware Memory Scheduling” by Chirag Sangani, Mathangi Venkatesan and Rakesh Ramesh (17). The idea is to start from analysis performed on the memory traffic of the whole system, establish them as assumptions, and build the algorithm around these statements. This approach has been also adopted in the definition of the Priority Queues Scheduling Algorithm. In the case of the Phase Aware Memory Scheduling Algorithm, due to an extensive work on the emulation environment, it was possible to observe that the overall GPUs executions present two phases interleaving periodically. In detail, the GPU present periodic patterns that show memory intensive intervals, where the module is exploiting a huge amount of the available memory bandwidth, and computer intensive intervals, where the memory accesses are relatively limited and the module is busy in computation operations. The idea of the algorithm is to build a Scheduling strategy able to monitor these phases and schedule the CPU memory instructions in the time periods where the bandwidth is not saturated by the GPU traffic. The algorithm is application related, taking into account a behavior that is common in computer graphics workloads like videogames and image processing operations, and can be designed and built over analysis' and simulations made a priori. But differently from the previous example, the analysis and the core of the strategy is built over a lower level, on the memory traffic of the whole system and not at the application layer. Finally, one of the main differences from the previous Algorithm is that the behavior of the system is analyzed, while in the previous case the application related constrains played were at the center of the priority assignment logic.

3.2 The Reference Algorithm: Staged Memory Scheduling

This algorithm, published in 2012 in the paper Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems by Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, Onur Mutlu (1), has been used as reference in the estimation and comparative analysis for the developed Priority-Based algorithm. The goal of this implementation is to build a Scheduling Algorithm able to perform a choice over the instructions to be picked from the queuing system based on two different levels of abstraction. On a higher level, the algorithm shrinks the selection according to the module it wants to serve while, on a lower level, the algorithm tries to sort the commands according to the row they‘re trying to access. The whole scheduling process, for this reason, is split and performed in two different stages. The ultimate goal, achieved as demonstrated by the performance results in the paper, is to accomplish an improvement on the timing performances of the CPU, while maintaining the GPU performance almost unchanged. One of the most interesting features of this algorithm is that the priority assignment is not based on application specific considerations but on probability assumptions that determine whether the scheduling is performing a Round Robin or a Shortest Job First strategy across the sources. Additionally, since the priority depends on the probability value introduced in the system, the Scheduling Algorithm appears fully customizable according to the desired priority configuration used to optimize the overall performance. Furthermore, one of the main achievements of the Staged Memory Scheduling algorithm is to solve the problem of the closing of the Row Buffer from instructions coming from different cores. As described in the previous chapter, in fact, the row buffer is used as a cache for the main memory accesses. Since the machine is comprehensive of multiple computation cores, in the same time period, different cores access different rows. For this reason, the row buffer would be closed and opened several times, lowering the hit ratio and worsening the timing performance. But since the requests are already sorted by accessing row, the hit ratio on the banks is maximized. From the architectural point of view, the scheduling algorithm can be designed according to its logical staged implementation, dividing the whole design into sequential stages. Logically, this can be seen as a chain of different steps where each logic develops its own part of the scheduling algorithm.

This algorithm considers different sources as, for example, CPU cores and GPU cores. A source is defined as one of the modules that issue a set of Memory Commands. The algorithm is structured in three stages as follows:

1. The algorithm picks up, for each source, all the commands and groups them in batches having the same row. So each memory request of the same batch accesses the same memory row.
2. This higher level scheduler picks up batches according to Shortest Job First or Round Robin Policies.
    - Shortest Job First: it picks up the oldest ready batch from the source with the fewest total in-flight memory requests across all the three stages of the Staged Memory Scheduling Algorithm. It favorites latency-concerned instructions.

  - Round Robin: it picks up the next ready batch scanning each source. When it arrives to the last one, the cycle begins again from the first one.

3. Bank level commands are issued in order to retrieve the required data. Here the algorithm works at a lower level and basically performs the standard tasks of the memory controller to manage the memory accesses because the requests have been already optimized for row access in the previous steps.

3.2.1 The Visibility Problem

The main problem highlighted by the considered paper itself is the Visibility. Since the number of GPU requests is usually much larger than the number of commands issued by the CPU, it can happen that while the Scheduling Algorithm tries to acknowledge which command is the best one to be processed, the low number of CPU requests can be overshadowed by the heavy GPU workload. One possible solution, highlighted by the paper's authors, is to use a large buffer, which would be able to store every request and, at the same time, provide a clear information about the order of incoming instructions in terms of time of arrival. The solution, proposed by the SMS Algorithm Implementation is to split the single queue and to have one buffer for each source, intended as computational core. This solution is similar to the one adopted by the Priority-Based Algorithm, where instead of having a queue for each core, the implementation is made of two application-related queues: one for the CPU computational cores and one for the GPU computational cores.

3.2.2 Open Problems

The first problem with the algorithm is directly related to the main advantage, the opportunity to have a high number of sequential commands querying the same DRAM row. Since a ready batch is defined as a sequence of one or more requests accessing the same row, it is necessary to introduce a countdown to unleash the batch even if a command addressed to a different row has not been issued yet. While a countdown to issue an uncompleted batch is necessary to avoid excessive wait times for incomplete batches, on the other hand the performance can be related to the type of workload loaded. The influence of the countdown would be almost negligible when considering workloads accessing the same row for a limited number of times but could have a negative impact on performance when the memory controller is dealing with a large number of requests addressed to the same row. Another problem related to the SMS algorithm is the possible starvation when dealing with heavy loaded sources. The system under study is loaded with graphics applications, and as explained in the previous sections it is correct to assume the GPU will issue a large number of commands. According to the algorithm, once a batch is picked up every instruction contained in it has to be issued to the higher hierarchy levels in order to go on and perform the same on the next batch. Now, since the number of GPU requests can be larger than the number of requests coming from the CPU, it can happen that a very large GPU batch is created and has to be issued. Assuming that one single command is processed at each clock cycle, it‘s possible to understand the chance that a large GPU queue could need a considerable amount of clock cycles to be processed in order, before letting the controller move on to the next computational source. Finally, in the second stage, which is in charge of choosing the source from where the next instruction will be picked up, there is not a direct methodology to pick up an instruction from a CPU or a GPU source. From the previous analysis, it is possible to understand how the different constrains of the general purpose and graphics applications are related to the priority assigned to one of the two modules. In the SMS algorithm, the accesses to the sources is performed in a Round Robin way, which does not introduce a priority based choice, or Shortest Job First, which should prioritize the CPU commands. The priority is assigned in terms of probability, choosing one protocol over the other based on a fixed parameter p but a direct and dynamic strategy according to the current queue system status is not implemented.

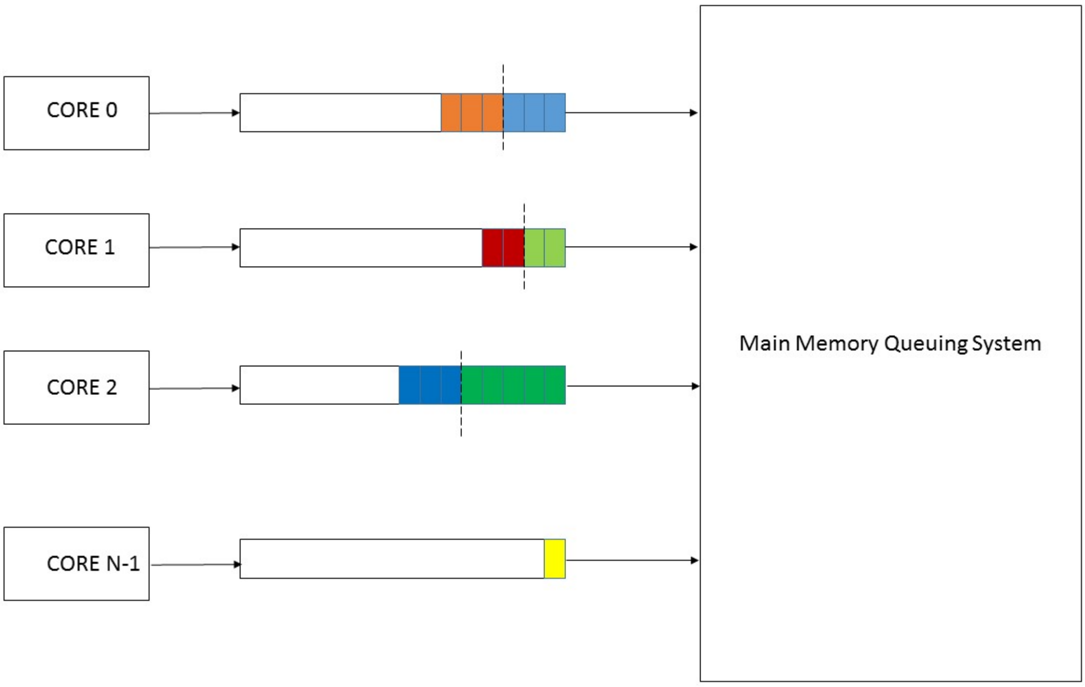


Figure 9: SMS batch creation scheme

3.3 Staged Memory Scheduling Implementation

The implementation of the Staged Memory Scheduling Algorithm follows the same logical scheme as a pipelined series of operations split in different steps of the same Algorithm. Below, a list of all the steps is presented with a description of all the main operations performed in that same step.

- Stage 1

  In the first stage of the Staged Memory Scheduling Algorithm, the requests of each source are sorted and grouped in batches having the same row access (Figure 10). So we have a queue for each source and for each one we need to implement an algorithm that sorts the requests and divides it in batches. In order to implement this mechanism, the enqueue function of the CommandQueue object is modified and the instructions are inserted in order, according to the row that has to be accessed by the memory request. Furthermore, it is necessary to know in advance the number of cores that are used during the simulation in order to implement one queue for each core. According to the paper, only READY batches are taken in the second stage. A batch is ready when, after a sequence of one or more requests accessing the same row, we have a requests that tries to access a different row.

- Stage 2

  Here the algorithm performs a higher level scheduling and chooses from which core it has to pick up the next batch of instructions. It is done according a Shortest Job First (SJF) policy, or a Round Robin (RR) policy. Using the round robin policy, the scheduler takes the next ready batch from the next queue. For this reason, when we pick up a batch we have to consider if this is a ready batch, a sequence of requests to the same row followed by at least a memory request accessing a different row. Following the DRAMSim architecture, it is possible to implement it using sequence checks when the getCommandQueue method is called by the CommandQueue object.

- Stage 3

  In this stage the batch has been selected and in the following N clock cycles, the scheduler picks up a Memory Command from the chosen batch and processes it, where N is the number of commands in the selected batch. In order to implement this feature, some checks on whether the batch has been taken or not are performed and the selected batch queue is stored for the next clock cycles until it is void.

CHAPTER 4

THE PRIORITY-BASED SCHEDULING ALGORITHM

In the following sections the core of this research work, the Priority Based (PB) Algorithm, a brand new Scheduling Algorithm for CPU-GPU workloads aimed to be applied into CPU-GPU architectures is introduced. The goal of this project is to improve the overall performances of the system implementing a dynamic priority scheme based on the majority of the architectural and behavioral analysis' performed in the previous sections. The design of the scheduling algorithm finds its roots in an extensive study of previously developed theoretical knowledge of the common memory patterns in CPU, GPU and CPU-GPU architectures. In the first section, these assumptions are summarized and presented all in one place. Then the architectural high level design of the algorithm is presented. Many of the implementation details are here omitted or just introduced because will be part of the implementation description that will be covered in the third section. Finally, details about the code implementation are described in order to present the practical code tasks developed for the accomplishment of a working prototype of the algorithm.

4.0.1 Assumptions

The Priority Based Scheduling Algorithm is essentially an application focused strategy. The concept is to provide a scheme that can improve the performance of the whole system when it is executing a set of common workloads with determined characteristics. In the definitions of these behavior assumptions, the objective is to focus the attention on applications that would be common in a real world scenario. Given that, the idea was to build an algorithm that could exploit certain features that have place when loading the CPU with a General Purpose workload and the GPU with a Graphics or Multimedia application. These are very common environments, and offer a platform where it is possible to formulate a few assumptions that have already been mentioned in the previous section and are here summarized as a base foundation for the developed algorithm.

1. Bandwidth and Latency

   As seen in the previous chapters (Section 2.4), and as highlighted in many previous Research works cited above, usually the CPU General Purpose workloads are latency sensitive. Due to fixed deadlines, in fact, it is necessary to reduce the time CPU memory requests are served. On the other hand, due to some execution framework like the Tiling Operations described as example in the previous chapter, the GPU can usually tolerate higher values of latency even if it usually requires a large data bandwidth to process huge parallel data sets. The result of this considerations is that by default, to avoid deadline misses, it is better to prioritize by default the CPU module.

2. Number of Memory Accesses

   As explained before (Sections 2.2 and 2.4), usually the CPU issues a lower number of memory accesses with respect of the GPU. Again, this is an assumption that is related to the bandwidth issues previously described, but is built over different tests where workloads from CPU-only environments are compared with GPU specific applications. From these analysis it is possible to observe that even the most memory stressing CPU benchmark presents a much lower stress on the memory in terms of accesses. This is an experimental assumptions that brought to the design of a Scheduling Algorithm that tries to optimize scenarios where this condition is met and, for this reason, that would improve common environments as the one here described.

3. The Number of Memory Accesses is not Constant over Time

   As highlighted in the Phase Aware Memory Scheduling, and as highlighted in the previous section (Section 3.1.2), when examining the overall memory traffic in a CPU-GPU environment it is possible to observe specific periodic patterns where the number of memory requests bursts to a large amount, and then falls down to lower values. This is noticeable in the CPU workloads but it is even more evident in the GPU workloads where the number of requests is larger. As explained in detail before, this characteristic is a consequence to the fact that usually computation periods are interleaved with memory stressing periods. Additionally, the GPU triggers the majority of the memory requests only once the computation period is over. The result of these previously developed analysis' is that the number of instructions coming into the DRAM system is not constant over time but is highly periodic.

4. Row Buffer Hits

   As seen in the previous section (Sections 2.1.5 and 3.2), the Row Buffer Hits have to be increased in order to improve the overall timing performance. A Row Buffer Hit means that the data we're accessing is already stored in a cache implemented to exploit the Space and Time Locality. So the overall access time of an instruction trying to access the Open Row of its Bank is lower than the one required to access a closed row. For this reason, it is necessary to improve the number of processed memory requests when a row has already been opened.

4.1 Priority-Based Algorithm: Architectural Design

The main concept of the Priority Based Algorithm is to build a priority assignment strategy based on the set of behavioral patterns described in the previous section. Due to the fact that a reordering of memory commands is performed, it is necessary to surround it with additional assets used to provide a reliable platform that avoids unattended effects like starvation of one or more cores and memory commands stuck due to the different order in which the instructions are processed. The design of the Scheduling Algorithm, from an architectural point of view, is quite modular and the main components are here presented and discussed in detail.

4.1.1 The Queuing System

The core concept behind the design of the Priority-Based Scheduling Algorithm is to provide a system able to speedup the memory requests coming from the CPU with respect to the GPU, but without affecting too much its performance. From these intentions, the objective was to implement a trade-off able to prioritize, by default, the CPU, but avoiding a too penalizing behavior for the GPU. In order to achieve this goal, a brand new queuing system has been developed and implemented into the architecture, and the main components are here introduced and discussed (Figure 11). As a remainder, it is important to highlight that as described in the previous chapter, the memory controller is able to know which core has originated each injected memory command.

1. CPU Queue

   This queue is used to store the memory commands coming from a CPU source, a core that is loaded with a General Purpose CPU workload. This queue, by default, will be

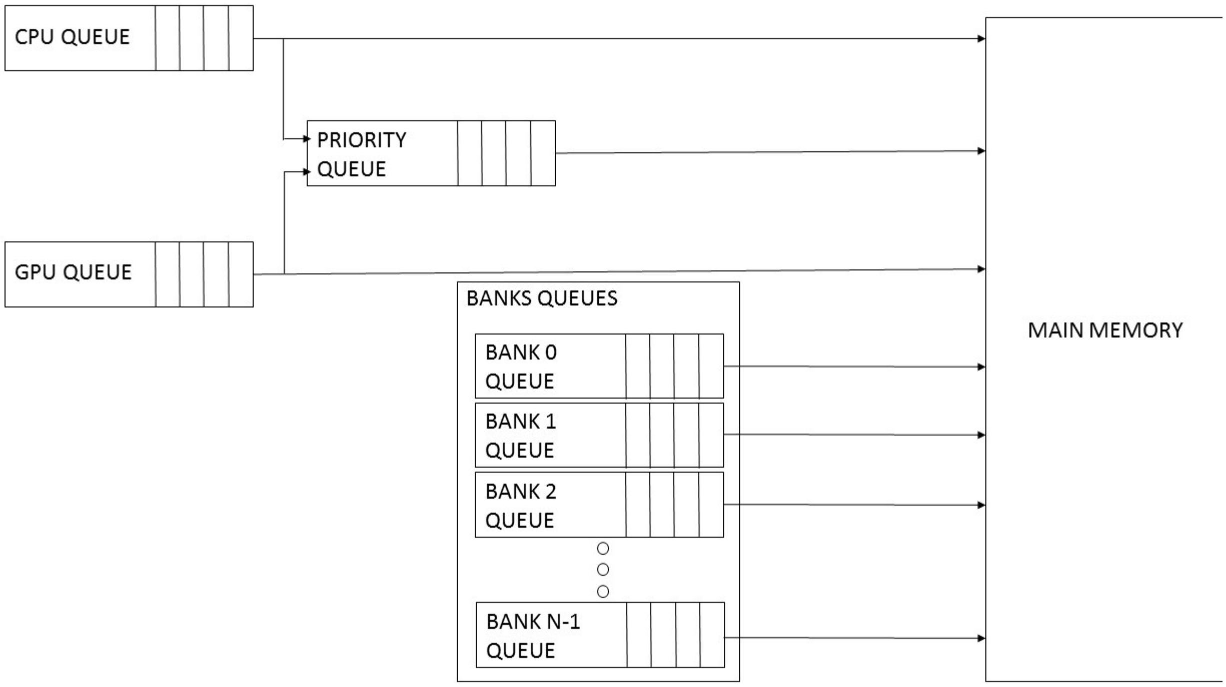


Figure 10: Queuing system in the priority based scheduling algorithm

prioritized by the Scheduler with respect to the GPU one and, if the Priority Queue is empty then it will pick the instructions from this queue.

2. GPU Queue

   This queue is very similar to the previous one and is used to store memory request commands coming from sources (computation cores) loaded with GPU workloads. This queue is usually accessed when both the Priority and CPU queues are empty.

3. Priority Queue This queue is used to process the Relocation Operation, that will be discussed in the following section. It is the first queue that is tried to be accessed by the Scheduler and can store both CPU and GPU memory commands. The instructions are enqueued here only when the Scheduler has to balance the whole queuing system to avoid GPU starvation. For this reason, since the Priority queue is not empty when a threshold has been crossed, it has the highest priority.
4. Bank Queues This queue system is similar to the default one used in DRAMSim2. Basically, for each bank of the memory system a queue is implemented. In the Priority-Based Scheduling Algorithm, these queues are just used to manage the memory commands operations that have to be performed when a Refresh operation is triggered and a direct access to the memory commands of a single bank in a determined rank is needed. In the majority of the memory accesses, this queue system is not used. Each queue of each bank, is a First-In-First-Out architecture that stores all the previously issued memory commands directed to a row of that rank-bank subset. Actually, this support queuing system can be omitted using more complex and inefficient search methods on the previously described queues, but can be useful to have a fast and direct access during the Refresh operation management.

4.1.2 The Relocation Operation

As described in the previous sections, by default, the Priority Based Scheduler Algorithm prioritizes the CPU over the GPU. The amount of priority, and the mechanics used to provide a reliable way to process all the memory commands coming from all the computation cores

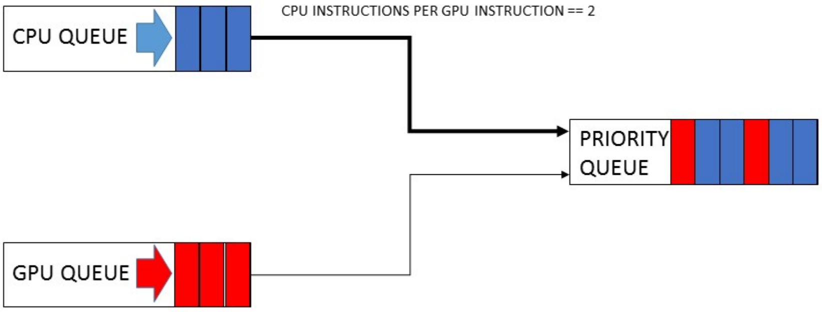


Figure 11: The relocation operation

are based on the triggering of the Relocation Operation (Figure 12). This function is used to bring balance to the picked up instructions when too many requests coming from the GPU are stored in the system or according to other parameters related to the time the instructions are spending into the system. The Relocation Operation is triggered by the Scheduler Priority Assignment Policy. When it happens, the memory commands stored in both the CPU and GPU queues, are relocated into the Priority Queue according to different parameters. In this way, all the instructions of the system are moved and sorted in order to balance the way the next instructions will be picked up and hopefully, to avoid excessive waiting times and too high latency values for the GPU issued instructions. Obviously, the Priority Assignment Policy has to be designed properly to decrease as much as possible the number of times the function is called and, as a consequence, an inefficient scenario where the Priority Queue is continuously fulfilled. Furthermore, another policy that had to been designed is how the instructions in the CPU and GPU queues are relocated into the Priority queue. When the Relocation Operation is triggered, it means that a certain threshold has been crossed and a balance is needed to avoid GPU starvation. The system is in a situation where the Scheduler must modify the Scheduling Policy in order to output, through the Pop operation, a number of GPU instructions and restore a balance where it is possible to prioritize again the CPU, optimizing the performance. For this reason, the memory commands are relocated taking into account these considerations and for each CPU operation a variable number of instructions from the GPU has to be enqueued into the Priority Queue. This parameter is the Number of GPU instructions per CPU instruction and is here introduced, as design choice, statically (does not change during execution). In order to make it dynamic, it would need to be dependent from the number of instruction stored in the GPU queue, which as will be described later is part of the triggering policy. The current implementation has a fixed number of GPU instructions that have to be enqueued in the Priority queue but a dynamic triggering policy.

4.1.3 The Priority Assignment Policy

As introduced and described in the previous section, the Relocation Operation is the process that is in charge of balancing the following picked instructions from the different sources. After the Relocation process, the system should avoid GPU starvation and return back to its normal flow of execution. For this reason, the strategies developed to control the triggering of this operations are the base of the logic behind the stabilization of the whole system. The relocation operation is triggered according to two main parameters: the current number of instructions in the GPU queue, and a timeout value, the GPU Timeout (GPUT). Different simulations and tests have proven that the combination of both these parameter was the best choice to provide to the system flexibility over the number of CPU instruction picked up, and reliability to avoid the GPU memory requests starvation. The number of GPU memory commands is the size of the GPU queue and can be compared with a threshold to determine if the queue is almost full. It does not provide information about how long the requests have been waiting in the queue, but only about how many requests are stored in the memory system. For this reason this parameter is used to avoid to have too many GPU instructions that need to be brought out to the system and its main purpose is to avoid an excessive number of calls of the Relocation function. The system needs to have a controlled number of GPU instructions or it would not be able to bring them out fast enough, resulting in a continuous flow of calls for relocation. On the other hand, GPUT is a threshold used to ensure that an instruction is not waiting for too much time in the system. This reliability can not be provided just by the number of GPU instructions in the queue because, for light workload, the number of instructions could be relatively small even if the instructions had been stuck into the queue for a long time. Due to these considerations the combination of both parameters has been used as a parameter for the triggering of the Relocation operation.

4.1.4 The Reordering Management

The Priority Based Scheduling Algorithm is a reordering based scheduling. It means that, in order to improve the performances, the picked up instructions sequence using this strategy is different from the one resulting from a non reordering scheduling algorithm like a simple FIFO queue. In this implementation, however, the local precedence order is still respected. If each single core is considered, in fact, it is possible to state that each memory command is processed before the commands that were introduced later in the system. Processed but not provided as output because of physical related constrains (the bank could be busy for a Refresh or Precharge operation). On the other hand, it is not possible to state that an instruction coming from the GPU before a CPU instruction will be served in the same order. The precedence order is respected on a per core basis but not if we consider the whole system. This introduces additional complexity to the system because of some implementation problems that will be described in the following sections. From an higher level point of view, the general concept is that additional logic has to be provided to ensure that when a row is opened the attached Read or Write operation has to be performed before the issuing of a successive row activation command on the same bank. The risk in fact is to issue two consecutive Activate commands. If this happens, the row would be opened for a Read or Write operation, but then closed due to a successive Activation that would open a different row. For this reason, the Read or Write operation attached to the first Activate would be stuck into the queue and would be released only when another command addressing the same row of the same bank and rank would be issued by the computation units.

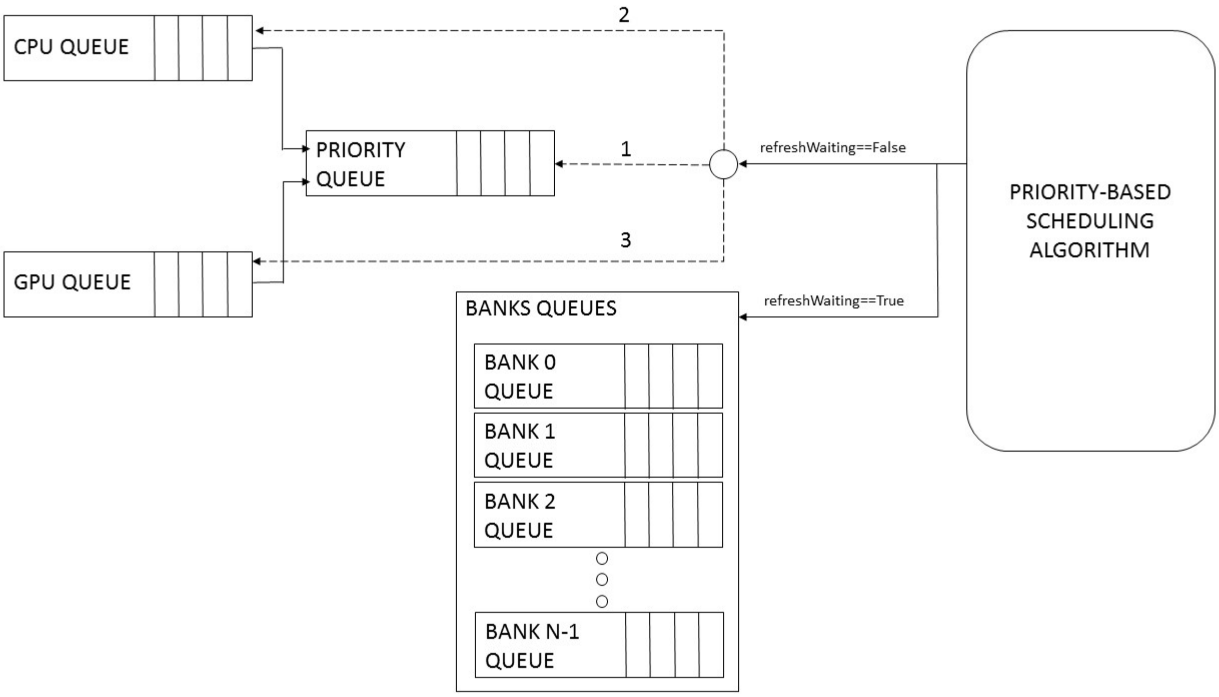


Figure 12: Queues precedence order in the priority-based scheduling algorithm.

4.1.5 The Scheduling Algorithm

Now that all the components and some of the mechanisms of the Priority Based Scheduling Algorithm have been introduced and described, this section will try to put all the pieces together in order to present a comprehensive vision of the whole system. The memory commands coming from the CPU and GPU cores are marked with the information about their respective sources. So this information is used by the memory controller to acknowledge whether the instruction is coming from a CPU core or a GPU one and is used as soon as the instruction is injected into the memory system because determines in which queue it will be placed. Once a new instruction is issued in the system, it is enqueued in the respective queue (CPU or GPU) and into the general bank queuing structure used, as mentioned before, to support the Refresh operations that needs a direct access to the instructions of a given bank and rank. Then, the Scheduling Algorithm picks up an instruction according to the following policy (Figure 13).

- If a Refresh operation is issued on a selected bank, then checks using the Bank Queues if there is a pending request on the currently open row. If the Refresh bank queue contains one or more commands addressing the currently open row, then they have to be served, else a Refresh command is issued.
- if there is not a bank waiting for a Refresh, then the Scheduler tries to access the Priority Queue: if it is not empty, a command is extracted and processed, otherwise it searches in the CPU queue and, if also this last one is empty, then the GPU queue is accessed. This hierarchical order is also applied if the queues are not empty but no packet can be issued because of electrical issuability constrains. For example, due to technical boundaries introduced by the different memory models some time, in terms of memory clock cycles, has to pass between two consecutive Activate commands on the same row.
- If no packets had been considered issuable from all the different queues and certain electrical conditions are met, then a Precharge packet is issued to close the open row and in order to limit the overhead and not waste the clock cycle.

When the packet that will be popped out from the memory system in the current clock cycle has been decided, if it is coming from the GPU queue, then the GPUT is reset to the timeout value.

Concurrently to the operations listed above, in the same clock cycle, the priority analysis and assignment process is executed. First of all, the GPU countdown is decremented by one, Then, a check on the current value and the number of instructions stored into the GPU queue is performed. If the GPU countdown is zero or the number of instructions is above a certain threshold, then the relocation operation is triggered, and the CPU and GPU queues are merged, according to the number of GPU instructions per CPU instruction value. At the end of the process, both the CPU and the GPU queues will be empty, while the Priority Queue will be fulfilled of all the instructions of the system, in a balanced sequence of CPU and GPU instructions that would provide the avoidance of GPU starvation and, at the same time, an excessive penalization on the CPU instructions latency. When it does happen, the instructions are processed in the Priority Queues order and, in the meantime, the CPU and GPU queues are accepting new packets coming from the computation cores. It is important to choose the number of GPU instructions that triggers the Relocation, and the GPU timeout properly to avoid a continuous relocation process. Additionally, a final check for starving memory requests is performed to find if any memory command is stuck in the system. The only cause would be a double Activation on the same bank without serving the attached Read or Write. If this happens, or if a Bank needs a Precharge, a Precharge packet is issued in the system. This is a reliability countermeasure to the double activation problem which is not triggered often, according to the load of simulations performed. But it can be useful to avoid, in certain rare cases, the starvation of a GPU packet. According to the performed simulations, usually the number of packets that need this operation is below the 0,005%, so the Forced Precharge packet is issued just a few times over the whole execution. When a packet is popped out from the memory system, it has to be removed from all the queues: both the copy in the CPU-GPU-Priority assemble, and the one in the bank queues set. For this reason, a complete deleting procedure has been implement to ensure that no copies of the issued memory instructions are still present in the system.

One last feature implemented in the Scheduler Algorithm is a check on the CPU and GPU queues when the total workload on the system is very light. When the Scheduler has to pick up an instruction from the two queues, it checks if the CPU queue is almost empty and, at the same time, if the number of instructions on the GPU queue is much larger, but not enough to cross the threshold. If in the CPU queues there are just a few packets, probably they have just entered the system and since the number of instructions in the GPU is much larger, a GPU instruction, available and issuable, is picked up. Subsequently, a flag is set and, at the following iteration, will tell the system to pick up the CPU instruction even if the previous conditions would be met again. This is a simple secondary support strategy used to decrease the number of relocation calls and designed to perform a less expensive balance operations over the queues in conditions when picking up a GPU instruction would not result in a worsening of performance.

4.2 Priority Queues Algorithm: Implementation

In the previous section, the Priority Based Scheduling Algorithm has been presented from an high level architectural point of view, describing each component and its own behavior and purpose inside the whole system. In this section, the same concept will be introduced explaining some implementation details. The main purpose of this is to describe some implementation choices and discuss the solutions to problems risen up during the design of the algorithm. The intention is not to comment any code structure written from the project, but focus on the problem-solving process that brought to the final algorithm implementation.

4.2.1 Implementation in the DRAMSim2 Environment

Due to the high level of modularity already present in the DRAMSim simulation environment, just a few of the many components have been modified to implement the Priority Based Scheduling Algorithm. How the simulator's architecture and code implementation has been modified to support the development of the algorithm has been extensively discussed in the previous chapter, here it will be descried how those implemented services are exploited in the actual algorithm version. From the execution prospective, the whole Scheduling Algorithm can be split and represented in four distinct parts. The first four listed can be considered as macro functions, each one implementing one of the main tasks the Algorithm has to provide. The Fifth one is a collection of small functions used to implement some services requested and triggered by the other four parts.

1. The Enqueue Procedure

   This procedure is called every time a new packet is issued from the computation cores into the memory system. It is in charge of pass the memory command through to the correct modules. It exploits one of the customized features of the simulator, the information about the source that injected the command (the core ID), to enqueue the instruction in the CPU or in the GPU queue. Additionally, it accesses the bank and bank the command is addressing and enqueues it also in the bank queuing structure. Finally, it assigns a packet ID to the incoming instructions that will be used in the Delete Procedure to access the correct data packet.

2. The Pop Procedure

   This macro function is the part of the algorithm in charge of providing a chosen memory command to the outer modules. It is called at each clock cycle and is the procedure triggering all the Scheduling logic. During the execution, many checks are performed in order to test which queue, if any, is empty and based on this analysis service packets (Refresh and Precharge) can be issued. The execution of the Pop Procedure, is different at any iteration according to the whole system status. For example, if a Refresh has been triggered, it is in charge of scanning the Bank queues systems to check if a packet accessing the currently opened row is enqueued. Otherwise, it performs its standard flow of execution and tries to find an issuable instruction through the whole queuing system, according to the policies described in the previous section. The Pop procedure is also in charge of checking whether an instruction is issuable or not according to some electrical and physical rules, like the maximum number a bank can be activated before a Precharge operation, or the time that has to pass between two consecutive Activate instructions.

3. The Update Procedure

   This procedure is triggered at each clock cycle and is used to perform a check on the whole system to review the whole status and initiate operations to balance the number of picked instructions from the modules. In order, the main activities of this operation are:

   - Updating the GPU countdown decreasing it by one.
   - Checking if a packet in the system has crossed a Safety Threshold and is stuck in the system due to a double Activate operation.
   - Checking the GPU countdown and the Number of instructions stored in the GPU: if the number is above the Threshold, it triggers a Relocation operation.
   - Cleaning the dirty queues after the Relocation Operation, in order to avoid the presence of copies of previously deleted memory requests.

4. The Remove Procedure

   The Remove operation is performed each time an instruction leaves the memory system. It has to provide two features:

   (a) If a Read or Write is issued and it is addressing an already opened row of a bank, then also the paired Activate command has to be removed.

   (b) It has to delete the issued packet from all the queues in the system. If a packet is picked up from the bank queues because of an issued Refresh, then the same packet has to be removed from the CPU or GPU queue. On the other hand, if the Scheduling algorithm is performing its default behavior, and a Refresh has not been issued, then the same packet and eventually the Activate command have to be removed from the bank queue structure.

5. Other Service Features

   While writing the whole Scheduling Algorithm, many functions and procedures have been written to provide lower level services to the main macro functions written above. Usually, these have been used to performs checks on the electrical level and determine if a proper packet is issuable. A standard methodology is already present in DRAMSim2, but those functions had to be modified to support the new Queuing System. Other basic instructions have been developed to check the origin of the packet, to create new Precharge packets when one Precharge operation had to be forced into the system and finally, due to the different nature of this work, to provide additional or modified simulation results according to the module's performance the user is interested in.

CHAPTER 5

SIMULATION RESULTS AND ANALYSIS

In this chapter, the results of the simulations performed to test the new Priority-Based Scheduling Algorithm are introduced and discussed. These results are compared with the ones obtained using a previously developed Scheduling Algorithm, the Staged Memory Scheduling Algorithm. Both strategies have been extensively discussed in the previous chapters and are here analyzed only from the results point of view. The first section introduces the metrics that will be used for the comparison of the two algorithms. A large set of these has been selected to cover all the possible different performance parameters. The following section is dedicated to the presentation of all the test cases and the explanation of the motivation behind their choice. Here the different benchmarks are introduced and the strategy of the testing process is described. The following two sections, are dedicated to the actual simulation results on two main machine environments. In the third, a machine made of 2 cores is used as platform for testing the algorithms, in the fourth, the tests are performed on a four cores machine. Finally, a summary of all the analysis developed from the results of each simulation concludes the chapter.

5.1 Metrics

In this section, the chosen metrics for the comparison of the Algorithms are introduced and discussed. When defining how to produce the simulation results, the goal was to provide a set of data able to cover all the characteristics that have a weight on a CPU-GPU environment. For this reasons, it was necessary to have metrics that would show the performance of the single computation units, as long as parameters that are related to the whole system.

The following is a list of the metrics and their description that will be used extensively in the following sections.

1. Instructions per Cycle (IPC)

   $$IPC = \frac{Number\_of\_Instructions}{Number\_of\_Cycles} \tag{5.1}$$

   This metric refers to the whole system and is one of the main parameters to describe how the whole simulation has been performed on the designed environment. Usually the simulations are launched for a fixed number of instructions, for this reason the numerator is usually fixed. The value of the denominator, on the other hand, is variable and depends on the Algorithm overall performance. The Number of Instructions is the sum of all the instructions committed by all the processing units of both CPU and GPU.

2. CPU Instructions per Cycle (CPUIPC)

   $$CPUIPC = \frac{Number\_of\_CPU\_Instructions}{Number\_of\_Cycles} \tag{5.2}$$

This metric is similar to the previous one but takes into account just the Instructions issued by a computation core loaded with a General Purpose Application. This information is used to acknowledge the improvements or worsening of the CPU execution speed using the different Scheduling Algorithms.

3. GPU Instructions per Cycle (GPUIPC)

$$GPUIPC = \frac{Number\_of\_GPU\_Instructions}{Number\_of\_Cycles} \tag{5.3}$$

This metric is similar to the previous one but takes into account just the instructions issued by a computation core loaded with a GPU related Application.

4. Speedup (S)

$$S = \frac{IPC\_PB}{IPC\_SMS} \tag{5.4}$$

$$S_{CPU} = \frac{IPC\_PB_{CPU}}{IPC\_SMS_{CPU}} \tag{5.5}$$

$$S_{GPU} = \frac{IPC\_PB_{GPU}}{IPC\_SMS_{GPU}} \tag{5.6}$$

The Speedup is a metric used to compare the different IPC values previously introduced between the two algorithms. The Speedup provides a direct information on the Priority-Based Algorithm behavior with respect to the reference algorithm.

5. Weighted Speedup (WS)

$$WS = \frac{\sum_{i=1}^{n} \frac{IPC\_PB_i}{IPC\_SMS_i}}{n}, n = Number\_of\_Workloads \tag{5.7}$$

This metric is a weighted average of all the ratios of the IPCs across all the workloads. It takes into account the whole simulation process and provides information about the general behavior of the Priority Based Algorithm across different environments. This is the most general metric of comparison because the assumptions are not considered and it refers to all the different testing scenarios.

6. GPU Respect of Deadline (GPURD)

This metric is introduced to measure how the possible decrease of performance on the GPU would affect the whole GPU execution performance in a real world scenario. The concept is to have a metric that estimates how much the GPU system would have to be latency tolerant in order to execute a GPU workload without missing a deadline. As execution step, a frame is considered, estimating that it takes 20 millions instructions to be processed. From the GPUIPC value, it is possible to evaluate the number of clock cycles needed by the Reference Algorithm, to complete the frame. Then, the error is computed using the same strategy on the priority based GPUIPC value and it represents the amount of delay that would be introduced to complete the frame evaluation.

$$GPURD = \frac{Clock_{PB} - Clock_{SMS}}{Clock_{SMS}} \times 100 \tag{5.8}$$

7. Number of Instructions Ratio (NIR)

$$NIR = \frac{Number\_of\_Memory\_Commands\_CPU}{Number\_of\_Memory\_Commands\_GPU} \tag{5.9}$$

This parameter is not used for comparison purposes, but it is an important information about how the memory commands issued by the computation units are distributed among CPU and GPU. This parameter will be used in relationship with the assumption that the Algorithm is designed for Scenarios where the number of CPU instruction is lower than the number of GPU issued instructions. And in order to know how close the simulation scenario is to the assumption, this value had to be introduced.

5.2 Test Cases

In order to test the algorithm, two machine configuration have been developed and used in the simulations.

1. A Two Cores Machine, in which one core is dedicated to the CPU workload and one core to the GPU workload.
2. A Four Cores Machine, in which each module is made of two cores, and for this reason we have two cores dedicated to the CPU workloads and two to the GPU workloads.

For each machine, eight test cases have been built using the SPEC2006 Benchmark Suite. Different benchmarks had to be used to provide a wide set of different scenarios. If the CPU and GPU are considered black boxes, from the memory controller point of view, the input is made of two flows of memory requests. Now, according on the different benchmarks loaded into the computation cores, the number of memory requests is different from one benchmark to another. For this reason, using different benchmarks leads to having different distributions of memory requests, which is necessary to study how the performances changes according to the different weights on the different modules. This is also related to the NIR parameter, which can be described as the difference, in terms of memory requests, of two sets of workloads.

Furthermore, in order to extend the test cases from the machines point of view, all the tests on the Four Cores machine had been performed on a single channel memory model and on a dual channel memory model.

TABLE III: TEST CASES PERFORMED ON THE TWO CORES MACHINE CONFIGURATION

Name	CPU Workload	GPU Workload	NIR
Test 1	bzip2	povray	11.82%
Test 2	gcc	leslie3D	17.53%
Test 3	calculix	sphinx3	202.12%
Test 4	bzip2	lbm	22.74%
Test 5	bzip	mcf	111.03%
Test 6	perlbench	gobmk	441.93%
Test 7	gcc	calculix	19.35%
Test 8	calculix	gcc	147.40%

The tables show the sets of benchmarks used to test the Priority Based and the Staged Memory Scheduling algorithms on the different machines. As highlighted by the NIR column, the goal was to provide a various enough set of different loads on the two computation cores.

TABLE IV: TEST CASES PERFORMED ON THE FOUR CORES MACHINE CONFIGURATION

Name	CPU_Core0 Workload	CPU_Core1 Workload	GPU_Core0 Workload	GPU_Core1 Workload	NIR
Test 1	bzip2	mfc	povray	lbm	100.01%
Test 2	astar	bzip2	perlbench	gobmk	135.75%
Test 3	calculix	bzip2	sphinx3	lbm	12.55%
Test 4	gcc	calculix	povray	lbm	13.46%
Test 5	gcc	perlbench	leslie3d	4264ref	41.82%
Test 6	calculix	gcc	perlbench	mcf	158.06%
Test 7	bzip2	gcc	gromacs	zeus-mp	104.38%
Test 8	povray	lbm	gcc	calculix	522.12%

5.3 Two-Core Machine Results

The attached charts are here introduced to present the results of the simulations performed on a Two Cores machine configuration.

As highlighted in the graph containing the whole system IPC results (Figure 14), in which the workloads have been sorted on the x-axis according to their respective NIR value, the Priority Based Algorithm shows an improvement in performance in all the workloads having the number of CPU instructions lower than the number of GPU instructions. As soon as the memory requests numbers are similar, and the NIR approaches values equals or greater than 1, the improvements decrease and the Priority Based Algorithm behavior is similar, in terms of performances, to the Staged Memory Scheduling Algorithm. Except for one single case where the performance, in terms of Whole System Speedup (Figure 15), are slightly worse (0.05%), even with unbalanced workloads the Priority Based Scheduling algorithm behaves better then

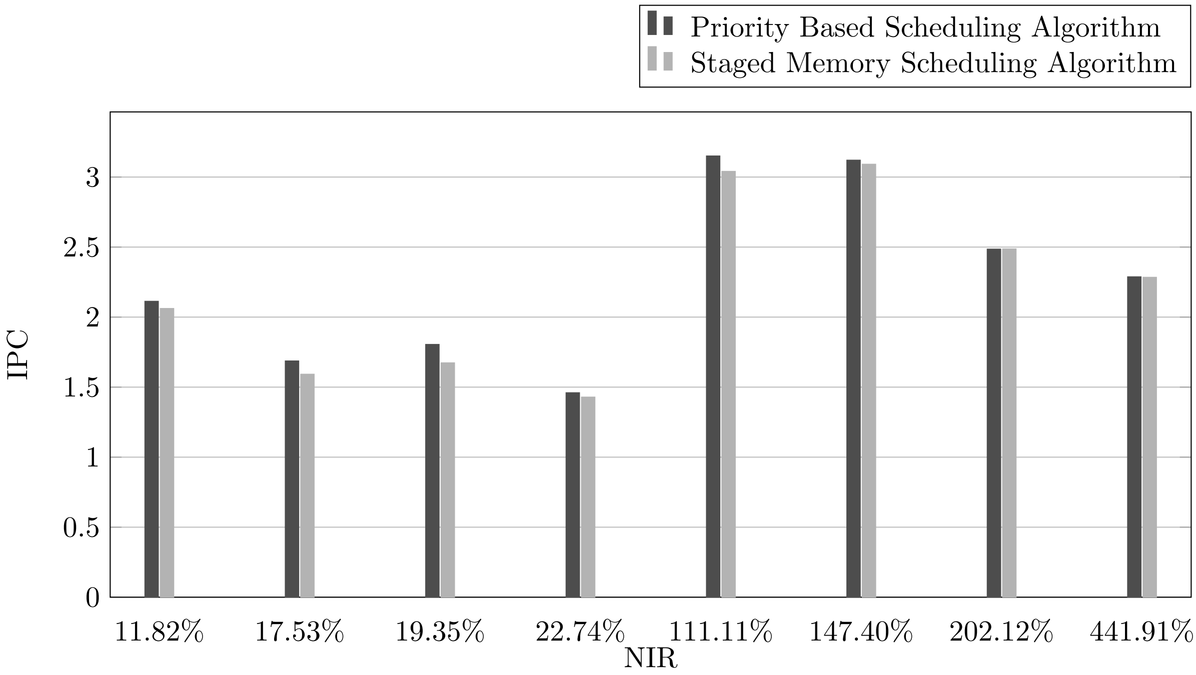


Figure 13: IPC, two cores machine configuration

the Reference Algorithm. These characteristics were expected and were part of the assumptions taken in consideration during the design process of the Algorithm. Usually, in fact, the CPU is dealing with a number of memory requests lower then the number of GPU memory requests, and the improvements show up when those conditions are met.

The overall performance, among all the tested workloads, can be represented by the Weighted Speedup:

$$WS = \frac{\sum_{i=1}^{n} \frac{IPC\_PB_i}{IPC\_SMS_i}}{n} = 2.8825\% \tag{5.10}$$

As shown in the graphs related to the IPC of the single modules, it is proven that, for the majority of the tests, the benefits are caused by an improvement of the CPU IPC values with respect of the GPU counterpart. On the other hand, the GPU performance in some cases are worse in the Priority Based Scheduling Algorithm. This was an expected behavior, since the GPU overall performance is less sensitive to latency then the CPU, and the overall Speedup is increased, the trade-off is advantageous. Furthermore, the concept of the Priority-Based Scheduling Algorithm is to prioritize the CPU execution over the GPU one, as long as the decrease of performance is tolerable.

The GPUDM computation for each workload brought to a 2.20% for test4 and 0.5% for test7. All the other benchmarks presented a slightly improvement or a negligible (<0.1%) decrease of performance. In these latest cases, the values are so small that can be considered simulation dependent. In general, the slight decrease of performances can be explained by the fact that, while simulating the algorithms on a machine made of two cores, the overall weight on the memory is moderate and the effect on GPUs is almost negligible.

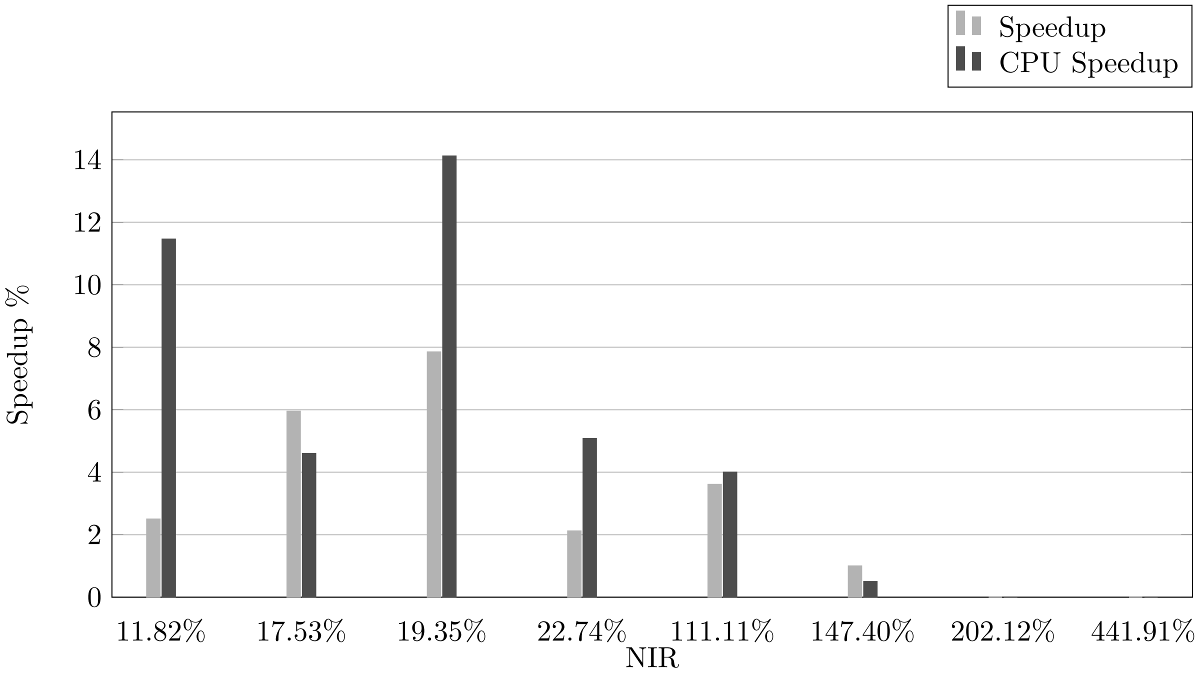


Figure 14: System and CPU speedup, two cores machine configuration

5.4 Four-Core Machine Results

In this section, the results of the Four Core Machine configuration are presented. The tests have been launched on two different configurations: the first is a single channel memory system, while the second design is provided of two channels. The simulations have been processed according to the benchmarks listed in the previous section (Table IV).

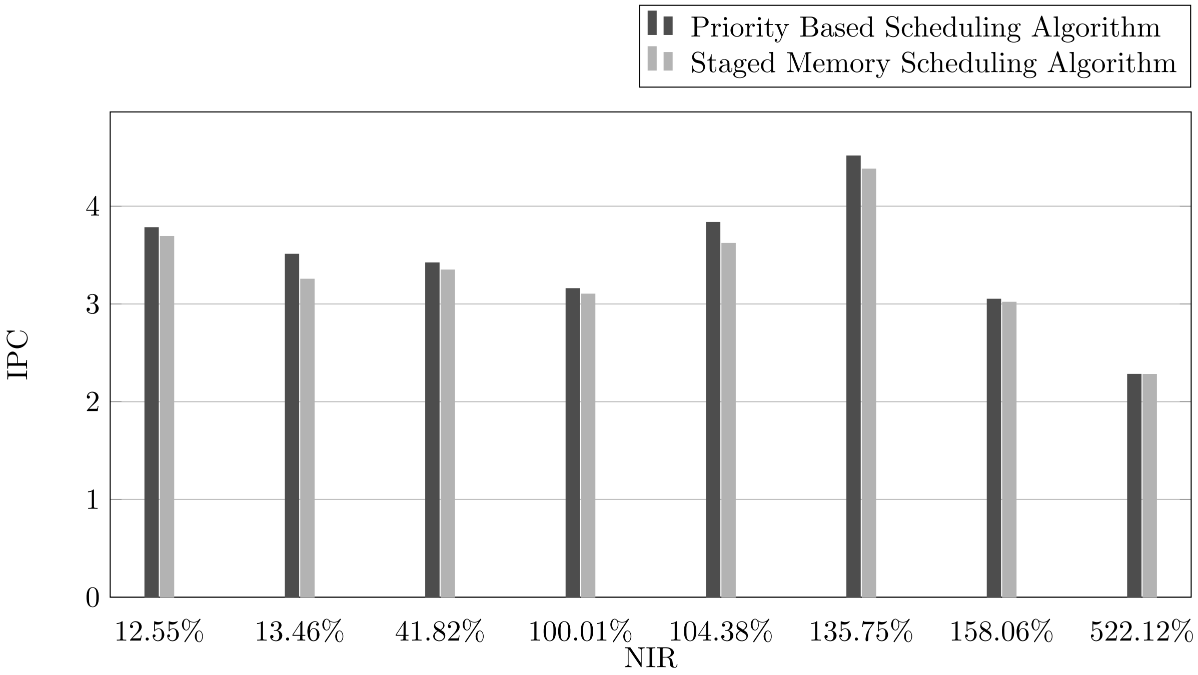


Figure 15: IPC, four cores machine configuration, one channel

5.4.1 One-Channel Results

As highlighted by the attached bar charts, the results show a behavior similar to the one observed in the case of the two cores machine configuration. The IPC chart (Figure 16) shows that the major improvements are noticeable in relationships with low NIR values. Once the assumption of having less CPU memory instructions than GPU is met, the Priority Based Algorithm is able to prioritize efficiently, introducing a low loss in terms of GPU timing performances, the CPU and to improve the whole system behavior. The IPC gain is stable until the NIR is around 1.0, then it decreases, making the Priority Based Algorithm's behavior similar

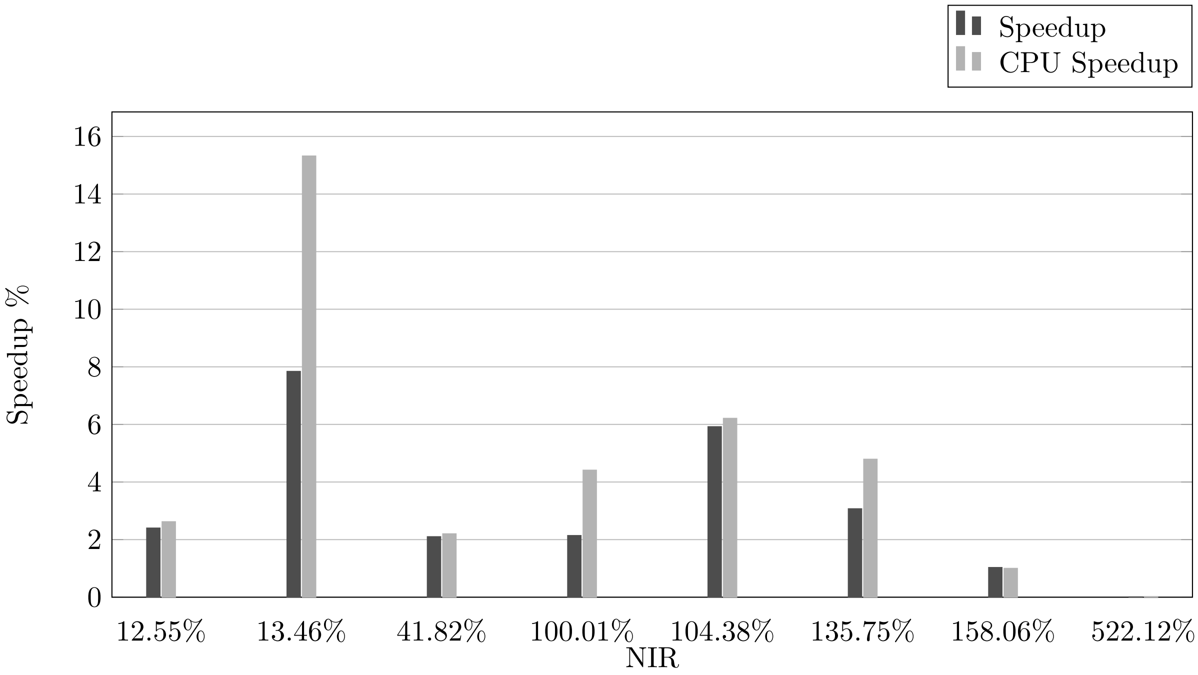


Figure 16: System and CPU speedup, four cores machine configuration, one channel

to the Staged Memory Scheduling Algorithm.

In terms of speedup (Figure 17), the whole systems performances are increased for a range that starts from 1.5% for high NIR values, to 8% for workloads with low NIR and not very high traffic load. Higher values can be found examining the Speedup of the CPU modules, which are the main cause of the general improvements on the whole machine. When the initial assumptions are not met anymore, the Speedup goes to zero and the Priority-Based Algorithm behaves like the Staged Memory Scheduling Algorithm. The overall performance of the Algorithm, among

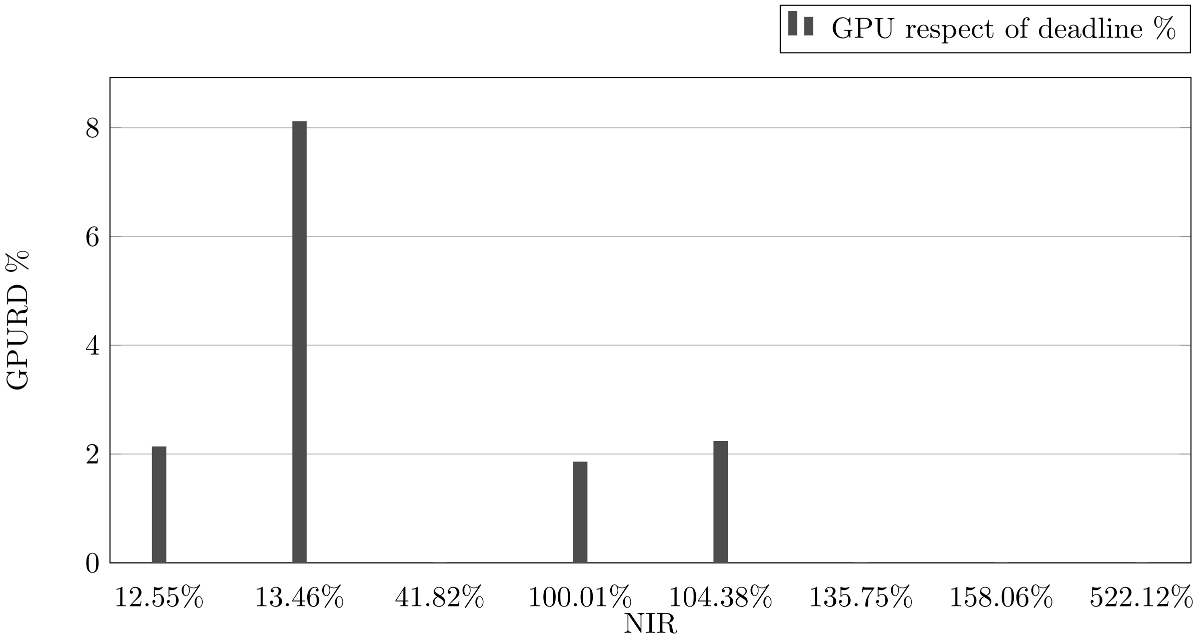


Figure 17: GPU respect of deadline, four cores machine configuration, one channel

different scenarios and workload environments is represented again by the Weighted Speedup value.

$$WS = \frac{\sum_{i=1}^{n} \frac{IPC\_PB_i}{IPC\_SMS_i}}{n} = 3.06\% \tag{5.11}$$

From this average value, it is possible to estimate a general benefit from the application of the Priority based Scheduling Algorithm. As highlighted in the bar chart presenting the GPURD results (Figure 18), the improvement on the whole system IPC ratio is provided with a decrease of GPU timing performances. In general, the tests that registered the best results in terms of improvement, are showing the highest values of delay in the GPU execution. The worst case has been found in test4, where in order to avoid deadline misses, the GPU application should tolerate a 8.11% latency delay on the GPU execution with respect to the Staged memory Scheduling algorithm. As a remainder, test4 represents also the best result, in terms of general Speedup, of the whole benchmark set. In addition, as highlighted by the same chart, the other benchmark ran with GPURD values ranging from 1.85% to 2.23%. As long as the NIR value increases and as described above the Speedup decreases, the GPURD value becomes negligible and the GPU behavior is similar to the one of the Staged Memory Scheduling algorithm.

5.4.2 Two-Channel Results

The following results had been extracted from simulations performed on a four core machine provided with a memory system made of two channels. The goal was to build a wider set of test cases using two different machine configurations. The attached bar charts show again the Algorithms' behavior with the metrics defined in the previous section. Once again, it is possible to appreciate the improvements generated from the Priority Based Algorithm, especially when dealing with low NIR values (Figure 19).

From the Speedup chart (Figure 20), it is possible to notice that an improvement is noticeable in all the tested workloads. In almost all the cases, the whole system improvement is paired with an higher improvement in the CPU IPC, except for test7 where the values are similar. The graphs shows again a behavior similar to the one expected: the CPU instructions are prioritized with respect to the reference algorithm and this improvement is translated into an increase of the overall IPC that leads to positive Speedup values, ranging from 2.1% to 7.5%.

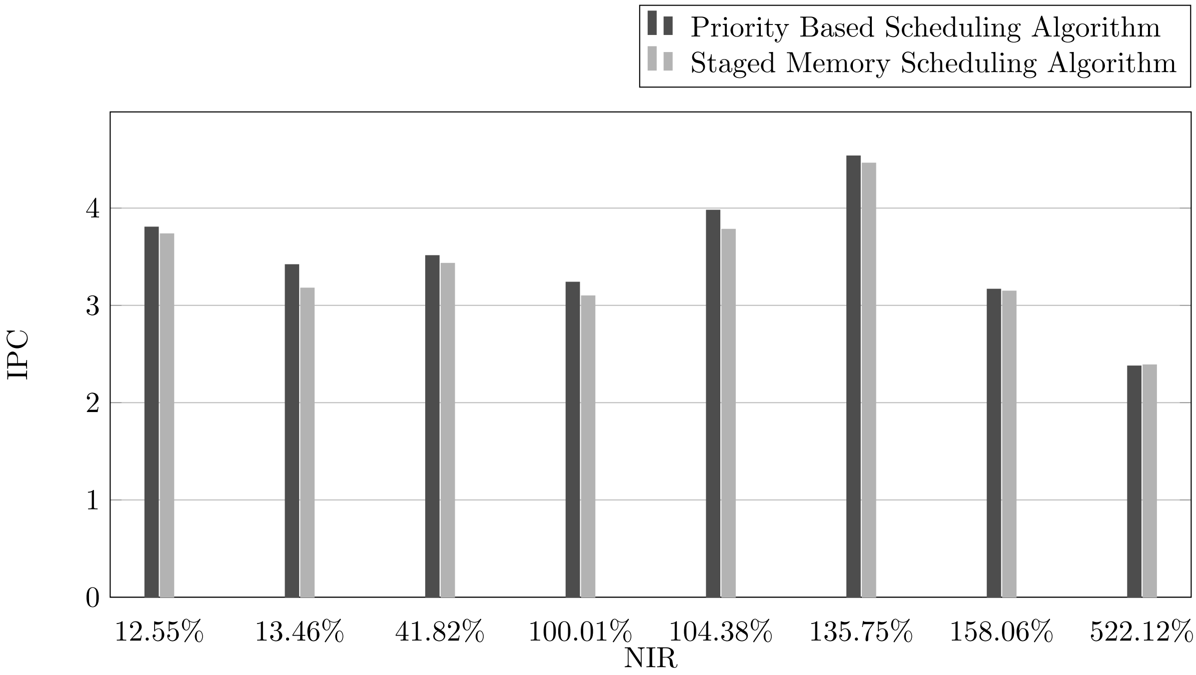


Figure 18: IPC, four cores machine configuration, two channels

The Weighted Speedup among all the tested workload and related to the whole system IPCs is the following:

$$WS = \frac{\sum_{i=1}^{n} \frac{IPC\_PB_i}{IPC\_SMS_i}}{n} = 3.49875\% \tag{5.12}$$

This test case shows, like in previous examples, that the benefits of the Priority Based Scheduling algorithm are more evident when dealing with NIR ¡ 1.5. When considering higher values, in fact, the behavior is similar as the one of the Staged Memory Scheduling algorithm. Regarding the GPU, the GPURD graph (Figure 21) shows a behavior similar to the one examined for the single channel machine configuration. While in many cases, marked with a zero in

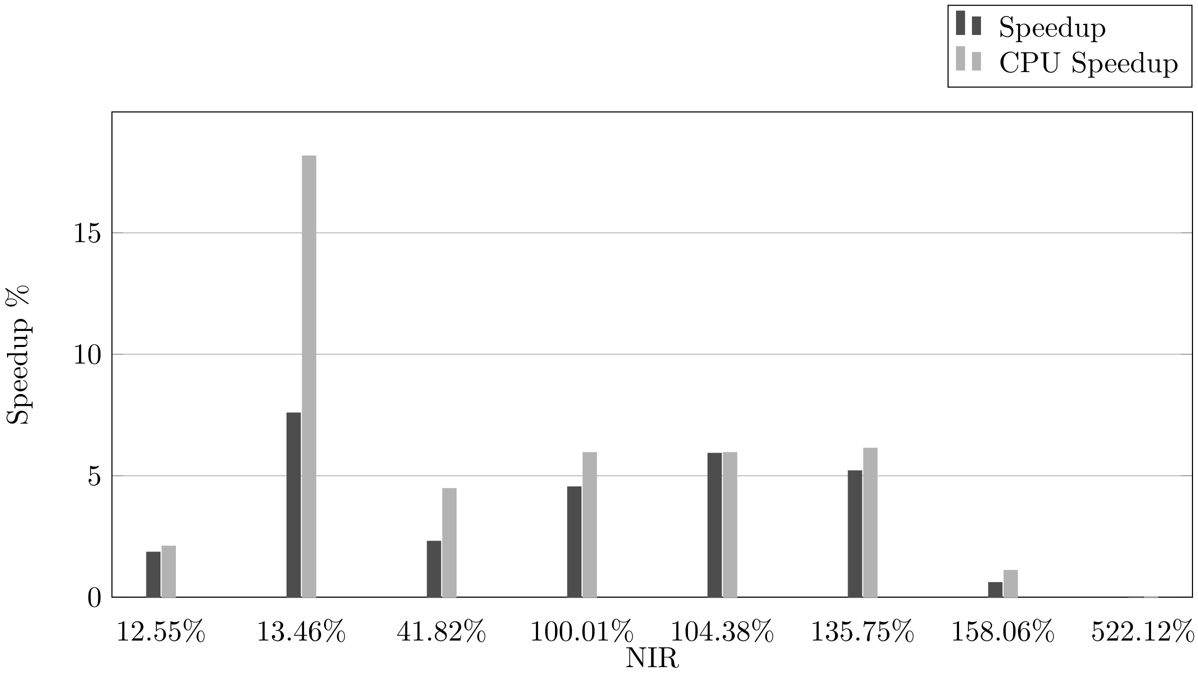


Figure 19: System and CPU speedup, four cores machine configuration, two channels

the chart, the percentage of delay introduced is negligible (¡0.1%), the worst case would require to the GPU module a delay tolerance of 9.08%. However, the average GPURD value among all the workloads is much lower, 2.01% considering the negligible cases and 4.01% otherwise. Again, the peak value is reported on test4, which is also the one that shows the best result in terms of overall Speedup and CPU Speedup.

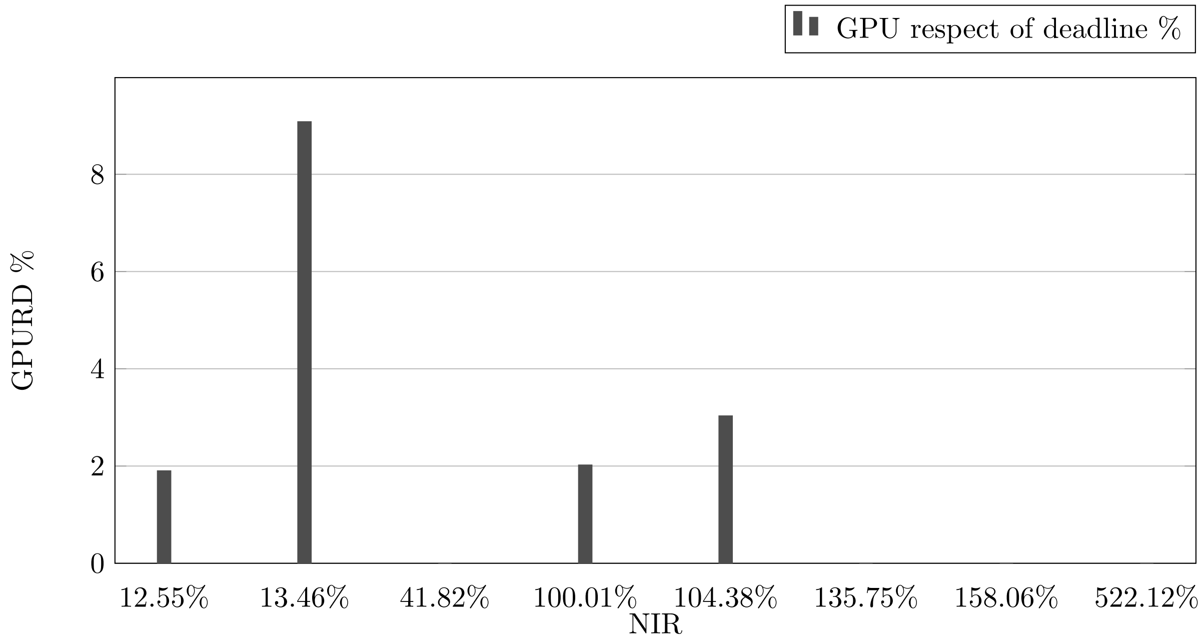


Figure 20: GPU respect of deadline, four cores machine configuration, two channels

5.5 Result Summary

The different tests presented and discussed above showed similar behaviors and common characteristics. In all the machine configurations, it was possible to acknowledge an improvement in the overall IPC and CPU IPC that led to positive Speedup values. While in almost any case the Priority-Based Scheduling Algorithm introduced positive Speedup values, it was possible to notice higher values when dealing with NIR ranges below the 1.5 threshold. This result was expected when designing the algorithm and, according to the considerations discussed in the previous chapters, the environment where the major benefits had been measured is the most common in a real world scenario, where usually the number of memory requests issued by the CPU is lower than the number of the same commands issued by the GPU. In addition, in all the cases the computed Weighted Speedup value showed positive results that demonstrate the fact that, even if not in the assumptions‘ NIR ranges, the Priority based scheduling algorithm performance is at least similar to the one evaluated with the Staged Memory Scheduling algorithm. While the latency sensitive CPU computation cores‘ performance had be improved to increase the overall IPC, in certain cases the GPU performance decreased. However, as shown with the chart related to the GPURD values, all the results had been lower than 10% and, in the average, the additional delay tolerance represented by the same metric would be below the 5.0% threshold. Additionally, in many of the tested workloads the tolerance value measured was negligible (below 0.1%). This worsening of the GPU performance was expected due to the additional priority assigned to the CPU computation cores, and according to the performance results, the decrease of these performance can be considered a positive trade-off with respect to the overall improvements.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

The presented research work's aim was to introduce a brand new Scheduling strategy for memory accesses in a CPU-GPU environment, exploiting a priority-based concept, dedicated architectural choices and a set of previously developed analysis‘, in order to improve the overall system performance. The first step was to shrink the working environment in order to focus the algorithm design process on scenarios that would match the characteristics and the needs of a real CPU-GPU execution session. This was achieved by studying the previous works developed on these architectures which, according to simulations and real time analysis‘, led to the definition of the assumptions introduced in the previous chapters and that had been the main reference for the Priority Based Scheduling Algorithm design. From these basis, the algorithm was developed through all the architectural required components, and implemented taking into account a wide set of relocation related problems and developing solutions to them. Finally, the algorithm had been extensively tested among different workloads and several comparisons with a previously developed Scheduling Algorithm had been performed to measure the actual improvements reported in the Results chapter. From the extracted data provided by the simulations, it is possible to observe that, when the previously stated assumptions are met, and the scheduling algorithm is working in an environment that presents common characteristics, the improvements are positive in almost all the tested cases and all the considered machines.

Even if the main features of the Scheduling Algorithm are all presented and working, additional features can be taken into account for future developments of the same strategy. Even if the original implementation already sets many parameters, like the triggering of the Relocation Procedure or the reset operation on the GPU countdown, according to the status of the whole system, in the future it would be possible to introduce additional mechanisms to make some features more dynamic and test them to acknowledge if they would bring additional improvements in terms of performance.

Finally, while the algorithm exploits the space and time localities introducing controls on the enqueued memory commands, additional adjustments on the original Queuing System architecture would improve the effectiveness of those two concepts from a lower level point of view. Again, it would be interesting to see how the resulting improvements, if any, would differ from the original implementation here introduced and implemented.

CITED LITERATURE

1. Ausavarungnirun, R., Chang, K. K.-W., Subramanian, L., Loh, G. H., and Mutlu, O.: Staged memory scheduling: achieving high performance and scalability in heterogeneous systems. ISCA ’12 Proceedings of the 39th Annual International Symposium on Computer Architecture, 40(3):416–427, June 2012.

2. Lu, S.-L. L.: Memory architectures. Technical Report 9780471346081, Wiley Online Library, 1999.

3. Passerone, C.: Electronics for Embedded Systems. Torino, Italia, CLUT, 2014.

4. Shaaban, M.: Memory architecture. Technical report, Rochester Institute of Technology, too1.

5. Justin Meza, J. L. and Mutlu, O.: A case for small row buffers in non-volatile main memories. 2012.

6. Song, Y. and Parihar, R.: Dram memory controller and optimizations. `http://www.ece.rochester.edu/ parihar/pres/Pres_DRAM-Scheduling.pdf, accessed in 2009.`

7. Brown, R. G.: Bottlenecks: Or...why can’t life be simple? `https://www.phy.duke.edu/ rgb/Beowulf/beowulf_book/beowulf_book/node24.html, accessed May 24, 2004.`

8. Feng, W.: Memory architectures. `http://web.mit.edu/6.173/www/currentsemester/readings/R04` `accessed May 24, 2004.`

9. Stenstrom, P.: A survey of cache coherence-schemes for multiprocessors. Technical report, Lund University, 1990. `http://web.mit.edu/6.173/www/currentsemester/readings/R04-cache-coherence-surve`

10. Azriel, L.: Peripheral memory: a technique for fighting memory bandwidth bottleneck. Computer Architecture Letters, PP(99):1, April 2014.

11. Zhu, D., Wang, R., Wang, H., Qian, D., Luan, Z., and Chu, T.: A fair thread-aware memory scheduling algorithm for chip multiprocessor. Algorithms and Architectures for Parallel Processing, 6081:174–185, May 2010.

12. Kim, Y., Papamichael, M., Mutlu, O., and Harchol-Balter, M.: Thread cluster memory scheduling: Exploiting differences in memory access behavior. MICRO '43 Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, pages 65–76, December 2010.

13. Ribble, M.: Next-gen tile-based gpus. `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media` `accessed Feb 18, 2008.`

14. Boudier, P.: Memory system on fusion apus. `http://developer.amd.com/wordpress/media/2013/06/` `accessed Jun 10, 2010.`

15. Spafford, K. L., Meredith, J. S., Lee, S., Li, D., Roth, P. C., and Vetter, J.: The tradeoffs of fused memory hierarchies in heterogeneous computing architectures. Conf. Computing Frontiers, pages 103–112, 2010.

16. Stevens, A.: Qos for high-performance and power-efficient hd multimedia. Technical report, ARM, April 2010.

17. Sangani, C., Venkatesan, M., and Ramesh, R.: Phase aware memory scheduling memory scheduling algorithms for integrated heterogeneous multicores. `http://web.stanford.edu/ rakeshr1/Downloads/cs316_final_report_PAMS.pdf,` `2010.`

18. Patel, A., Afram, F., Chen, S., and Ghose, K.: MARSSx86: A Full System Simulator for x86 CPUs. In Design Automation Conference 2011 (DAC'11), 2011.

19. Rosenfeld, P., Cooper-Balis, E., and Jacob, B.: DRAMSim2: A Cycle Accurate Memory System Simulator. In IEEE computer architecture letters, 2011.

20. Jeong, M. K., Erez, M., Sudanthi, C., and Paver, N.: A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc. Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE, pages 850 – 855, June 2012.

21. University, C. M.: Multi core scheduling. `http://www.sei.cmu.edu/cyber-physical/research/timi` `2012.`

VITA

NAME: Fabio Ghiozzi

EDUCATION: Laurea di I livello in Ingegneria Informatica, Politecnico di Torino, 2013.

Laurea Specialistica in Computer Engineering, Politecnico di Torino, 2015.

Master of Science in Electrical and Computer Engineering, University of Illinois at Chicago, 2015

HONORS: Mobility Scholarship, Politecnico di Torino, August 2014

Master Thesis Abroad Scholarship, Politecnico di Torino, October 2014