UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Efficient L2 Cache Management to Boost GPGPU Performance

July 2019

Author: Francisco Candel Margaix

Advisors: Salvador Petit Martí

Julio Sahuquillo Borrás

Abstract

In recent years, the growing need for computing capacity has become a challenge that has led the industry to look for alternative architectures to conventional out-of-order superscalar processors, with the goal of enabling an increase of computing power while achieving higher energy efficiency.

GPU architectures, which just a decade ago were applied to accelerate computer graphics exclusively, have been one of the most employed alternatives for several years to reach the mentioned goal. A particular characteristic of GPUs is their high main memory bandwidth, which allows executing a large number of threads in a very efficient way. This feature, as well as their high computational power regarding floating-point operations, have caused the emergence of the *GPGPU computing* paradigm, where GPU architectures perform general purpose computations. The aforementioned characteristics make GPU devices very appropriate for the execution of massively parallel applications that have been traditionally executed in conventional high-performance processors.

The work performed in this thesis aims to help improve the performance of GPUs in the execution of GPGPU applications. To this end, as a first step, a characterization study is carried out. In this study, the most important features of GPGPU applications, with respect to the memory hierarchy and its impact on performance, are identified. For this purpose, a detailed cycle-accurate simulator is used to model the architecture of a recent GPU. The study reveals that it is necessary to model with more detail some critical components of the GPU memory hierarchy in order to obtain accurate results. In addition, it shows that the achieved benefits can vary up to a factor of 3× depending on how these critical components are modeled.

Due to this reason, as a second step before realizing a novel proposal, the work in this thesis focuses on determining which components of the GPU memory hierarchy must be modeled with more detail to increase the accuracy of simulator results and improving the existing simulator models of these components. Moreover, a validation study is performed comparing the results obtained with the improved GPU models against those from a real commercial GPU. The implemented simulator improvements reduce the deviation of the results obtained with the simulator from results obtained with the real GPU by about 96%.

Finally, once simulation accuracy is increased, this thesis proposes a novel approach, called FRC (*Fetch and Replacement Cache*), which highly improves the GPU computational power by enhancing main memory-level parallelism. The proposal increases the number of parallel accesses to main memory by accelerating the management of fetch and replacement actions corresponding to those cache accesses that miss in the cache. The FRC approach is based on a small auxiliary cache structure that efficiently unclogs the memory subsystem, enhancing the GPU performance up to 118% on average compared to the studied baseline. In addition, the FRC approach reduces the energy consumption of the memory hierarchy by a 57%.

Resumen

En los últimos años, la creciente necesidad de la capacidad de cómputo ha supuesto un reto que ha llevado a la industria a buscar arquitecturas alternativas a los procesadores superescalares con ejecución fuera de orden convencionales, con el objetivo de incrementar la potencia de cómputo con una mayor eficiencia energética.

Las GPU, que hasta hace apenas una década se dedicaban exclusivamente a la aceleración de los gráficos en los computadores, han sido una de las arquitecturas alternativas más utilizadas durante varios años para alcanzar el mencionado objetivo. Una de las características particulares de las GPU es su gran ancho de banda para acceder a memoria principal, lo que les permite ejecutar un gran número de hilos de forma muy eficiente. Esta característica, así como su elevada potencia computacional ejecutando operaciones de coma flotante, ha originado la aparición del paradigma de computación denominado *GPGPU computing*, paradigma en el que las GPU realizan cómputo de propósito general. Las citadas características convierten a las GPU en dispositivos especialmente apropiados para la ejecución de aplicaciones masivamente paralelas que tradicionalmente se habían ejecutado en procesadores convencionales de altas prestaciones.

El trabajo desarrollado en esta tesis persigue ayudar a mejorar las prestaciones de las GPU en la ejecución de aplicaciones GPGPU. Con este fin, como primer paso, se realiza un estudio de caracterización donde se identifican las características más importantes de estas aplicaciones desde el punto de vista de la jerarquía de memoria y su impacto en las prestaciones. Para ello, se utiliza un simulador detallado ciclo a ciclo donde se modela la arquitectura de una GPU reciente. El estudio revela que es necesario modelar de forma más detallada algunos componentes críticos de la jerarquía de memoria de las GPU para obtener resultados precisos.

Los resultados obtenidos muestran que las prestaciones alcanzadas pueden variar hasta en un factor de 3× dependiendo de cómo se modelen estos componentes críticos.

Por este motivo, como segundo paso antes de elaborar la propuesta de mejora, el trabajo se centra en determinar qué componentes de la jerarquía de memoria de la GPU necesitan modelarse con mayor detalle para mejorar la precisión de los resultados del simulador, y en mejorar los modelos existentes de estos componentes. Además, se realiza un estudio de validación que compara los resultados obtenidos con los modelos mejorados contra los de una GPU comercial real. Las mejoras implementadas reducen la desviación de los resultados del simulador sobre los resultados reales alrededor de un 96%.

Finalmente, una vez mejorada la precisión del simulador, en esta tesis se presenta una propuesta innovadora, denominada FRC (siglas en inglés de *Fetch and Replacement Cache*), que mejora en gran medida la potencia computacional de la GPU, gracias a que aumenta el paralelismo en el acceso a memoria principal. La propuesta incrementa el número de accesos en paralelo a memoria principal mediante la aceleración de la gestión de las acciones de búsqueda y reemplazo relacionadas con los accesos que fallan en la cache. La propuesta FRC se basa en una pequeña estructura cache auxiliar que descongestiona el subsistema de memoria eficientemente, aumentando las prestaciones de la GPU hasta un 118% de media respecto al sistema base. Además, también reduce en 57% el consumo energético de la jerarquía de memoria.

Resum

En els últims anys, la creixent necessitat de capacitat de còmput ha suposat un repte que ha portat a la indústria a buscar arquitectures alternatives als processadors superescalars amb execució fora d'ordre convencionals, amb l'objectiu d'incrementar la potència de còmput alhora que s'aconsegueix una major eficiència energètica.

Les arquitectures GPU, les quals fins fa només una dècada es dedicaven exclusivament a l'acceleració dels gràfics en els computadors, han sigut una de les alternatives més utilitzades durant alguns anys per a aconseguir l'esmentat objectiu. Una de les característiques particulars de les GPU és el seu elevat ample de banda per a accedir a memòria principal, la qual cosa permet executar un gran nombre de fils de forma molt eficient. Aquesta característica, així com la seua elevada potència computacional executant operacions de coma flotant, ha originat l'aparició del paradigma de computació anomenat *GPGPU computing*, paradigma on les GPU realitzen còmput de propòsit general. Les citades característiques converteixen a les GPU en dispositius especialment apropiats per a l'execució d'aplicacions massivament paral·leles que tradicionalment s'havien executat en processadors convencionals d'altes prestacions.

El treball desenvolupat en aquesta tesi persegueix ajudar a millorar les prestacions de les GPU en l'execució de les aplicacions GPGPU. A aquest efecte, com a primer pas, es realitza un estudi de caracterització on s'identifiquen les característiques més importants d'aquestes aplicacions des del punt de vista de la jerarquia de memòria i el seu impacte en les prestacions. Per a això s'utilitza un simulador detallat cicle a cicle on es modela l'arquitectura d'una GPU recent. L'estudi revela que és necessari modelar de forma més detallada alguns components crítics de la jerarquia de memòria de les GPU per a obtindre resultats precisos. Els resultats obtinguts mostren que les prestacions aconseguides poden variar fins i tot en un factor de 3× depenent de com es modelen aquests components crítics.

Per aquest motiu, com a segon pas abans d'elaborar la proposta de millora, el treball se centra en determinar quins components de la jerarquia de memòria de la GPU necessiten modelar-se amb major detall per a millorar la precisió dels resultats del simulador i en millorar els models existents d'aquests components. A més, es realitza un estudi de validació que compara els resultats obtinguts amb els models millorats contra els d'una GPU comercial real. Les millores implementades redueixen la desviació dels resultats del simulador sobre els resultats reals al voltant d'un 96%.

Finalment, una vegada millorada la precisió del simulador, en aquesta tesi es presenta una proposta innovadora, denominada FRC (sigles en anglés de *Fetch and Replacement Cache*), que millora en gran manera la potència computacional de la GPU, gràcies a que augmenta el paral·lelisme en l'accés a memòria principal. La proposta incrementa el nombre d'accessos en paral·lel a memòria principal mitjançant l'acceleració de la gestió de les accions de recerca i reemplaçament relacionades amb els accessos que fallen en la cache. La proposta FRC es basa en una xicoteta estructura cache auxiliar que descongestiona el subsistema de memòria eficientment, augmentant les prestacions de la GPU fins a un 118% de mitjana respecte al sistema base. A més, també redueix, al voltant d'un 57%, el consum energètic de la jerarquia de memòria.

Contents

Chapter 1

Introduction

This chapter introduces both basic and fundamental concepts to help understand this dissertation and presents the motivation for the work developed in this thesis.

1.1 Motivation

In the last decade, GPU (Graphics Processing Unit) architectures have acquired a great relevance in both high-performance computing and heterogeneous computing. The main reason of this increasing relevance is that GPUs are much more energy efficient than CPUs [27, 29], since they provide a much higher thread-level parallelism and a better performance to power ratio. As a consequence, many of the most powerful and energy-efficient supercomputers in the world, ranked in both Top500 and Green500 lists [71], rely on GPUs.

The huge computational power that GPUs can provide mainly comes from their ultra-parallel architecture. They are composed of around one hundred of Single Instruction Multiple Data (SIMD) units that work in parallel, while a single SIMD unit can execute the same instruction for tens of different threads in the same clock cycle. This architecture allows GPU applications to be characterized by their massive parallelism, and they are composed of thousands of logical threads.

However, to feed all the SIMD units with enough data, memory accesses need to be handled in a fast manner; otherwise these units would suffer starvation, preventing the GPU from achieving its peak performance. This means that the memory hierarchy plays a key role in GPU performance and it must be designed to provide much higher bandwidth than conventional multi-core CPU memory subsystems. Based on these requirements, the memory hierarchy of a GPU is not designed to reduce the latency of individual accesses but to support a huge number of concurrent accesses. In this way, the high thread-level parallelism allows to hide the major part of the main memory latency.

GPUs were originally built in the 70s as specialized hardware to accelerate graphic processing. The first designs could not be programmed and their functionality was implemented directly in the hardware. With the evolution of GPU designs this constraint was removed and, in the 2000s, the first cards with programmable shaders were presented, that is, small programs that run for every pixel that is rendered on the screen. Not much later, manufacturers introduced support that enabled the use of these programmable GPUs not only for graphics but for accelerating the processing of matrices or vectors, giving birth to the General Purpose computing on Graphics Processing Units (GPGPU) programming paradigm.

Nowadays, GPGPU is a major computing paradigm, whose requirements drive the development of current and future GPU devices. Nevertheless, programming high-performance GPGPU applications widely differs from programming CPU applications because the application work must be distributed among the highest possible number of threads. Unfortunately, the high level of parallelism increases programming complexity and requires from efficient architectures to support it. To deal with this fact, and to help programmers to accelerate the development of fast GPGPU applications, the industry is both facilitating GPU programmability and raising the computational power of GPU devices by adapting different architectural mechanisms, like cache memories or prefetchers, which have worked successfully in CPUs. Even including these system components, GPU memory subsystems present serious performance bottlenecks caused by the huge level of parallelism. To palliate this problem, as GPU architectures evolve they include larger on-chip memory subsystems that allow improving the Memory-Level Parallelism (MLP) and so the system performance. For instance, Nvidia has systematically enlarged the Last-Level Cache (LLC) size in 2MB on consecutive recent architectures (e.g., LLC sizes of Maxwell [53], Pascal [54], and Volta [55] GPUs are 2MB, 4MB, and 6MB, respectively).

The importance of facilitating the programmability of GPUs and increasing their memory subsystem performance is currently driving the research on the memory hierarchy of GPUs. In order to design and evaluate new memory subsystem approaches for GPUs, researchers often use complex simulation environments because they are more affordable and easier to implement than real hardware. In addition, this software allows researchers to focus only on those components having a significant impact on the system performance, while paying less or no attention to the implementation details of other, less significant, components. However, the continuous evolution of real GPU architectures makes simulators to quickly loose accuracy and becoming less representative with respect to the real hardware of last GPU generations. Therefore, simulation software needs to be revised and updated from time to time to ensure that the simulation results remain valid. In particular, the modeling of components of new architectures critical for performance needs to be accurately covered.

This thesis pursues to improve the GPU performance by acting on the GPU memory hierarchy. To this end, several steps have been carried out progressively. First, characterization study has been performed to provide a sound understanding on the main memory subsystem bottlenecks that impact on the performance of GPU applications. The main aim of this characterization study is to help improving the memory subsystem of modern GPUs. This study reveals identifies some critical memory subsystem components that need to be accurately modeled in recent simulators in order to provide solid and representative proposals. These components have been modeled in a state-of-the-art simulator and its validated by comparing them with those of a recent commercial GPU. This validation is required because, to the best of our knowledge, there is not any official published information about the GPU memory subsystem from major GPU manufacturers, such as AMD or Nvidia, that deals with GPU memory hierarchy microarchitectural details (e.g. miss management). Finally, a new approach that efficiently improves the performance of the GPU memory subsystem, and thus the overall GPU throughput, has been proposed. The proposal improves the management of the LLC cache misses, increasing both the MLP and the hit ratio. Moreover, it scales well both in terms of performance and energy consumption with larger GPUs. Note that although this thesis is focused on memory hierarchies from AMD GPUs, some of its contents are also useful for GPU caches and coherence protocols based on academic proposals like NMOESI.

1.2 Background

This section provides some background on the GPGPU programming model, the GPU architecture and memory subsystem, and the simulation framework used in this thesis.

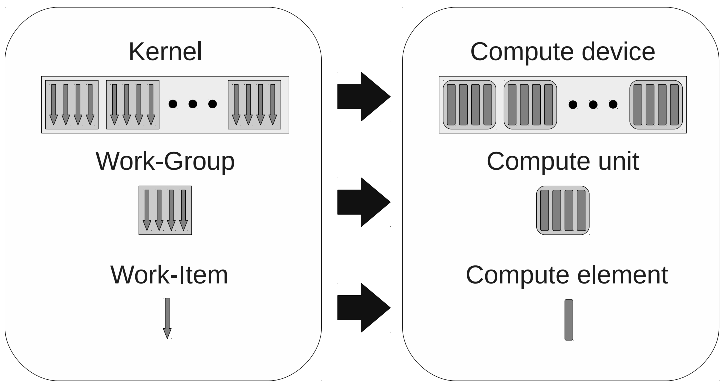


Figure 1.1: OpenCL platform and execution models.

1.2.1 OpenCL GPGPU programming model

There are two main GPGPU programming platforms and models, CUDA [52] from Nvidia and OpenCL [36] from the Khronos group. While CUDA is only supported by GPUs designed by Nvidia, OpenCL is, *de facto*, an industry standard programming model [66] and it is supported on devices from different brands such as Intel, AMD, ARM, or even Nvidia itself.

OpenCL defines a platform model, which is an abstraction of a real system where kernels are executed. The model comprises a hierarchy of *Compute Devices* (CDs), *Compute Units* (CUs), and *Processing Elements* (PEs), which refer to the GPU devices in the system, multi-core units inside GPU devices, and cores that execute scalar operations, respectively. The platform model maps the execution model, where the executing threads are also organized in a hierarchical manner. Each individual thread, which is executed in a single PE, is defined as a *work-item*. Work-items are grouped in *work-groups*, which are mapped to CUs. Finally, a kernel executing in a GPU is composed of several work-groups. Figure 1.1 depicts a block diagram of both models and how they map each other.

1.2.2 Graphics Core Next microarchitecture

The experimental work developed on this dissertation has considered several GPU architectures from AMD: Southern Islands [68], Arctic-Islands [6, 5], and the most recent Vega [9, 7]. These GPU architectures implement different versions of the same Graphics Core Next (GCN) CU microarchitecture, introduced in 2012 by AMD. The microarchitecture has evolved in the last few years across these versions; for instance, the working frequency has steadily increased from 1GHz in Southern Islands up to 1.5GHz in Vega, but the basic GCN design have remained almost stable.

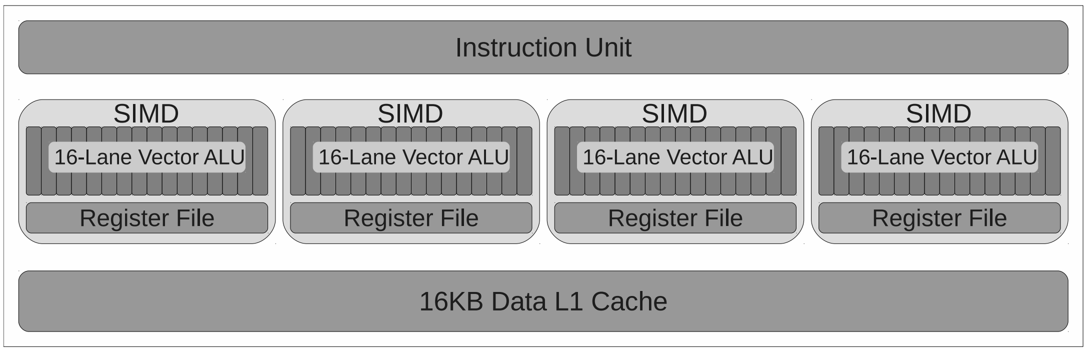


Figure 1.2: Graphics Core Next compute unit.

Figure 1.2 presents a block diagram of the GCN CU microarchitecture. A GCN CU consists of four 16-lane SIMD units; thus, it is capable of executing 64 work-items at the same time. In addition, a GCN CU includes several load/store units and a scalar unit to process scalar data.

Each kernel work-group is assigned to a specific SIMD unit, where it is executed. To be executed, the work-group is divided in 64-thread bundles, named *wavefronts*, consisting of 64 work-items. In turn, these wavefronts are subdivided in 4 sets composed of 16 work-items (also known as *subwavefronts*).

The instructions from the 64 work-items of a wavefront are executed in a lockstep manner. To do that, the SIMD unit executes sequentially the same instruction for the corresponding 4 subwavefronts. Therefore, the instruction (i.e. the entire wavefront) takes 4 cycles (i.e. one cycle per subwavefront) to execute.

To increase resource utilization and improve throughput, the GPU scheduler ensures that each SIMD unit is assigned tens of wavefronts, during most of the execution time of a kernel. SIMD units switch among wavefronts in a fine-grained multithreading manner, which helps hide the memory latencies.

1.2.3 Memory subsystem

Memory reference instructions are also executed following the SIMD paradigm; that is, a wavefront can generate up to 64 memory requests per memory reference. To reduce the overall amount of cache accesses, those requests addressing the same 64-byte cache block are *coalesced* into a single cache access, which is issued to the memory subsystem.

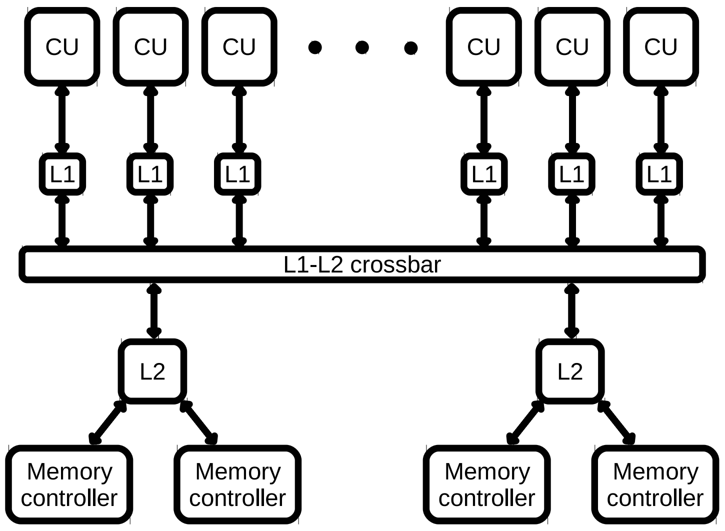


Figure 1.3: Memory hierarchy.

As in a conventional processor, the memory subsystem is organized hierarchically (see Figure 1.3). Those requests that miss the L1 cache are forwarded through an all-to-all crossbar switch to a multi-banked L2 cache, which acts as the LLC. Cache block addresses are interleaved among among L2 banks at a granularity of 256 bytes.

In the Southern Islands and Arctic Islands architectures, each bank is connected to a dual-channel memory controller that manages the corresponding off-chip GDDR5 main memory, while in the Vega architecture, the GDDR5 main memory modules are replaced with two stacks of the second version of the High Bandwidth Memory (HBM2) [31]. HBM2 is a a high-performance RAM interface for 3D-stacked DRAM from Samsung, AMD, and Hynix. This standard allows stacking up to 8 DRAM dies, each one with its own independent memory channel. In other words, each L2 bank in Vega architecture is directly connected to a single memory channel. This design reduces the number of channel conflicts and increases the memory bandwidth utilization.

1.2.4 Critical memory subsystem components

As mentioned above, in this thesis we identified critical memory subsystem components that have a significant impact on the system performance and whose accuracy is not properly modeled in state-of-the-art simulators. This section briefly discusses these components as background to help understanding the contribution.

Vector memory instruction buffer

The GCN microarchitecture implements the Vector Memory-instruction Buffer (VMB) in the CU. This buffer keeps track of the memory instructions issued to the cache until all their associated memory operations finish. Using the VMB and other structures, GPU architectures implement mechanisms that group memory requests of the same type (load or store) targeting the same cache line in a single memory access, so reducing the effective number of memory accesses. This way greatly reduces the pressure on the memory hierarchy.

Memory request coalescing and merging mechanisms

Two main approaches, namely *coalescing* and *merging*, can be found in modern GPUs to reduce the number of memory accesses. The coalescing approach combines all the requests of the same instruction targeting the same cache block into a single cache access in the CU just before issuing the instruction to the memory subsystem. For instance, the AMD Evergreen [2, 49] implements coalescing of both load and store instructions.

In contrast, the merging approach is implemented in the memory subsystem, decoupled from the CU. Merging is more flexible since it can applied to multiple memory requests regardless of whether they have been generated by the same memory instruction or not. Nevertheless, its use must be restricted to deal with memory coherence and memory consistency issues.

MSHR file

GPUs generate a huge quantity of memory accesses, but only a limited number of pending cache requests can be supported simultaneously. For this purpose, current non-blocking caches implement Miss Status Holding Registers (MSHR) files. Upon a cache miss, the MSHR file is looked up to check if the target block is already being fetched. On such a case, the missing memory access is queued into the MSHR entry associated to the target block.

A single MSHR entry is in charge of tracking all the memory accesses associated to a given cache block (i.e., all the requests whose target address falls within the same block). Therefore, the maximum number of outstanding memory accesses is limited to the number of MSHR entries. Consequently, if all MSHR entries are busy and the missing cache block is not being fetched, the memory access is stalled until an MSHR entry is released.

GPU cache coherence protocols

Cache coherence protocols were originally designed to support data coherence among caches in CPU multiprocessors. These protocols tolerate a moderate traffic of coherence requests, however, they are rather complex and would strangle the performance if they were directly applied to GPUs, mainly due to GPUs are designed to support a massive amount of memory requests generated by typical GPU applications. In short, neither GPUs nor heterogeneous CPU-GPU systems work properly with typical CPU protocols. To deal with this fact, alternative protocols have been devised both by the academia and the industry.

NMOESI coherence protocol. To support GPU cache coherence, Multi2Sim implements NMOESI, that extends the well-known MOESI protocol [67] implemented in a wide range of CPU multicores. NMOESI extends this protocol to support memory coherence in both CPU and GPU applications, and it is especially suited for heterogeneous CPU-GPU systems with a cache hierarchy shared among CUs and CPU cores. Under MOESI, a given cache block can be in one of five main states (M, O, E, S and I). NMOESI extends this protocol by adding a new non-coherent state (N) to be used in GPUs. This state avoids that non-coherent write requests, which are common in GPU applications, generate coherence traffic.

SI protocol. The protocol deployed in the Southern Islands (SI) GPU family, hereafter SI protocol, supports a relaxed memory consistency model based on Release Consistency [26]. This consistency model allows the compiler to specify when data modifications performed by a given CU must be visible to other CUs, which enables the implementation of simpler coherence protocols. To support the consistency model, the opcode of a SI memory instruction includes 2 bits called GLC (Global Coherent) and SLC (System Level Coherent), which indicate the coherency scope.

Memory controller

As conventional DDR SDRAM memories, Graphics DDR (GDDR) memory contain multiple independent DRAM banks. A bank is implemented as a matrix of DRAM cells. When a bank is accessed, the entire row also referred to as *memory page*, is accessed. The accessed memory page is stored in the DRAM sense amplifiers associated to the bank, also referred to as *row buffer*.

The memory controller uses three commands that are issued sequentially to a bank in order to access the target data [38]. First, the *precharge* command writes back the row content to the bank, and then precharges the row bitlines for accessing the target row. Second, the *activate* command reads the target row and stores its information into the row buffer. Finally, the *read/write* command reads or write the requested data in the row buffer. After issuing the last command, the memory controller can either keep the accessed memory page in the row buffer (open page policy) or close the row buffer by issuing a precharge command (closed page policy). Depending on the implemented page policy, the latency of the next access varies. For example, with an open page policy, if the requested block is already present in the row buffer (i.e., a row buffer hit), only a read/write command needs to be issued by the memory controller, thus the latency can be significantly reduced. However, a row buffer miss would require to serialize the issuing of the three mentioned commands, roughly trebling the latency of a row buffer hit.

Off-chip GDDR memory

The memory bus is used to read from or write to the memory device. In conventional DDR memories, the memory bus 64-bit width, while in GDDR memories this width is typically 32 bits. Since the typical cache block size is 64 bytes, transferring a cache block through the data bus doubles the number of transfers over DDR memories.

An option to reduce the total transfer time would be the use of a wider memory bus. Since GDDR devices are standardized to a 32-bit bus, working with a wider memory bus would require multiple memory devices to operate in lockstep. For example, the Intel i875P memory controller connects through a 128-bit memory data bus to matching pairs of 64-bit wide DIMMs (Dual In-Line Memory Modules). This paired DIMM configuration is often referred to as dual channel configuration [30].

1.2.5 GPU simulators

In comparison with CPU research simulators, the number of available GPU simulators is much lower. Moreover, existing GPU simulators are relatively recent and still maturing. The main causes of this situation are the lack of documentation provided by GPU manufacturers and the fast evolution of GPU architectures, which complicates the development of GPU simulators, since it requires stable and well-known architecture models.

Nevertheless, due to the growing use of GPUs, some GPU simulation frameworks have become recently available. Among them, it is worth mentioning GPGPU-Sim [23, 11] and Barra [22]. GPGPU-Sim is currently one of the most referenced GPU simulators and models a GPU microarchitecture that resembles the Nvidia GeForce 8x, 9x, and Fermi series. However, due to its dependence on Nvidia drivers, which only support OpenCL 1.1, GPGPU-Sim does not provide support for the execution of GPGPU benchmark suites like those provided by AMD [3] with modern OpenCL code. On the other hand, Barra is a parallel GPU simulator that implements both a CUDA driver emulator and an Nvidia Tesla GPU simulator. Unfortunately, Barra does not model the GPU microarchitecture, thus it cannot be used for the purposes of this thesis, which requires the evaluation of possible enhancements in the memory subsystem.

In this thesis, the Multi2Sim [73, 72] simulation framework has been selected as the main experimental platform. Multi2Sim is an accurate cycle-by-cycle execution-driven simulator for CPU-GPU heterogeneous computing. Release and development versions of Multi2Sim are available. It provides a fully configurable memory subsystem with several cache levels and interconnection network. Multi2Sim implements several GPU architectures from both AMD (e.g., Evergreen and Southern Islands) and Nvidia (e.g., Fermi) as well as CPU architectures like x86, MIPS-32 and ARM. The Multi2Sim developer team is currently modeling the HSA heterogeneous architecture [1], where CPU and GPU share the same memory subsystem. Finally, Multi2Sim includes its own implementation of OpenCL and CUDA libraries. In this way, it can provide dynamic information about CPU-GPU interaction by instrumenting OpenCL and CUDA calls.

1.3 Thesis Objectives

The general objective of this dissertation is improving current GPU memory subsystems in order to increase the overall system performance when executing GPGPU applications.

For this purpose, we need first to characterize the behavior and demands of GPGPU applications from the memory hierarchy point of view, as well as the impact of this hierarchy on performance. This study will provide insights on the main performance bottlenecks on the memory hierarchy. Based on this study new approaches will be devised to remove or mitigate the identified bottlenecks. After that, we need to implement the devised approaches in a state of the art simulator modeling recent GPU architectures. To this end, we need to update existing simulators to accurately modeling the GPU memory subsystem of current GPU architectures.

In short, this thesis pursues as a key objective the design of an efficient memory hierarchy management approach to boost the performance of GPGPU applications, which needs from the previous achievement of two sub-objectives: i) a detailed characterization study relating the impact of the memory hierarchy on the performance, ii) extending state-of-the-art simulators to accurately modeling current GPUs.

1.4 Contributions

This thesis makes three main contributions, each one addressing a specific sub-objective, discussed below:

- A characterization is performed in order to better understand the behaviour of GPGPU applications. The study modifies the underlying coherence protocols and the size of the MSHR file, which directly affects the available Memory Level Parallelism (MLP).
- The accuracy of a state-of-the-art GPU simulator is improved and validated by modeling several critical memory subsystem components and extending the models of existing ones. The results have been validated against those obtained with a real commercial GPU.
- A new proposal that improves the performance of GPU memory subsystems is presented. The proposed approach raises the GPU computational power by unclogging the LLC miss management and improving the hit ratio. It can scale to the largest GPUs from AMD while reducing the energy consumption of the memory hierarchy.

1.5 Outline

Following the UPV rules, this thesis has been written as a compendium of articles. Therefore, the rest of this thesis is organized as follows:

Chapters 2 to 6 present the scientific publications derived from the work performed in this thesis. They have been adapted to the required formatting style.

In Chapter 7, a general discussion of the results of the main contributions of this thesis is given.

Finally, in Chapter 8, some conclusions and ideas for future work are presented.

Chapter 2

Impact of Memory-Level Parallelism on the Performance of GPU Coherence Protocols

- **Authors:** Francisco Candel, Salvador Petit, Julio Sahuquillo, José Duato
- **Type:** Conference
- **Conference:** 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)
- **Location:** Heraklion, Greece
- **Year:** 2016
- **DOI:** 10.1109/PDP.2016.67

2.1 Abstract

Graphics Processing Units (GPUs) are being implemented in heterogeneous CPU/GPU systems due their high efficiency when executing massively parallel applications. New challenges appear to deal with *heterogenous* coherence in these systems due to the huge amount (hundreds or thousands) of on-going memory requests of GPUs, which is limited by the Miss Status Holding Register (MSHR) file size associated to the L1 cache. This paper analyzes how the number of MSHRs i) affects to typical memory performance metrics and ii) impacts on the system performance under two recent GPU coherence protocols, called NMOESI and SI (Southern Islands), which introduce distinct coherence traffic. We find two key findings that can help improve the performance of coherence protocols. First, there is a strong correlation between system performance and memory subsystem latency regardless of the used protocol. Second, system performance varies with the number of supported cache misses; however, counterintuitively, supporting more cache misses does not always bring enhanced performance but it can turn into performance drops.

2.2 Introduction

Nowadays, heterogeneous CPU/GPU processors are being introduced in the market. These systems combine CPU with GPU computing capabilities [21]. The CPU is used to accelerate the execution of the sequential part of the applications, while the GPU allow the execution of a massive number of threads in parallel.

This paper studies the impact on performance of the supported MLP (Memory level paralelism) by GPU considering both GPU and CPU coherence protocols. For this purpose, we first characterize the behavior of GPGPU (General Purpose GPU) applications increasing the supported MLP up to 256 memory requests. This study is done in two state-of-the-art GPU coherence protocols: NMOESI from the academia and Southern Islands (SI) from AMD.

Two important findings are presented that can help improve existing GPU protocol designs. First, unlike CPU memory systems, we find that a higher number of MSHRs can rise cache and memory contention, which can turn into performance drops in some applications. Consequently, in this paper we claim that GPU systems must support a configurable MSHR file size for better performance. Second, huge memory latencies (by 2K processor cycles) cannot be hidden even by the massive thread parallelism of current GPUs. Thus, latency values higher than

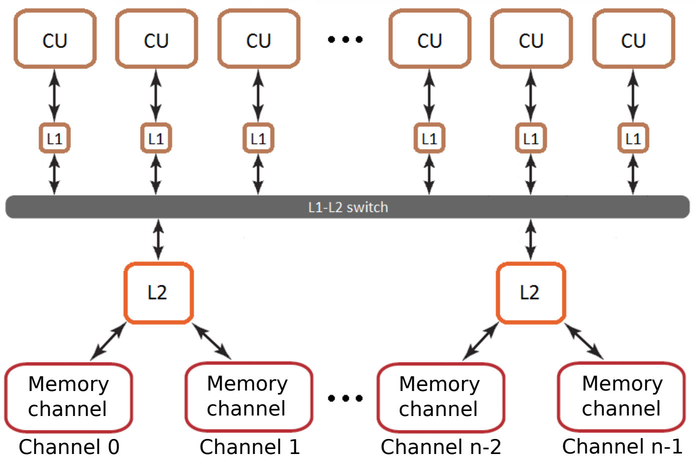


Figure 2.1: Southern Islands memory hierarchy.

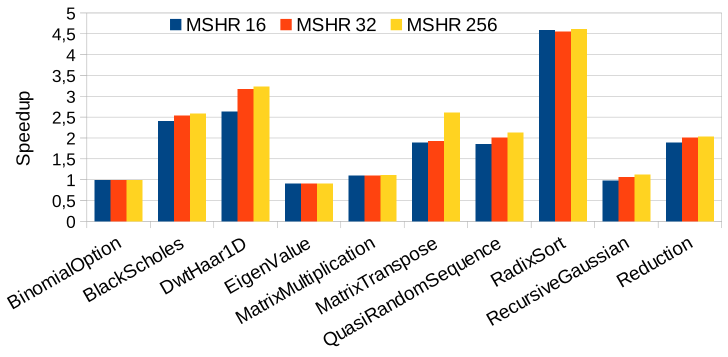
this threshold present an inverse correlation with performance. Finally, we also show that specific GPU protocols are required since both NMOESI and SI protocols provide performance improvements up to 4× over MOESI.

2.3 GPU Architecture

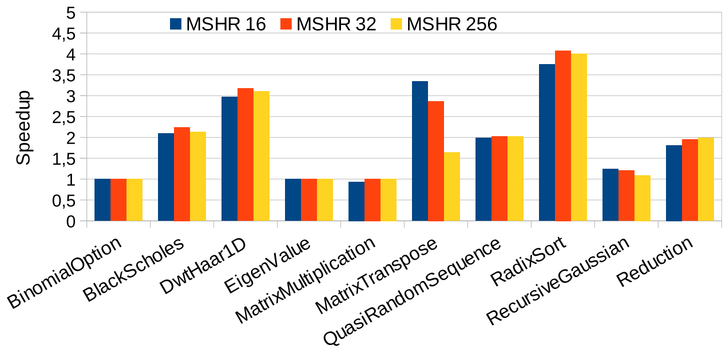
Southern Islands family [8] was the first GPU implementing the AMD's Graphics Core Next (GCN) architecture [68]. Its memory subsystem consists of 3 memory levels as depicted in Figure 2.1: first-level caches, second-level caches, and main memory. All these levels work with a 64-byte block size. The first level cache is composed of data caches (read-write), instruction caches (read-only), and constant caches (read-only). L1-instruction and L1-data caches are private to each Compute Unit (CU), while each constant cache is shared by a group of 4 CUs. L2 is composed of a single cache partitioned into modules, each of them connected to a different dual-channel main memory controller. L1 caches and L2 modules are connected through a crossbar. Finally, depending on the specific card model, main memory is divided into 4, 8, or 12 GDDR5 memory modules.

2.4 Axes of Characterization

This section summarizes the main features of the memory protocols studied in this paper and describes the MSHR file. Both elements of the system are used in the next section to characterize the memory behavior of GPU kernels, since as experimental results will show, the overall system performance strongly depends on them.



(a) SI



(b) NMOESI

Figure 2.2: Speedup of SI and NMOESI with respect to MOESI varying the MSHR file size.

A. Coherence protocols

MOESI: Currently, MOESI is the protocol commonly implemented to guarantee cache coherence in conventional CPU processors. Under this protocol, a given cache block can be in one of five different states (M,O,E,S and I).

NMOESI: This protocol is an extension of MOESI proposed by Multi2Sim team to improve the performance of MOESI in GPU memory systems. A new state N is added to save unnecessary coherence traffic caused by *non-coherent* blocks. When a GPU write access is issued, the requested block is brought to L1 and its state is set to N without invalidating other copies of the block. Thus, multiple non-coherent copies of the same block are allowed in different L1 caches. Then, when a block in state N is replaced, –the part of the block that has been locally modified– is updated in L2, which properly combines the modifications of individual CUs.

SI: This protocol refers to our implementation of the coherence protocol deployed in the Southern Islands GPUs family, which has been implemented based on the oficial SI instruction set architecture [8].

B. Miss status holding registers

Miss Status Holding Registers (MSHRs) are used in non-blocking caches to handle multiple memory accesses at the same time. Each MSHR records all inflight accesses to a specific block. Therefore, the maximum number of inflight requests is limited by the number of available MSHRs. In this work we vary the MSHR file size of L1 caches to control the available number of blocks being fetched (i.e. MLP) [15].

2.5 Experimental Evaluation

Experimental results have been obtained with the Multi2Sim simulation tool[72]. Multi2Sim is a detailed simulator for heterogeneous CPU/GPU systems. It provides cycle-accurate simulation of the processor pipeline and memory subsystem. Multi2Sim supports the MOESI (baseline in our experiments) and NMOESI protocols, and we extended it to support the SI protocol implemented in recent GPUs. The experiments have been carried out with the OpenCL SDK 2.5 benchmarks [3].

The GPU configuration is shown in Table 2.1, which represents the AMD HD 7770 GPU.

A. Performance of SI and NMOESI with respect to MOESI

This section studies the effect of limiting the MLP on the system performance under the SI and NMOESI protocols. For this experiment, we increase the L1 MSHR file size from 32 to 64 and up to 256 entries. Figure 2.2 presents the speedup of the studied benchmarks for each protocol and MSHR configuration with respect to MOESI with 256 MSHRs. As observed, both NMOESI and SI obtain significant performance enhancements in 6 of 10 benchmarks. In general, compared to MOESI, both NMOESI and SI achieve better performance.

The MSHR file size does not equally affect to NMOESI and SI protocols; but depending on the protocol and the application, a high number of MSHRs (e.g. `MatrixTranspose` with SI) or a low number achieves the best performance. In general, SI (see Figure 2.2a) achieves higher performance benefits as the number of MSHR entries increases. That is, this protocol should be deployed with a high number of MSHRs. In contrast, Figure 2.2b shows that the optimal MSHR file size for NMOESI varies with the application. NMOESI, however, achieves in general its poorest performance with a 256 MSHR file size, and improves as the number of supported misses is constrained. To remark that when the number of MSHRs is significantly reduced, NMOESI's performance can be also affected due to the constrained MLP (e.g., `DwtHaar1D`). In

Table 2.1: GCN configuration and memory subsystem.

GCN & Memory subsystem configuration	
Compute Units	10
Work-groups per CU	10
Wavefronts per wrok-group	4
Work-items per wavefront	64
SIMD units per CU	4
All caches	LRU, 64B-lines, 2 ports
L1 caches	16KB, 4 ways, 1 cicle
L1 texture cache	1 cache per CU
L2 caches	2 modules, 128KB per module, 16 ways, 10 cycles
Main memory	2 channels per L2 module, 100 cycles

contrast, SI shows a scalable behavior, improving its performance as the number of MSHRs is increased.

Overall, NMOESI offers the best performance for standard MSHR file sizes (e.g., 16 entries) in half of the studied applications. However, in some applications like `BlackScholes`, this protocol presents worse performance than SI regardless of the MSHR file size.

B. Benchmark characterization

The supported MLP affects in a different way the applications performance depending on the deployed protocol. In this section, we analyze the execution time of the studied benchmarks and classify them into four categories. As example, Figure 2.3 presents the execution time for a benchmark in each category. Next, we present these categories.

SI always better: This category includes those benchmarks where SI achieves better performance than NMOESI regardless of the MSHR file size. This is the case of `BlackScholes` (see Figure 2.3a).

Similar behavior: This category includes those benchmarks where both protocols present similar performance when varying the MSHR file size. Figure 2.3b shows `Reduction` as example.

SI better for large MSHR file sizes: In some benchmarks, both protocols present similar performance for a small number of MSHRs, but differences appear as the MSHR file size in-

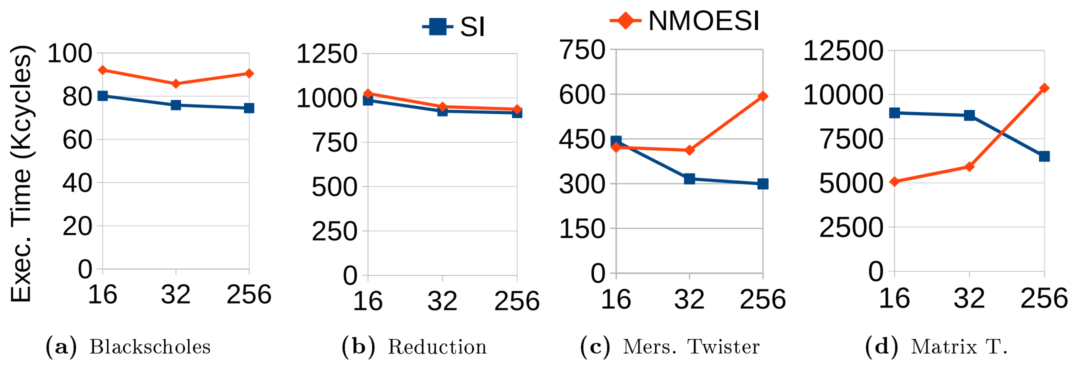


Figure 2.3: Execution time of NMOESI and SI for each benchmark category.

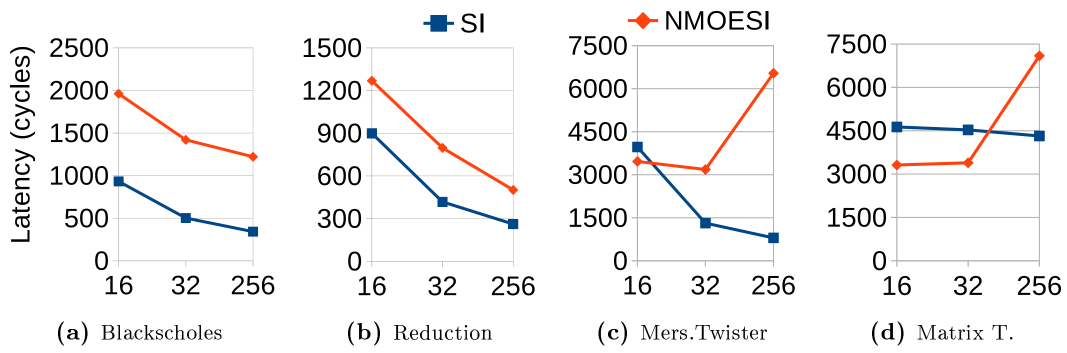


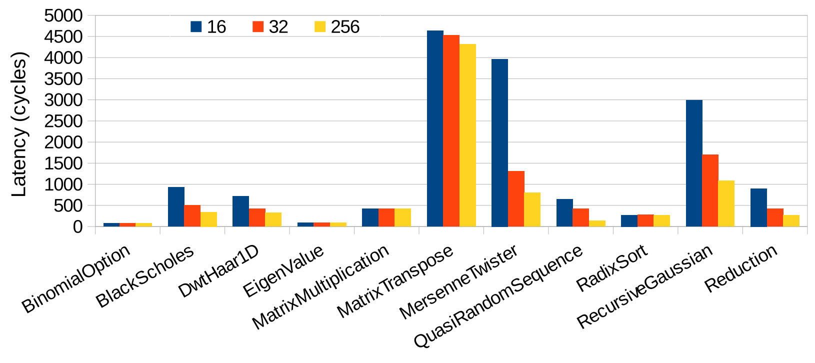
Figure 2.4: Memory latency of NMOESI and SI for each benchmark category.

creases (see Figure 2.3c).

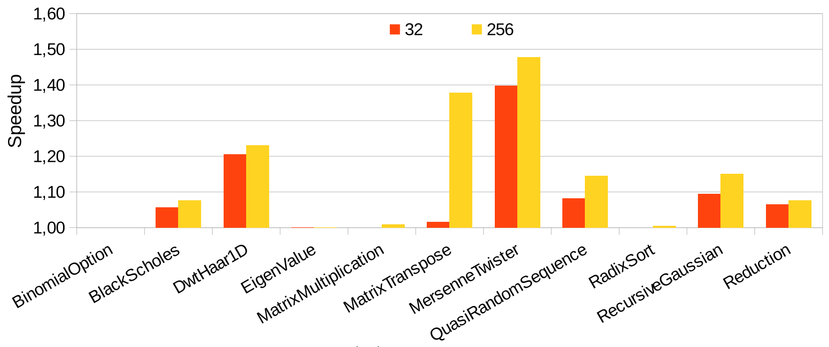
NMOESI better for small MSHR file sizes: This caterogy includes benchmarks,like MatrixTranspose (see Figure 2.3d), where NMOESI with small number of MSHRs achieves better performance, but when the number of MSHRs is increased the SI protocol becomes the best protocol.

To provide a sound understanding of the relationship between execution time and MLP we analyzed multiple memory related metrics such as hit ratio, Misses Per Kilo Instructions (MPKI), and memory access latency. We found that memory latency is the metric that better explains changes in performance due to the low temporal locality and high parallelism of some benchmarks.

Regarding memory latency, remark that GPU memory instructions affect a whole vector of data items, thus potentially generating multiple memory accesses. In addition, current GPU architectures execute instructions belonging to the same wavefront in order. In other words, a GPU must wait for all the accesses generated by a given GPU memory instruction to complete before issuing subsequent instructions. Taken into account this behavior, the latency presented



(a) Memory latency

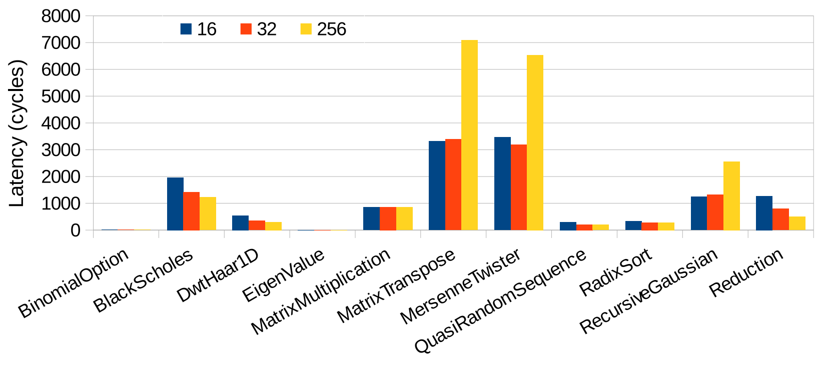


(b) Speedup

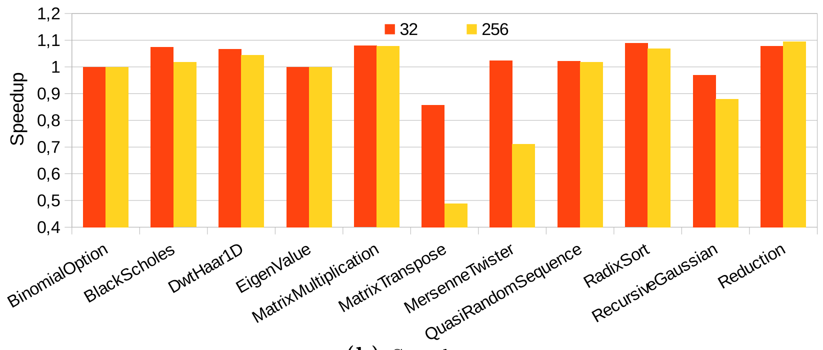
Figure 2.5: SI performance varying the MSHR file size over 16 MSHRs.

by a given memory instruction has been quantified as the maximum latency among all its generated memory accesses; i.e. the maximum latency per vector instruction. We feel that these values accurately reflect what happens in this scenario so we used them to focus the analysis.

Figure 2.4 presents, for the applications representing the four categories, the average of this maximum latency per vector instruction for the NMOESI and SI protocols varying the MSHR file size. Compared to the previous execution time plots, it can be observed that a significant latency reduction does not necessarily reduce the execution time. This is due to the latency-hiding capabilities of GPUs, which come from the fact that application's work-groups are executed following a time-multiplexing manner. In short, in order to latency savings affect the performance, the original memory latency must be higher than a given threshold. For instance, in `Reduction` (Figure 2.4b) a 256-MSHR file reduces latency more than half over 16 MSHRs in both protocols but the execution time is barely affected. Moreover, SI allows by 40% more latency savings than MOESI and, again, this latency improvement does not significantly reduce the execution time. A similar effect can be observed in `BlackScholes` (Figure 2.4a). In contrast, latency plots of `MersenneTwister` (Figure 2.4c) and `MatrixTranspose` (Figure 2.4d) show a similar shape as their corresponding plot in Figure 2.3.



(a) Memory latency



(b) Speedup

Figure 2.6: NMOESI performance varying the MSHR file size over 16 MSHRs.

In summary, we can conclude that the average maximum latency per GPU memory instruction is a good indicator of the expected performance behavior in these applications. The main cause is that their memory latencies are, in general, much higher (by 8 thousand cycles) and cannot be hidden by the GPU microarchitecture.

C. Analyzing the relationship between memory latency and performance in SI and NMOESI

Figures 2.5 and 2.6 present the memory latency (upper plots) and performance (lower plots) under the SI and NMOESI protocols, respectively, varying the MSHR file size. In all the plots it has been used the same protocol with a 16-entry MSHR as baseline. This way allows study how performance is affected by memory latency.

It can be appreciated a high correlation between memory latency variations and performance in both protocols. This correlation is much stronger for high memory latencies (by above 1000 cycles). For instance, `BinomialOption` performance does not vary when changing the protocol. This is because all the studied configurations successfully hide the memory latency, which is very low. This is also the case of `EigenValue`, which presents a very low latency in both NMOESI and SI, thanks to its high L1 hit ratio (by 99%). On the other hand, the performance of `Matrixtranspose`, `MersenneTwister`, and `RecursiveGaussian` clearly depends on the huge memory latencies. In SI, these latencies decrease with the number of MSHRs, which turn into performance improvements. However, the results for the same applications with NMOESI present a noticeable latency increase as the MSHR file size grows, being 16 or 32 MSHRs –depending on the application–, the best configurations for this protocol.

In summary, NMOESI works in general better in configurations with few MSHRs, whereas SI improves its performance with large MSHR files (e.g. 256 entries). On the other hand, SI allows obtaining better memory latencies due to its lack of coherence messages, which enables fast invalidations of blocks.

2.6 Related Work

This section relates important work focusing on caches and protocols for GPUs. Regarding caches, an interesting study analyzing the benefits of cache memories versus scratchpad memories is presented in [33]. This study concludes that some applications improve their performance with cache memories, while the performance may significantly drop in others applications. One possible solution would be to design adaptive memory structures that behave differently depending on the workload characteristics. Regarding coherence protocols, it is known that they should deal with the massively parallel computing capabilities of GPUs. Protocols were originally designed for typical processors having a relatively few number of cache misses at any point in time. Therefore, when they work on GPUs with thousands of requests in flight, they are easily saturated, with the consequent performance loss. In [58], it is presented a coherence protocol that avoids that a massive number of memory accesses saturate the protocol directory. In [65], authors propose a directoryless protocol, which is one of the major performance bottlenecks. The order in which the large amount of memory transactions are processed in the GPU also can significantly affect the performance. In [32], a technique to reorder memory accesses is presented. In [48], reordering requests is also investigated, but focusing on L2 and main memory.

2.7 Conclusions

This paper has shown that coherence protocols especially designed for GPU memory subsystems can accelerate up to 4 times the performance of some applications compared to conventional coherence protocols. Nevertheless, unlike CPU memory subsystems, allowing a higher level of memory level parallelism (i.e. increasing the MSHR file size) can reduce the system performance. This behavior is due to the memory protocol induced contention and can widely vary with the application. In addition, this paper demostrates that: i) the potential negative effect of the protocol on performance can be detected by measuring the maximum latency per vector instruction, and ii) the best MSHR file size depends on the running GPU kernel and the underlying memory protocol.

Chapter 3

Accurately Modeling the GPU Memory Subsystem

- **Authors:** Francisco Candel, Salvador Petit, Julio Sahuquillo, José Duato
- **Type:** Conference
- **Conference:** The 2015 International Conference on High Performance Computing & Simulation (HPCS)
- **Location:** Amsterdam, Holland
- **Year:** 2015
- **DOI:** 10.1109/HPCSim.2015.7237038

3.1 Abstract

Nowadays, research on GPU processor architecture is extraordinarily active since these architectures offer much more performance per watt than CPU architectures. This is the main reason why massive deployment of GPU multiprocessors is considered one of the most feasible solutions to attain exascale computing capabilities. In this context, ongoing GPU architecture research is required to improve GPU programmability as well as to integrate CPU and GPU cores in the same die.

One of the most important research topics in current GPUs, is the GPU memory hierarchy, since its design goals are very different from those of conventional CPU memory hierarchies. To explore novel designs to better support General Purpose computing in GPUs (GPGPU computing) as well as to improve the performance of GPU and CPU/GPU systems, researchers often require advanced microarchitectural simulators with detailed models of the memory subsystem.

Nevertheless, due to fast speed at which current GPU architectures evolve, simulation accuracy of existing state-of-the-art simulators suffers. This paper focuses on accurately modeling the GPU memory subsystem. We identified three main aspects that should be modeled with more accuracy: i) miss status holding registers, ii) coalescing vector memory requests, and iii) non-blocking GPU stores. In this sense, we extend the Multi2Sim heterogeneous CPU/GPU processor simulator to model these aspects with enough accuracy. Experimental results show that if these aspects are not considered in the simulation framework, performance deviations can rise in some applications up to 70%, 75%, and 60%, respectively.

3.2 Introduction

In the recent years there have been an steady increase in the use of GPUs (Graphics Processing Units) for general purpose computing. The main cause is due to General Purpose computing in GPUs or simply GPGPU computing is much more energy-efficient than conventional computing. That is, for the same energy budget, it can provide higher computational power, especially in the execution of massively parallel workloads. Because of this fact, most supercomputers in the top 10 of the top 500 list [71] implement GPUs. For instance, the Titan supercomputer, ranged in second place of the top 500 list in november 2014, was built with Nvidia K20x devices; and the top one, Titanhe-2 was deployed with the Intel Xeon Phi, which incorporates a large graphic unit that occupies a significant part of its layout. However, GPU programmability is still harder than that of conventional computing. To deal with this fact, computer architects are trying to adapt different techniques (e.g. caches and prefetching) that have successfully worked on CPUs to ease programmability and also increase their computational power.

The huge computational power of GPUs comes from implementing hundreds of processing elements that work in parallel. To keep busy all these processing elements with data, memory accesses must be properly handled. This means that the memory hierarchy must provide much more bandwidth than the memory hierarchy of conventional CPU multicores. On the other hand, GPU applications are characterized by their massive parallelism applications (they are usually composed of thousands of logical threads). Based on this fact, the memory hierarchy in the GPU is not designed to reduce latencies as in the CPU but to tolerate a high number of concurrent accesses. This way allows to hide most of the main memory latencies.

The importance of easing the progammability of GPUs for GPGPU computing, as well as the integration in the same chip of CPU and GPU cores (i.e., heterogeneous multicores), which present very different memory hierarchy designs, is driving GPU memory hierarchy research at this moment. To explore and evaluate new proposals and enhancements on the memory subsystem, researchers use complex and detailed simulation frameworks. These software packages are abstractions of the real hardware and model its functionality, concentrating on those hardware components that have a significant impact on the system performance.

However, due to the fast speed at which current systems evolve, as well as their high complexity, simulation accuracy is not always as good as it should. For this purpose, and in order to get representative results, simulators should be continuously updated to reflect the behavior of the real hardware and capture its impact on performance. On the other hand, research on heterogeneous multicores requires from powerful simulation environments that usually model a generic system that often miss significant specific details of GPU architectures.

This paper focuses on enhancing the model of the GPU memory subsystem in the Multi2Sim simulation framework, which is widely used across the scientific community and the academia. Multi2Sim simulates the newest AMD GPU architectures in detail and allows users to configure internal architectural parameters, the characteristics of the modules of the cache hierarchy, as well as the interconnection network. Unfortunately, some parts of the memory subsystem that, as experimental results will show, have a high impact on GPU performance are not accurately modeled, leading sometimes to important performance deviations.

In particular, this paper enhances the Multi2Sim accuracy by modeling three key aspects of current GPU memory subsystems: i) the Miss Status Holding Register (MSHR) file, ii) a coalescing unit at the processor pipeline of the memory requests generated by vector memory instructions, and iii) non-blocking GPU store instructions. The first mechanism allows estimating the effect of the MSHR file on performance. The second Multi2Sim extension coalesces the memory requests issued by the same vector memory instruction at the processor pipeline level, which provides a more realistic model of the memory access patterns affecting the memory subsystem. Finally, the third enhancement avoids the GPU processor pipeline to be blocked when a vector store instruction is located at the head of the vector memory buffer.

Experimental results show that: i) modeling the MSHR file can reduce the performance up to 3 times with respect to assuming an unbounded MSHR file; ii) coalescing at the processor pipeline can speedup the execution time higher than 30% in some applications; iii) non-blocking stores improve the performance across all the studied benchmarks and up to 60% in some cases. In summary, not modeling these realistic hardware mechanisms can result in important performance deviations.

The remainder of this work is organized as follows. Section 3.3 presents a relevant subset of current GPU simulators. Section 3.4 describes the Southern Islands architecture and its programming model. In Section 3.5, the proposed Multi2Sim extensions are described in detail. Section 3.6 presents the experimental results. Finally, in Section 3.7 some concluding remarks are drawn.

3.3 Related Work

GPU research simulators are relatively young and still maturating. In fact, the number of available GPU simulation frameworks is nowadays much lower than that of CPU simulators. The main reasons of this lack of tools is the few information given by GPU manufacturers as well as the fact that the architecture of modern GPUs has been and is quickly evolving, hampering the design of GPU simulators which require an established and well-known architecture model. In spite of this fact, due to the growing use of GPUs, some GPU simulation frameworks have become recently available. Below, we describe a representative set of them.

GPGPU-Sim [23, 11] is currently one of the most referenced GPU simulators. It is a detailed cycle by cycle simulator that supports CUDA version 3.1. It models a GPU microarchitecture similar as the Nvidia GeForce 8x, 9x, and Fermi series. GPGPU-Sim also simulates the interconnection network between SIMT cores and memory modules. Recently, the Gem5 [12] discrete event driven computer system simulator platform was combined with GPGPU-Sim to implement a full heterogenous system simulator. Moreover, GPGPU-Sim version 3.2.0 and later integrate GPUWattch [39] as well, an energy model based upon McPAT [42]. However, due to its dependence on Nvidia drivers, which only support OpenCL 1.1, GPGPU-Sim is not appropriate to evaluate GPGPU benchmark suites like that provided by AMD [3] with modern OpenCL code.

Barra [22] is a parallel GPU functional simulator. It is based in the UNISIM framework [10] and implements both a CUDA driver emulator and a Nvidia Tesla GPU simulator. In this way, Barra can execute directly unmodified CUDA programs and generate statistics at the instruction level. However, presents two main shortcomings. It only supports CUDA 2.2 while nowadays Nvidia has already launched CUDA 7. In addition, Barra does not simulate the GPU microarchitecture, thus it does not provide support to evaluate possible enhancements in the memory subsystem.

Multi2Sim [73, 72] is an accurate cycle by cycle execution driven simulation framework for CPU-GPU heterogeneous computing. Release and development versions of multi2sim are available. It provides a fully configurable memory subsystem with several cache levels and interconnection network. Multi2Sim implements several GPU architectures from both AMD (Evergreen, Southern Islands) and Nvidia (Fermi) as well as CPU architectures like x86, MIPS-32 and ARM. The Multi2Sim developer team is currently modeling the HSA heterogeneous architecture [1], where CPU and GPU share the same memory subsystem. Finally, Multi2Sim includes its own implementation of OpenCL and CUDA libraries. In this way, it can provide dynamic information about CPU-GPU interaction by instrumenting OpenCL and CUDA calls.

In summary, we chose Multi2Sim since it i) simulates a full system cycle by cycle, ii) implements the recent AMD GPU core architectures called GCN [68], iii) includes its own OpenCL and CUDA libraries, and iv) support for HSA architecture is being developed.

3.4 Southern Islands GPU Architecture and Programming Model

This section describes the architecture and programming model of a recent GPU to illustrate how GPUs work. For this purpose, we selected the Southern Islands GPU from AMD, presented

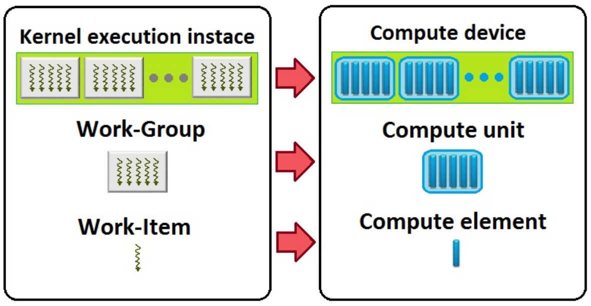


Figure 3.1: OpenCL relation between platform and execution models

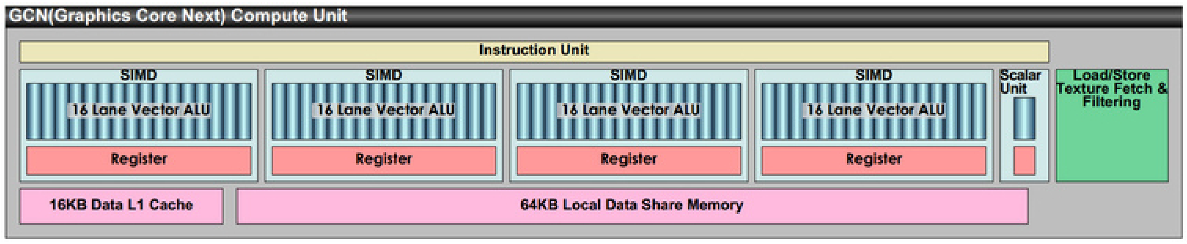


Figure 3.2: Graphics Core Next microarchitecture. Source: Hiroshige Goto (PC Watch)

in 2012, and modeled by Multi2Sim. This GPU is internally implemented with distinct cores that share the memory subsystem. The architecture of the cores and the memory subsystem are described below. In addition, the OpenCL framework, which is used to program the Southern Islands GPU, is also introduced.

3.4.1 OpenCL Framework

There are two main frameworks for GPGPU programming: CUDA from Nvidia and OpenCL from the Khronos group. While CUDA only is supported by GPUs manufactured by Nvidia, there are OpenCL implementations that work on devices from different brands such as Intel, AMD, ARM, or even Nvidia.

The platform model defines the concepts of *Compute Device*, *Compute Unit* (CU), and *Processing Element* (PE) to refer to whole GPU chip, an individual GPU core, and the computing node within the core where the thread is allocated to, respectively, as illustrated in Figure 3.1.

The OpenCL execution model defines several levels of thread organizations. An individual thread is defined as a *Work-Item*, which are clustered in *Work-Groups*. A given GPU application (also known as a *Kernel*) is composed of several work-groups. Figure 3.1 depicts the relationship between the platform and execution models.

3.4.2 Graphics Core Next Microarchitecture

A Southern Islands GPU can include up to 32 CUs implementing the AMD's Graphics Core Next (GCN) microarchitecture. A GCN compute unit is capable of executing 64 work-items in parallel. Together, these 64 work-items are named a *wavefront* and execute instructions in lockstep. Thus, at a given point in time, the 64 work-items composing a wavefront are executing the same instruction on multiple data (SIMD).

The SIMD hardware in a CU is divided in 4 16-lane vector ALUs (see Figure 3.2). Each vector ALU is in charge of executing 16 work-items or a *subwavefront*. In addition, a GCN compute unit also includes a scalar unit and several load/store units.

3.4.3 Memory Subsystem

In Southern Islands GPUs, the 64 memory requests generated by a vector load instruction are coalesced before accessing the memory subsystem. The coalescing mechanism combines in the same memory access those load requests that reference the same cache line. In this way, the number of potential memory accesses is highly reduced. Regarding store instructions, they are not coalesced in the same way that load instructions, but merged once they arrive to the memory queues accessing the different memory and cache modules. Remark that the above management distinction between load and store instructions is specific to the Southern Islands architecture. Other GPU architectures (e.g., AMD's Evergreen family [49]) do not present this management.

Once the memory requests have been coalesced, they are issued to a 16KB L1 data cache (see Figure 3.2), which represents the first level of the memory hierarchy. Only those accesses that miss in the L1 cache access the multi-banked L2 cache through an all-to-all crossbar switch. In addition, the CU has a 64KB Local Data Share (LDS) memory.

L2 banks are address interleaved and connected to main memory modules with private (one per L2 bank) 64-bit wide dual channel memory controllers. Southern Islands GPUs may include up to 6 of these memory controllers. Thus, up to 12 DRAM modules can be installed in the system.

3.5 Proposed Multi2Sim GPU Extensions

Multi2Sim GPU memory subsystem model shares the same source code as its CPU counterpart. This way, which eases the modeling of heterogeneous CPU-GPU processors and allows a more generic implementation, is one of the main reasons due to some particular aspects of GPUs are not modeled with enough detail or in a more accurate way (e.g., coalescing vector load requests in the CU pipelines versus merging them in the memory queues). As we show in Section 3.6, these variations incur significant (positive or negative) impact on performance.

In order to improve the accuracy of the Multi2Sim GPU model, we have implemented three main extensions, detailed below: i) Miss Status Holding Registers (MSHR) file, ii) coalescing vector memory requests, and iii) non-blocking GPU store instructions.

3.5.1 MSHR File Modeling

GPUs generate a huge quantity of memory accesses, but only a limited number of pending cache requests are allowed at a given point in time. For this purpose, current non-blocking caches implement MSHR files. Upon a cache miss, the MSHR file is looked up to check if the target block is already being fetched. On such a case, the missing memory access is queued into the MSHR entry associated to the target block.

Note that a given MSHR entry is in charge of tracking all the memory accesses to a given cache block (i.e., all the requests whose data address falls within the block). Therefore, the maximum number of outstanding memory accesses is limited by the number of MSHR entries. Consequently, if all MSHR entries are busy and the missing cache block is not being fetched, the memory access is stalled until a MSHR entry is released.

Multi2Sim MSHR Model

In Multi2Sim, two main parts can be distinguished in a CPU or a GPU model, the processor pipeline and the memory subsystem. The processor pipeline models the hardware more closely related to the processor pipe stages excluding first-level caches. First-level caches are modeled in the memory subsystem, which considers all the parts of the cache hierarchy and the main memory. In this context, a memory access enters into the memory subsystem as soon as it is issued by the pipeline logic to access the first level of the memory hierarchy.

Multi2Sim only models the MSHR file in the CPU pipeline but no MSHR file is modeled for the GPU pipeline. Moreover, the model only considers first-level cache misses. Thus, in a Multi2Sim GPU model, the number of outstanding cache blocks handled by any L2 cache is virtually unbounded. As Section 3.6 will show, this implementation provides important performance deviations.

Modeled MSHR Extension

We propose to decouple the MSHR from the pipeline model, and to associate a MSHR file to each cache structure in the memory subsystem. This implementation provides a more accurate simulation in both CPU and GPU architectures, since they share the same source code for modeling the memory subsystem. Our implementation allows the MSHR files of distinct cache structures to present a different number of entries, closely mimicking the real implementation of commercial machines.

Our implementation works as follows. When a cache access misses in a given L1 cache, the associated MSHR file is accessed. If the comparison matches, the access is queued in the corresponding MSHR entry. If the comparison fails, a free MSHR entry is allocated. Then, the block is looked up in the corresponding L2 cache. If the L2 copy of the requested block is being involved in other operations (e.g., it is being replaced), then a *nack* signal is returned to the L1 cache, which will retry the operation later. When this situation occurs, to make an efficient use of the MSHR file, the associated MSHR entry is released and the associated queued accesses are moved to a special retry queue. When finally the missing block is transferred to the L1 cache, its associated MSHR entry is released and the memory accesses queued to that entry are satisfied, letting the processor pipeline follow with its normal operation. On a L2 cache miss, the described mechanism is applied recursively to a lower level of the memory hierarchy (e.g. L3 or main memory). Finally, if there are not any free MSHR entry available when it is required, the access waits for a free entry in the MSHR waiting queue, from where they are accessed in FIFO order as soon as an MSHR file entry is freed.

3.5.2 Coalescing Vector Memory Requests

Grouping threads in wavefronts helps improve the memory system performance. For example, in the GCN microarchitecture, one of the main factors limiting the amount of in-flight accesses is the size of the vector memory instruction buffer (VMB) within the vector memory unit (VMU) in the CU since each instruction stored in this buffer can generate up to 64 memory requests. A given instruction is stored in the VMB until all its associated memory requests finish. Assuming a 32-entry VMB (a default value used in our experiments in absence of publicly free available information from AMD), there can be up to 2048 (32×64) memory requests in flight at a given point in time. Note that this is the number of memory requests that can be issued by only 1 CU. This situation clearly makes the memory subsystem to become an important performance bottleneck.

To alleviate this situation, GPU architects group memory requests of the same type (load or store) to the same cache line into a single memory access, so reducing the effective number of memory accesses. This way reduces the pressure on the memory hierarchy.

Coalescing and Merging Approaches

Two main approaches, or a combination of them, are being followed in current GPU designs to reduce the number of memory accesses: coalescing and merging. The coalescing approach implements a coalesce logic that combines multiple requests belonging to the same vector memory instruction into a single cache access. This logic acts in the VMU just before the access is issued to the memory subsystem, thus it is synchronized with the instruction issue stage.

In contrast, the merging approach is implemented within the memory subsystem, decoupled from the VMU; in loads and store queues. Unlike the previous approach, memory requests from the same vector instruction may arrive at the memory subsystem at different points in time. For example, in the GCN microarchitecture each subwavefront issues the memory accesses from the same vector memory instruction in a different clock cycle. Thus, distinct cache accesses can potentially rise from requests from the same instruction, even if finally those cache accesses target the same cache line.

Different commercial GPUs implement one of both approaches or a specific combination. AMD Evergreen [2][49] support coalescing for both loads and stores. In contrast, in the Southern Islands architecture, load requests are coalesced while store requests are merged.

Multi2Sim Coalescing Model

Multi2Sim implements a generic merging model to access the L1 caches that is applied both in GPU and CPU architectures. This model can merge multiple memory accesses regardless of their amount and if they are produced by the same or different vector memory instructions, although some restrictions are applied to attend memory coherence and consistency issues.

However, coalescing is not implemented in Multi2Sim even for the GPU architectures, which highly benefit from this approach. As shown in Section 3.6, the Multi2Sim model is incomplete since coalescing instead of merging can sometimes lead to significant performance differences.

Modeled Coalescing Extension

We have extended Multi2Sim with a flexible coalescing *and* merging implementation that allows any approach or combination to be accurately simulated. In addition to the merging capabilities of the original implementation, the proposed extension can coalesce memory requests from the same instruction in a single memory access to be issued later to the memory subsystem.

3.5.3 Non-Blocking Stores

As explained above, a given vector memory instruction is stored in the VMB until its associated memory accesses are finished. In addition, due to in-order design of CU pipelines vector memory instructions release their VMB entry in program order. Nevertheless, commercial implementations can optimize this behavior by allowing store instructions to release its entry as soon as their memory accesses have been issued provided that the previous stores in program order have already issued their memory accesses. This optimization can be performed in GPUs due to the relaxed consistency model supported by OpenCL.

Minimum constraints to support OpenCL memory consistency

This section summarizes the OpenCL relaxed consistency model to analyze the restrictions that it may impose to real hardware. Note that to the best of our knowledge, there is not published information about any commercial implementation explaining how it supports the OpenCL consistency model.

OpenCL's relaxed consistency is organized hierarchically for a work-item, several work-items of the same work-group, and between work-groups as follows [24]:

1. Within a work-item (i.e. thread) two reads and writes to the same address are not re-ordered by the hardware.

2. For different work-items belonging to the same work-group, memory consistency is only guaranteed by barrier operations.

3. Consistency is not guaranteed between different work-groups.

To guarantee the first condition, there is no need to force that stores to the same address complete execution in order but that they issue in order to the memory subsystem. The reason is that the cache controller handles memory requests to the same address in arrival order. This means that a write can be issued as soon as it is ready and that all the previous memory instructions to the same address have already been issued.

Regarding the second and the third conditions, there are no guarantees of store ordering between barrier operations of different work-items, regardless they are in the same or different work-group. Thus, stores from different work-groups executing concurrently in the same CU do not need to follow a particular store ordering.

Multi2Sim Blocking Store Implementation

Multi2Sim, in its original implementation, does not allow a store to leave the VMB until all its associated write operations are finished. As explained above, this behavior, which may be necessary to support a more strict consistency model (like those supported in commercial CPUs) is unnecesarily restrictive for a GPU implementation and, as shown in Section 3.6 can yield to a significant performance impact.

Modeled Non-Blocking Stores Extension

The proposed extension is based on observations of the analysis discussed above. The extension speeds up the execution by early releasing of VMB entries of stores. More precisely, store instructions release their entry as soon as they are issued to the memory system. Nevertheless, we ensure that stores pertaining to the same work-item are not reordered.

3.6 Experimental Results

This section evaluates the proposed extensions and analyzes their impact on system performance. For comparison purposes, experiments have been carried out with the Multi2Sim simulation framework version 4.2 with and without considering the proposed extensions.

To obtain the results, we modeled the recent Southern Island architecture GPU architecture. Table 3.1 summarizes the main machine parameters.

The OpenCL SDK 2.5 benchmarks adapted for Multi2Sim [3] has been used in the evaluation study. These benchmarks are a subset of those that AMD includes in the APP-SDK (Application Parallel Programming - Software Development Kit). Each benchmark is composed of a x86 host program, which is compiled with Multi2Sim OpenCL library, and a pre-compiled version of the respective OpenCL Device Kernel. Three versions are available: x86, Evergreen and Southern Islands.

Performance are evaluated and compared in terms of Operations Per Cycle (OPC). This metric accounts the number of scalar operations performed by each GPU instruction during the execution of the workload. For instance, if 1 vector instruction accounts for 64 individual scalar operations, this metric accounts for 64 instead of 1. Notice that OPC is equivalent to the IPC metric used when evaluating CPU performance. Thus an X% improvement on the OPC speeds up the GPU execution in the same factor.

GPU AMD HD 7770	
GCN Configuration	
Compute Units	10
Work-groups per CU	10
Wavefronts per Work-group	4
Work-items per Wavefront	64
LDS Unit	64 KB, 1 cycle, 32 ports
SIMD Unit	4 per CU, 16 lines, 4 cycles per instruction
Scalar Unit	1 per CU, 1 cycle per instruction
Vector Memory Unit	1 per CU, VMB of 32 entries
Cache Hierarchy	
All Caches	LRU, 64B line, 2 ports, directory latency 1 cycles
L1 Scalar Cache	3 caches (shared by 4, 3, and 3 CUs) 16KB, 4 way, 1 cycle
L1 Texture Cache	1 per CU 1 per CU, 16KB, 4 way, 1 cycles
L2 Cache	2 modules each module is 128KB, 16 way, 10 cycles
Main Memory	2 channels per L2 module, 100 cycles

Table 3.1: Cache-hierarchy and GPU configuration

3.6.1 MSHR size variation

This section highlights the impact of the size of the MSHR file on performance. Experiments were launched varying the number of MSHR entries (4, 8, and 16), and compared to the baseline machine where no MSHR file is modeled. Notice that non-modeling the MSHR means that a virtually unbounded number of outstanding cache misses is supported. We did not found public information about the size of MSHR file implemented in commercial GPUs, thus for evaluation purposes we used the values obtained in [51]. In this work authors calculate using micro-benchmarking that each CU has a 6-entry MSHR file, similar to the MSHR file size of the CPU processors like the Pentium 4, which implements 8 entries in its L1 data cache [13].

Figure 3.3 and Figure 3.4 depict the performance in terms of OPC for those benchmarks presenting low OPC (>= 200) and high OPC (< 200), respectively. As observed, the MSHR size has a high influence on the performance of most benchmarks regardless the OPC, although the

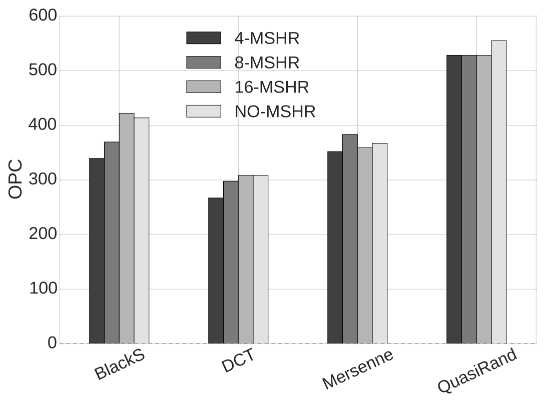


Figure 3.3: Impact of the MSHR file size on OPC ($OPC >= 200$)

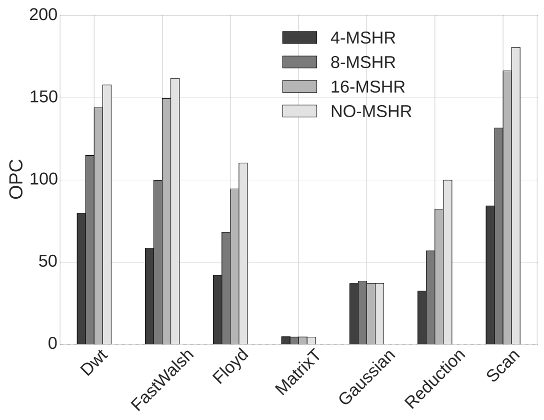


Figure 3.4: Impact of the MSHR file size on OPC ($OPC < 200$)

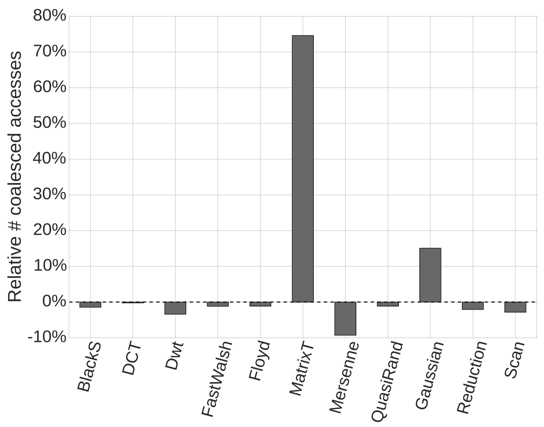


Figure 3.5: Relative number of coalesced with respect to merged accesses

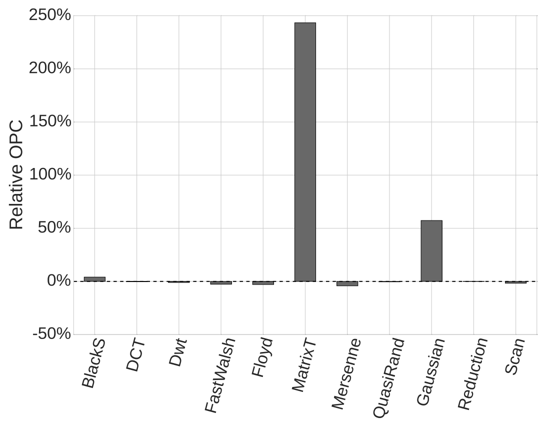


Figure 3.6: Relative speedup of coalescing with respect to merging

impact (in percentage) is stronger in low OPC applications than in high OPC applications. While in high OPC benchmarks the highest difference is `BlackS` (by 20%), in low OPC benchmarks non limiting the number of outstanding misses (NO-MSHR) can rise the performance more than 3× in some cases .

The rationale behind these results is that a high OPC means that the machine is able to extract a high *operation level parallelism* in the SIMD units, and that the memory subsystem is not a major performance bottleneck. Therefore, limiting the amount of outstanding misses to a relatively low number (e.g. 16 or 8) slightly impacts on the performance, even dropping this number to 4 has a scarce impact on most of the high OPC benchmarks. Analogously, a low OPC means that the machine is not able to extract a good *operation level parallelism*. Thus, it is likely that the memory subsystem is bottlenecking the performance. Consequently, if the number of supported outstanding misses is reduced, the performance can dramatically suffer.

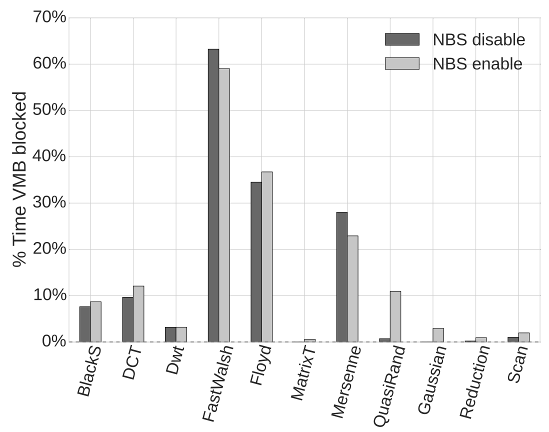


Figure 3.7: Impact of NBS on VMB blocked time

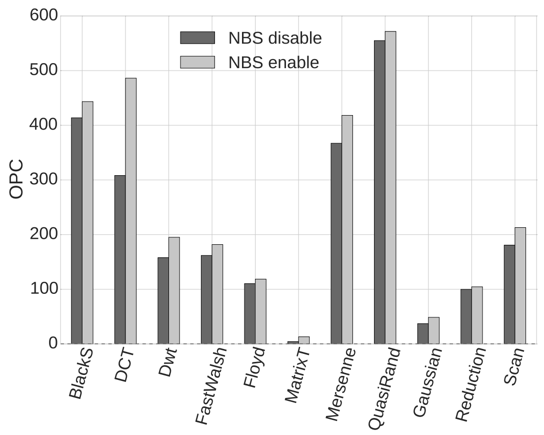


Figure 3.8: Impact of NBS on OPC

3.6.2 Coalescing Vector Memory Requests

This section presents the impact of the modeled coalesce extension on performance. We refer to the modeled GPU pipeline coalescing model as *vector instruction level* approach and to the Multi2Sim coalescing model (i.e., merging) as *access level* approach. This section compares both approaches.

Figure 3.5 and Figure 3.6 show the relative number of coalesced cache accesses and the relative OPC, respectively, of the *vector instruction level* approach over the *access level* approach. It can be observed that the number of cache accesses is quite similar in 9 out of 12 benchmarks; however, important differences appear between both approaches in some benchmarks that rise up to about 15% in Gaussian and 75% in `MatrixT`. Moreover, these values turn into important differences in performance (OPC), which grow by 3.4× and 1.6×, respectively. This happens because *vector instruction level* approach is less restrictive with store requests than the *access level* approach.

3.6.3 Non-Blocking Stores

Figure 3.7 shows the percentage of time the VMB is blocked in the baseline configuration (NBS disabled) and when applying the NBS mechanism. As observed, the VMB is blocked for longer if NBS is enabled in most of the applications. At a first glance, this could seem counterintuitive since NBS allows stores to release VMB entries earlier. However, early releasing VMB entries might unblock those work-groups waiting to the store at the VMB head to finish, allowing them to i) resume issuing memory instructions or ii) terminate the work-group execution so a new work-group can start to issue new memory instructions. On both cases, the VMB would block soon again due to new memory instructions entering the VMB.

Figure 3.8 shows the achieved OPC for the compared machines. It can be appreciated that enabling NBS improves performance across all the studied applications. In some of them like `Mersenne`, OPC grows around 10% and in `DCT` performance grows up to 60%. The reason is due to the NBS mechanism improves significantly the MLP in these applications.

3.7 Conclusions

This work has presented three extensions for the Multi2Sim heterogeneous CPU/GPU simulator. These extensions improve the accuracy of the model of the GPU memory hierarchy, which currently is a very active research field in GPU design.

The first extension models cache miss status holding registers, which are more critical to GPU performance than to CPU due to the high memory level parallelism required by the former. The second extension models more accurately the GPU memory access coalesces, which in current GPU hardware is often performed at the pipeline instead of the memory subsystem. Finally, the third extension increases GPU write throughput by avoiding GPU store instructions to clog the pipeline while waiting the completion of pending memory accesses.

Experimental results show that: i) modeling the MSHR file can reduce the performance up to 3 times with respect to assuming an unbounded MSHR file; ii) coalescing at the processor pipeline can speedup the execution time higher than 30% in some applications; iii) non-blocking stores improve the performance across all the studied benchmarks and up to 60% in some cases. In summary, not modeling these realistic hardware mechanisms can result in important performance deviations.

Chapter 4

Accurately Modeling the On-chip and Off-chip GPU Memory Subsystem

- **Authors:** Francisco Candel, Salvador Petit, Julio Sahuquillo, José Duato
- **Type:** Journal
- **Journal:** Future Generation Computer Systems (FGCS)
- **Volume:** 82
- **Year:** 2018
- **DOI:** 10.1016/j.future.2017.02.012

4.1 Abstract

Research on GPU architecture is becoming pervasive in both the academia and the industry because these architectures offer much more performance per watt than typical CPU architectures. This is the main reason why massive deployment of GPU multiprocessors is considered one of the most feasible solutions to attain exascale computing capabilities.

The memory hierarchy of the GPU is a critical research topic, since its design goals widely differ from those of conventional CPU memory hierarchies. Researchers typically use detailed microarchitectural simulators to explore novel designs to better support GPGPU computing as well as to improve the performance of GPU and CPU-GPU systems. In this context, the memory hierarchy is a critical and continuously evolving subsystem.

Unfortunately, the fast evolution of current memory subsystems deteriorates the accuracy of existing state-of-the-art simulators. This paper focuses on accurately modeling the entire (both on-chip and off-chip) GPU memory subsystem. For this purpose, we identify four main memory related components that impact on the overall performance accuracy. Three of them belong to the on-chip memory hierarchy: i) memory request coalescing mechanisms, ii) miss status holding registers, and iii) cache coherence protocol; while the fourth component refers to the memory controller and GDDR memory working activity.

To evaluate and quantify our claims, we accurately modeled the aforementioned memory components in an extended version of the state-of-the-art Multi2Sim heterogeneous CPU-GPU processor simulator. Experimental results show important deviations, which can vary the final system performance provided by the simulation framework up to a factor of three. The proposed GPU model has been compared and validated against the original framework and the results from a real AMD Southern-Islands 7870HD GPU.

4.2 Introduction

In the recent years there has been an steady increase in the use of GPUs (Graphics Processing Units) for general purpose computing. The main reason is that general purpose computing in GPUs or simply GPGPU computing is much more energy-efficient [29] than conventional computing. In other words, for a given power budget, GPGPUs provide higher performance than their CPUs counterparts, especially when running massively parallel workloads. Because of this fact, most of the top 10 supercomputers in the top 500 list [71] rely on GPUs. For instance, the Titan supercomputer, ranged in second place of the list in November 2014, was built with Nvidia K20x devices. However, GPU programmability [28] is still harder than that of conventional computing. To deal with this shortcoming, computer architects are trying to adapt different components and mechanisms (e.g. caches and prefetching) that have successfully worked on CPUs to ease programmability.

The GPU architecture has been traditionally optimized to run graphic applications workloads, composed of thousands of logical threads, and that exhibit a massive parallelism. For this purpose, the GPU cores present a high computational power which come from including hundreds of processing elements, all of them working together. In order to feed such a high number of computational elements, the GPU core must be coupled with an efficient memory subsystem. Due to this reason, GPU memory subsystems are designed to tolerate a high number of concurrent accesses.

The importance of easing the programmability of GPUs for GPGPU computing as well as the irruption in the market of *heterogeneous* computing processors [14] that combine CPUs and GPUs on the same die, open a new design space for memory hierarchy designs, which is a hot topic in computer architecture research. To implement and evaluate their approaches, academic and industry researchers need from complex and detailed simulation frameworks. These software packages are abstractions that model the functionality of real hardware and focus on those hardware components that have a significant impact on the final system performance. However, because of the fast speed at which current systems evolve, state-of-the-art simulators often miss modeling important components and, consequently, simulation results are not as accurate as they should.

This paper focuses on the memory subsystem, both on-chip and off-chip, of contemporary GPUs. We find that four main important components, which present a significant contribution to the system performance, are not precisely modeled in state-of-the-art GPU simulators with respect to a real device. In particular, three of them correspond to the on-chip memory hierarchy: i) memory request coalescing mechanisms, ii) miss status holding registers, and iii) the cache coherence protocol; while the fourth component refers to the memory controller and the off-chip GDDR memory.

To quantify the impact on performance of these components, we enhance the modeling of the GPU memory subsystem in a state-of-the-art GPU simulator, we quantify the impact of each component on the system performance, and we validate all the components working together by comparing the results of the proposal to the execution time on a AMD Southern-Islands 7870HD GPU. For this purpose, we used the Multi2Sim simulation framework [72], widely used in both the academia and the industry. Experimental results show that each of the studied components, if not accurately modeled, can result in important (e.g. in a factor of $2\times$ or $3\times$) performance deviations in the simulated results.

The remainder of this work is organized as follows. Section 4.3 presents a relevant subset of current GPU simulators. Section 4.4 describes the Southern Islands architecture and its programming model. In Section 4.5, the proposed Multi2Sim extensions are described in detail. Section 4.6 presents the experimental results. Section 4.7 provides the accuracy improvements achieved by the proposed extensions. Finally, in Section 4.8 some concluding remarks are drawn.

4.3 Related Work

GPU research simulators are relatively young and still maturating. In fact, the number of available GPU simulation frameworks is nowadays much lower than that of CPU simulators. The main reasons of this lack of tools is that GPU manufacturers provide little information about the architecture of their processors as well as the fact that the architecture of modern GPUs has been and is quickly evolving, hampering the development of detailed architectural simulators which require an established and well-known model. In spite of this fact, due to the growing use of GPUs, some GPU simulation frameworks have become recently available. Below, we describe a representative set of them.

GPGPU-Sim [23, 11] is currently one of the most referenced GPU simulators. It is a detailed cycle by cycle simulator that supports CUDA version 3.1. It models a GPU microarchitecture similar to the Nvidia GeForce 8x, 9x, and Fermi series. GPGPU-Sim also simulates the interconnection network between GPU cores and memory modules.

Recently, the Gem5 [12] computer system simulator platform was combined with GPGPU-Sim to model a heterogenous CPU-GPU system. Moreover, GPGPU-Sim version 3.2.0 integrates GPUWattch [39], an energy model based on McPAT [42]; a power, area, and timing modeling framework. However, due to its dependence on Nvidia drivers, which only support OpenCL 1.1, GPGPU-Sim does not provide support for the execution of GPGPU benchmark suites like that provided by AMD [3] with modern OpenCL code.

Barra [22] is a parallel GPU functional simulator. It is based in the UNISIM framework [10] and it implements both a CUDA driver emulator and an Nvidia Tesla GPU simulator. In this way, Barra can execute directly unmodified CUDA programs and generate statistics at the instruction level. The major shortcoming of this simulator is that it does not model the GPU microarchitecture, thus it cannot be used to evaluate possible enhancements in the memory subsystem. In addition, this framework only supports a rather old CUDA version 2.2.

Multi2Sim [73, 72] is an accurate cycle by cycle execution driven simulation framework for CPU-GPU heterogeneous computing. Release and development versions of Multi2Sim are available. It provides a fully configurable memory subsystem with several cache levels and interconnection networks. Multi2Sim implements several GPU architectures from both AMD (Evergreen, Southern Islands) and Nvidia (Fermi) as well as CPU architectures like x86, MIPS-32 and ARM. The Multi2Sim developer team is currently modeling the HSA heterogeneous architecture [1], where both CPU and GPU share the same memory subsystem. Finally, Multi2Sim includes its own implementation of OpenCL and CUDA libraries. In this way, it can provide dynamic information about CPU-GPU interaction by instrumenting OpenCL and CUDA calls.

In summary, we chose Multi2Sim because i) it simulates a heterogeneous CPU-GPU cycle by cycle, ii) it implements the recent AMD GPU core architectures called GCN [68], iii) it includes its own OpenCL and CUDA libraries, and iv) support for the HSA architecture is being developed.

4.4 Southern Islands GPU Programming Model and Architecture

This section provides some background on how contemporary GPUs work. To this end, we focus on the state-of-the-art *Southern Islands* GPU from AMD introduced in 2012 which, to the best of our knowledge, is the most recent GPU architecture implemented on a detailed simulator framework. To understand this system, two main axis must be considered: i) its programming model, and ii) its architecture, which consists of multiple cores sharing the same memory hierarchy. Below, both axis are discussed.

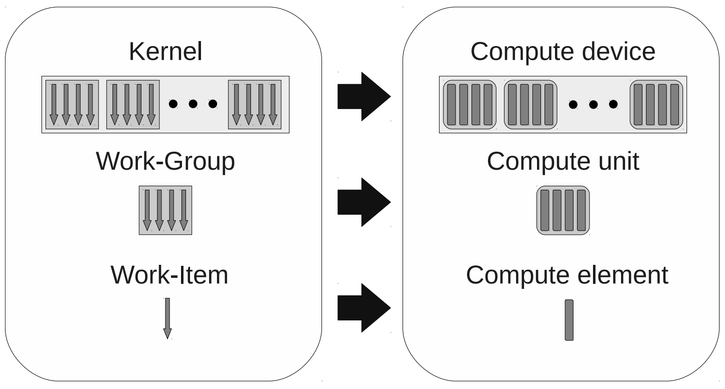


Figure 4.1: OpenCL mapping between execution and platform models.

4.4.1 The OpenCL Programming Model

Two main programming frameworks, CUDA [52] from Nvidia and OpenCL [36] from the Khronos group, are currently being used for developing programs targeting GPGPUs and other kinds of computing devices. OpenCL is*, "de facto",* an industry standard programming model [66]. There are OpenCL implementations that work on devices from different brands such as Intel, AMD, ARM, or even Nvidia, while CUDA is only supported in GPUs manufactured by Nvidia.

The OpenCL specification [35] defines a platform model and an execution model. The platform model is an abstraction of the real machine in which the program will be executed. This model considers one or more *compute devices* (e.g. one GPU) consisting of several *compute units* (CU), each one composed of multiple *processing elements* (PE). On the other hand, the execution model maps the GPU application to the platform model. For this purpose, the execution model defines a hierarchy in which threads are grouped in sets of increasing granularity. An individual thread is called a *work-item*, and they are arranged into *work-groups* limited to 256 work-items. Typically, a GPU program, referred to as a *kernel*, is composed of thousands of work-groups. Figure 4.1 depicts a block diagram of both models and their mapping.

4.4.2 Graphics Core Next Microarchitecture

The Southern Islands GPU can include up to 32 CUs implementing the *AMD's Graphics Core Next* (GCN) microarchitecture as depicted in Figure 4.2. Each CU consists of 4 *single-instruction multiple-data* (SIMD) 16-lane vector ALUs. Thus, considering the 4 SIMD ALUs, the GCN compute unit is capable of executing 64 work-items at the same time.

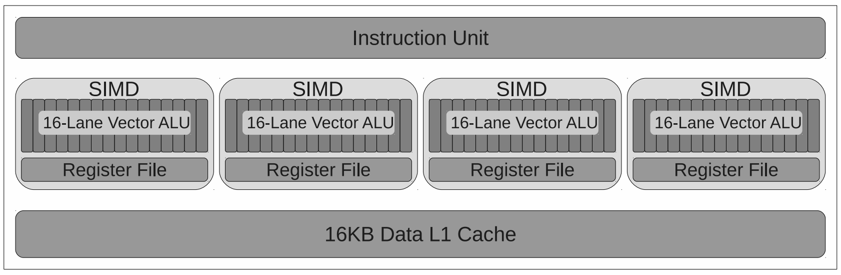


Figure 4.2: GCN compute unit.

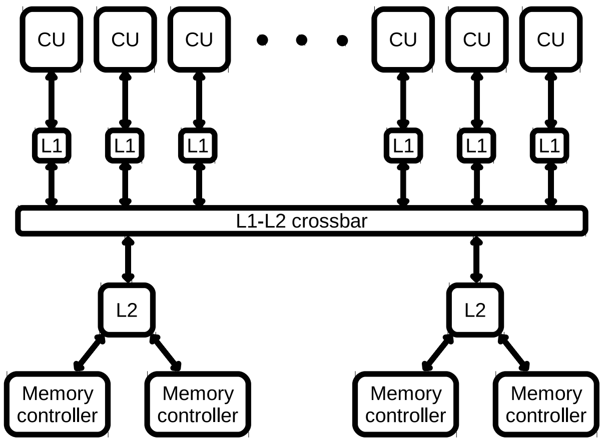


Figure 4.3: Southern Islands memory hierarchy.

In the GCN microarchitecture, a work-group that is mapped to a CU is assigned to a given SIMD ALU. To execute in this ALU, the workgroup is divided in *wavefronts* consisting of 64 work-items. In turn, these wavefronts are subdivided in 4 sets composed of 16 work-items (also known as *subwavefronts*). These subwavefronts are executed sequentially in the SIMD unit.

4.4.3 Memory Subsystem

In Southern Islands GPUs, a load instruction in a wavefront can generate up to 64 memory requests. All the requests generated by a given instruction that access the same cache block are coalesced into a single memory access at the CU before being issued to the memory subsystem. In this way, the number of memory accesses is highly reduced.

The memory subsystem, as in a conventional processor, is organized in a hierarchical way. After the issue stage of the memory instruction the associated memory accesses reach the 16KB L1 data cache (see Figure 4.3), which represents the first level of the hierarchy. Those load accesses that miss in the L1 cache, access the multi-banked L2 cache through an all-to-all crossbar switch. Each L2 bank is connected to two memory controllers that govern the off-

System Component	Multi2Sim Model Restriction	Proposed Extension
Miss Status Holding Registers	Only in L1 caches of CPU cores	Supported in any cache level and for both CPU and GPU cores
Memory Controller and GDDR	Only supports address interleaving among memory modules	Complete memory controller and GDDR model
Memory Request Coalesing	Only merge support	Support for any merging and coalescing combination
Cache Coherence Protocol	Only NMOESI	NMOESI and SI

Table 4.1: Summary of the proposed Multi2Sim extensions.

chip GDDR memory. To avoid channel conflicts and provide more bandwidth, L2 banks at interleaved at the granularity of 256 bytes (8-bit addresses).

Finally, in addition to the memory hierarchy discussed above, each CU includes a 64KB Local Data Share (LDS) memory that it is explicitly managed by the application.

4.5 Modeled Memory Subsystem Components

The Multi2Sim simulator was originally developed for CPU research, and then extended to support GPUs. This simulator models the GCN architecture discussed above in detail, however, it lacks the modeling of the Southern Islands memory subsystem, which as shown in this work can hugely impact on performance.

Multi2Sim GPU memory subsystem shares the same source code as its CPU counterpart. This way, which eases the modeling of heterogeneous CPU-GPU processors and allows a more generic implementation, is probably the main reason why some aspects of the memory hierarchy targeted to GPUs are not accurately modeled.

In order to improve the accuracy of the Multi2Sim GPU model, we have implemented four main extensions to the memory subsystem: i) Miss Status Holding Registers (MSHR) file, ii) memory controller and off-chip GDDR memory, iii) memory request coalescing mechanisms, and iv) a realistic GPU cache coherence protocol. Table 4.1 summarizes the proposed extensions that overcome the restrictions imposed by the current Multi2Sim implementation. Note that all the modifications are orthogonal to the GPU core architecture.

As we show in Section 4.6, the lack of modeling of any of these components incurs on a significant (positive or negative) deviation on the obtained performance. Below, we present and discuss each of the modeled and evaluated components.

4.5.1 MSHR File

GPUs generate a huge quantity of memory accesses, but only a limited number of pending cache requests can be supported simultaneously. For this purpose, current non-blocking caches implement MSHR files. Upon a cache miss, the MSHR file is looked up to check if the target block is already being fetched. On such a case, the missing memory access is queued into the MSHR entry associated to the target block.

A single MSHR entry is in charge of tracking all the memory accesses associated to a given cache block (i.e., all the requests whose data address falls within the same block). Therefore, the maximum number of outstanding memory accesses is limited to the number of MSHR entries. Consequently, if all MSHR entries are busy and the missing cache block is not being fetched, the memory access is stalled until an MSHR entry is released.

Multi2Sim MSHR Model. Multi2Sim only models the MSHR files associated to first-level caches of the CPU cores. However, they are not modeled in the GPU cores. Consequently, in the Multi2Sim GPU model, the number of outstanding misses handled by any cache is virtually unbounded, which is impractical in real devices. This implementation can present important performance deviations, since the GPU throughput highly depends on cache resources such as MSHRs [32].

Some recent works [15] consider the impact of modeling the MSHR file at the L1 caches. However, to the best of our knowledge, there is no any existing proposal modeling the MSHR associated to the L2 cache which, as experimental results will show, can introduce significant deviations in the execution time.

Modeled MSHR Extension. In this work we claim that, in order to obtain accurate results, an MSHR file must be associated to each cache structure in the memory subsystem. Our implementation allows the MSHR files of distinct cache structures to present a different number of entries, closely mimicking the hardware implementation of commercial machines.

Our implementation works as follows. When a cache access misses in the L1 cache, the associated MSHR file is searched. If there is a hit in any MSHR entry, the access is queued in the corresponding MSHR entry. Otherwise, a free MSHR entry (if any) is allocated. After that, the request proceeds by searching the block in the L2 cache. On an L2 cache miss, the described MSHR mechanism is similarly applied and the missing block is requested to the main memory. Finally, when the block is transferred to the caches (L1 and L2), the associated MSHR entry in each cache is released and the memory requests waiting for the block are notified that the data block has been fetched.

In case there is not any free L1 MSHR entry available, the access waits for a free entry in the MSHR *waiting queue*, from where they are accessed in FIFO order as soon as an L1 MSHR file entry is freed. The L2 MSHR file is handled differently to prevent deadlocks; if a request asks for an L2 MSHR entry and no entry is available, a *NACK* signal is returned to L1, and the operation is retried later. For this purpose, we implement an especial *retry queue*.

4.5.2 Memory Controller and Off-chip GDDR Memory

As conventional DDR SDRAM memories, Graphics DDR (GDDR) memory contain multiple independent DRAM banks. A bank is implemented as a matrix of DRAM cells. When a bank is accessed the whole row, also known as *memory page*, is accessed. The accessed memory page is stored in the DRAM sense amplifiers associated to the bank, also referred to as *row buffer*.

The memory controller uses three commands that are issued sequentially to a bank in order to access the target data [38]. First, the *precharge* command writes the contents actually stored in the row buffer to the bank and precharge the row bitlines for accessing the target row. Second, the *activate* command accesses the row that contains the requested data and stores it into the row buffer. Finally, the *read/write* command reads or write the requested data in the row buffer. After issuing the last command, the memory controller can either keep the accessed memory page in the row buffer (open page policy) or close the row buffer by issuing a *precharge command* (closed page policy). Depending on the implemented page policy, the latency of the next access varies. For example, with an open page policy, if the requested block is already present in the row buffer (i.e., a row buffer hit), only a read/write command needs to be issued by the memory controller, thus the latency can be significantly reduced. However, a row buffer miss would require to serialize the issuing of the three mentioned commands, roughly trebling the latency of a row buffer hit.

In a bank access, the memory data bus is used to read from or write to the memory device. In conventional DDR memories the memory bus width is 64 bits, while in GDDR memories this width is typically 32 bits. Since the typical cache block size is 64 bytes, transferring a cache block through the data bus takes several bus clock cycles. To reduce this transfer time, it is possible to increase the width of the memory bus. Since GDDR devices are standardized to a 32-bit bus to work with wider data buses multiple devices are required to operate in lockstep. For example, the Intel i875P memory controller connects through a 128-bit memory data bus to matching pairs of 64-bit wide DIMMs (Dual In-Line Memory Modules). This paired DIMM configuration is often referred to as dual channel configuration [30].

Multi2Sim Memory Model. Modern GPU systems integrate multiple memory controllers. To increase memory parallelism as well as effective memory bandwidth, block addresses are interleaved among the deployed memory controllers. Multi2Sim supports the modeling of this configuration since it allows main memory to be organized as an array of interleaved memory modules. However, it does not model other important aspects affecting memory latency and bandwidth such as banks and channels; thus bank contention and channel contention are not considered. In addition it does not support neither open nor closed page policies.

Integration of Multi2Sim and DRAMSim2. To provide a more realistic simulation of the memory controller and off-chip memory, and to check the impact of such an implementation on the obtained performance, we have combined Multi2Sim with the DRAMSim2 simulator [61]. DRAMSim2 is a recent cycle accurate memory system simulator that models DDR memory systems (memory devices, memory controllers, and memory buses) and supports configurations with multiple controllers and channels as well as typical memory controller policies. Moreover, DRAMSim2 provides accurate performance results that have been validated against real memory systems.

4.5.3 Memory Request Coalescing Mechanisms

Each memory instruction in the GCN architecture, as well as in most modern GPUs, is able to work with up to 64 data items thus it can generate up to 64 memory requests. Taking into account that hundred of memory instructions can be in flight on the entire GPU, we can observe that such a high number of memory requests would bottleneck the memory subsystem.

To deal with this shortcoming, current GPUs implement different schemes that reduce the number of effective cache accesses. Additional queues and memory instruction structures are required with this aim. The GCN microarchitecture implements the *vector memory instruction buffer* (VMB) in the CU. This buffer keeps track of the memory instructions issued to the cache until all their associated memory requests finish. For experimental purposes (in absence of publicly available information) we assume each core has a 32-entry VMB. That is, there can be up to 2048 (32×64) memory requests in flight per CU at a given point in time.

Using the VMB and other structures, as described below, GPU architectures implement mechanisms that group memory requests of the same type (load or store) targeting the same cache line in a single memory access, so reducing the effective number of memory accesses. This way greatly reduces the pressure on the memory hierarchy.

Coalescing and Merging. Two main approaches, namely *coalescing* and *merging*, can be found in modern GPUs to reduce the number of memory accesses. The coalescing approach combines all the requests of the same instruction into a single cache access in the VMU just before issuing the instruction to the memory subsystem. For instance, the AMD Evergreen [2, 49] implements coalescing of loads and stores.

In contrast, the merging approach is implemented in the memory subsystem, decoupled from the VMU. The key difference is that due to GCN microarchitecture constraints, requests from the same memory instruction reach the cache at four different points of time. More precisely, a memory instruction is executed in four phases (or subwavefronts) since a vector operator implements 16 lanes and the wavefront works with 64 data items. Thus a single memory instruction can potentially generate up to four accesses to the cache, even if all of them target the same cache line. A variant of this approach is implemented in Multi2Sim as described below..

Multi2Sim Merging Model. Multi2Sim models a common generic merging mechanism that applies in the L1 cache of its CPU and GPU implementations. This model can merge multiple memory requests regardless of whether they have been generated by the same memory instruction or not. In addition, some restrictions are applied to deal with memory coherence and memory consistency issues.

Coalescing is not implemented in Multi2Sim, however, for the GPU architectures. As shown in Section 4.6, performing coalescing instead of merging, can lead to significant deviations in the final results.

Modeled *Coalescing & Merging* Extension. We have implemented a flexible *coalescing & merging* approach that allows to evaluate each approach either separately or in a combined way.

4.5.4 GPU Cache Coherence Protocol

Cache coherence protocols were originally designed to support data coherence among caches in CPU multiprocessors. These protocols tolerate a moderate traffic of coherence requests, however, they are rather complex and would strangle the performance if they were directly applied to GPUs, mainly due to GPUs must be designed to support a massive amount of memory requests generated by typical GPU applications. In short, neither GPUs nor heterogeneous CPU-GPU systems work properly with typical CPU protocols. To deal with this fact, alternative protocols have been devised both by the academia and the industry.

NMOESI Coherence Protocol. To support GPU cache coherence, Multi2Sim implements NMOESI, that extends the well-known MOESI protocol [67] implemented in a wide range of CPU multicores. NMOESI extends this protocol to support memory coherence in both CPU and GPU applications, and it is especially suited for heterogeneous CPU-GPU systems with a cache hierarchy shared among CUs and CPU cores.

Under MOESI, a given cache block can be in one of five main states (M,O,E,S and I). NMOESI extends this protocol by adding a new non-coherent state (N) to be used in GPUs. This state avoids non-coherent write requests, which are common in GPU applications, generate coherence traffic. When a cache write request is issued, the requested block is brought to the L1 cache and its state is set to N, however, unlike typical write-invalidate protocols, no copy of the block is invalidated in the other L1 caches. In other words, this protocol allows non-coherent copies of a block to co-exist in multiple L1 caches. In case a block in state N is replaced in a L1 cache, only the data items of the block that have been locally modified are updated in the L2.

SI Protocol Extension. We have modeled the protocol deployed in the Southern Islands (SI) GPU family, hereafter SI protocol, which supports a relaxed memory consistency model based on Release Consistency [26]. This consistency model allows the compiler to specify when data modifications performed by a given CU must be visible to other CUs, which enables the implementation of simpler coherence protocols. To support the consistency model, the opcode of a SI memory instruction includes 2 bits called GLC (Global Coherent) and SLC (System Level Coherent), which indicate the coherency scope.

When the SLC bit is enabled in a given instruction, the memory requests that this instruction generates bypass the caches and directly access to main memory. On the other hand, the GLC bit behavior depends on the memory instruction type (load or store). If the GLC bit of a load instruction is set, the L1 cache is bypassed and the blocks are searched in the L2 cache. In contrast, store instructions write their data in the L1 cache regardless of the GLC bit. After the write, if the GLC bit is set, the affected lines are evicted and written back to L2 considering a dirty byte mask that specifies which bytes in the line have been modified [68]. A similar behavior is followed when a block is partially written regardless of the GLC bit. Note that evictions do not add latency to the offending write since they are not on the critical path. However, they increase the L1 cache miss ratio and thus the L2 cache contention, which can affect the performance of subsequent memory accesses.

All writes performed to L1 also modify the L2 copy of the block (i.e., L1 follows a write-through policy). In this way, the same block can be modified in L2 at the same time by several CUs, provided that each of them write to different bytes of the block. In contrast, the L2 cache follows a write-back policy; that is, the main memory is updated when a modified block is replaced from the L2 cache.

We find no information in the checked AMD documentation [4, 68, 8] about if the commercial device forces the inclusion principle among the L2 and the L1 caches, so we modeled the L2 cache as a non-inclusive cache because it generates less traffic in the memory subsystem than an inclusive cache.

4.6 Experimental Results

This section analyzes the impact of the discussed memory components on the system performance. For this purpose, we extended the Multi2Sim simulation framework version 4.2 by modeling (i) the discussed Southern Islands memory architecture, and (ii) the four components to be studied on this architecture. Experiments were launched with and without considering these extensions. Note that Multi2Sim is an *application-only* simulator that only considers the execution of the studied benchmark or user-level application, removing OS and device drivers from the software stack. An important feature of application-only simulators is that they pro-

GCN Configuration	
Frequency	1GHz
Compute Units	10
Work-groups per CU	10
Wavefronts per Work-group	4
Work-items per Wavefront	64
LDS Unit	64 KB, 1 cycle, 32 ports
SIMD Unit	4 per CU, 16 lines, 4 cycles per instruction
Scalar Unit	1 per CU, 1 cycle per instruction
Vector Memory Unit	1 per CU, 32-entry VMB
Memory Hierarchy	
All Caches	LRU, 64B line, 2 ports, directory latency 1 cycles
L1 Scalar Cache	3 caches (shared by 4, 3, and 3 CUs) 16KB, 4 way, 1 cycle
L1 Texture Cache	1 per CU 1 per CU, 16KB, 4 way, 1 cycles
L2 Cache	2 modules each module is 128KB, 16 way, 10 cycles
Main Memory	2 memory controllers per L2 module, 90 cycles

DRAMSIM
configuration timings
in cycles (tCK=0.667)
	CL=18, AL=17, BL=16, tRAS=42 ,tRCD=18,
	tRRD=9, tRC=60, tRP=18, tCCD=3, tRTP=3,
	tWTR=8, tWR=4, tRTRS=1, tRFC=278,
	tFAW=35, tCKE=6, tXP=7, tCMD=1

Table 4.2: Cache-hierarchy and GPU configuration.

duce deterministic results, thus the results presented in this work do not include confidence intervals.

Table 4.2 summarizes the main machine parameters. The OpenCL SDK 2.5 benchmarks adapted for Multi2Sim [3] has been used in the evaluation study. These benchmarks are a subset of the APP-SDK (Application Parallel Programming - Software Development Kit) by AMD. Each benchmark is composed of a x86 host program, which is compiled with Multi2Sim OpenCL library, and a pre-compiled version of the respective OpenCL Device Kernel. Three versions are available: x86, Evergreen and Southern Islands.

Performance has been quantified in terms of Operations Per Cycle (OPC) for comparison purposes. This metric accounts the number of scalar operations each GPU instruction performs, averaged per cycle, during the workload execution. For instance, if 1 vector instruction accounts for 64 individual scalar operations, this metric accounts for 64 instead of 1. Notice that

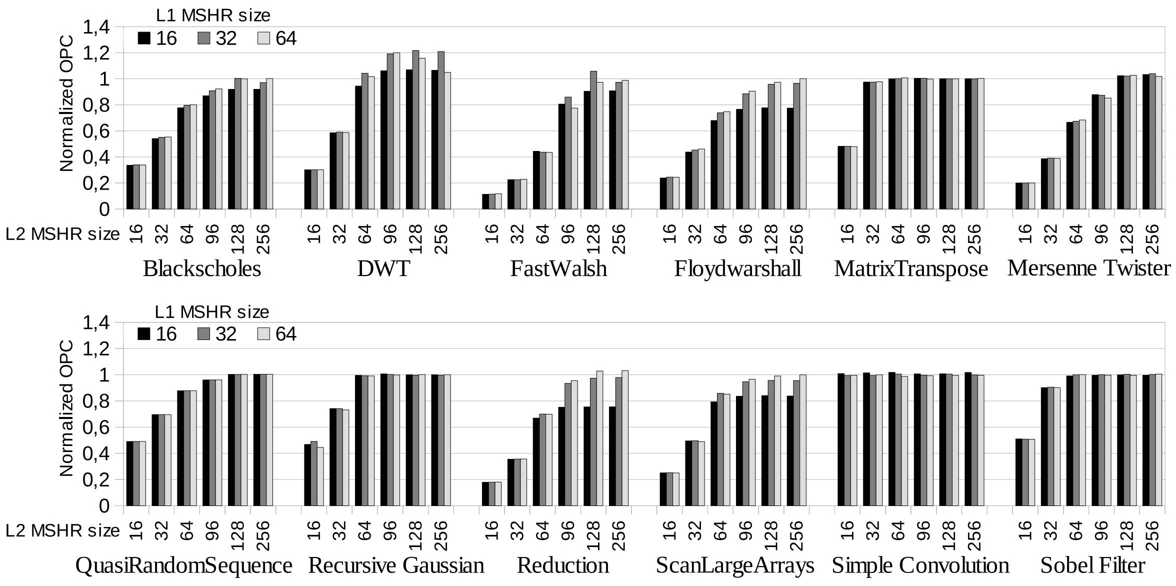


Figure 4.4: Impact of L1 and L2 MSHR file sizes on performance.

OPC is equivalent to the IPC metric used when evaluating CPU performance. Thus an X% improvement in the OPC speeds up the GPU execution in the same factor.

Below the four aforementioned memory subsystem components are evaluated in isolation, that is, each one without considering the effects of the remaining ones.

4.6.1 MSHR File

This section studies the impact of the MSHR file size on the final performance. Experiments were launched varying the size of both L1 and L2 MSHR files. There is not public information about the MSHRs size implemented in commercial GPUs, but recent studies [32][51] have empirically determined that this size is as large as 32 or 64 entries in the L1 of some recent GPUs. Many values have been explored but only a subset of them is presented for illustrative purposes. Regarding the L1 cache, we plot the results for 16-, 32-, and 64-entry MSHR files, and for each of them six MSHR sizes (16, 32, 64, 96, 128, and 256 entries) are presented for the L2 cache. This means that 18 different MSHR configurations are studied. The performance of each MSHR configuration is compared to the baseline machine without MSHR files. Notice that not modeling any MSHR file means that the system can support an unbounded number of outstanding cache misses.

Figure 4.4 depicts the relative performance (i.e. OPC) of each MSHR configuration with respect to the baseline. As observed, the MSHR size has a high influence on the results of most of

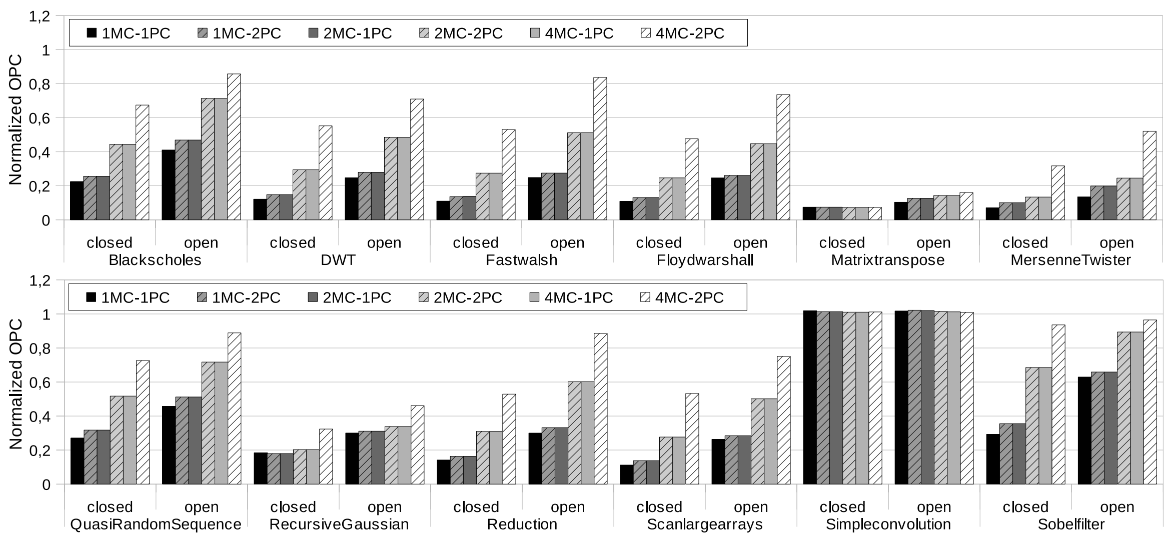


Figure 4.5: Impact of the number of memory controllers, physical channels, and page policy on performance.

the benchmarks. The largest performance variation is due to the L2 MSHR file size. For example, in most applications, the smallest tested L2 MSHR file (16 entries) can reduce the performance below 30% of the baseline performance. Notice that relative OPC is the inverse of the relative execution time (speedup or slowdown). For instance, a relative OPC of 20% over the baseline means that the execution time will take 5× longer than the baseline (e.g., MersenneTwister with a 16-entry L2 MSHR file). As expected, increasing the L2 MSHR file size always increase the performance but the improvements are minor for sizes larger than 96 entries in most benchmarks.

The L1 MSHR file size has a significant impact on the OPC when using L2 files larger than 64 entries in some benchmarks (FastWalsh, Floydwarshall, Reduction, and ScanLargeArrays). Contrary to L2, increasing the number of L1 entries beyond a given value can negatively impact the performance. This situation happens in DWT and FastWalsh. We have detected that this behavior is caused by contention in the L2 coherence directory. When a memory request cannot access the target block directory in the L2, the request is *nacked* and retried later, so increasing its latency. A relatively large L1 MSHR file size (e.g. 64 entries) causes a huge amount of requests to contend for L2, increasing the latency beyond values that cannot be hidden by the GPU massive parallelism. This also causes that, in some benchmarks (e.g., DWT), the performance when limiting the MSHR file can be higher than that of the baseline.

4.6.2 Memory Controller and Off-chip GDDR Memory

Implementation of current DRAM memory devices and memory controllers introduces a new contention level which causes a high variability in both memory access latencies and effective bandwidth. This means that the modeling of these components plays a key role in order to obtain representative performance.

This section explores how these components affect the performance varying the number (1, 2 and 4) of memory controllers connected to each L2 and the number of physical channels attached to each memory controller. Figure 4.5 plots the normalized performance over the baseline which does not model any of them. Each configuration is labeled as *xMC-yPC*, where x is the number of memory controllers and y is the number of physical channels. When only a single MC is available, it is shared by both L2 modules present in the system, while if there are more than one MC, each L2 is connected to half of them. For instance, in the configurations with 4 MCs, two of the MC are connected to the first L2 module and the remaining ones to the other L2 module.

As observed, modeling the memory controllers and the GDDR devices hugely impacts on the final performance. Only one of the applications (Simpleconvolution) is not significantly affected. Comparing the open page policy versus the closed page policy, it can be appreciated that similarly as happens in CPU workloads [50], leaving the page open after a memory access typically offers better performance, especially when the application exhibits good spatial locality, which is the case of typical GPU applications. Regarding the number of memory controllers and physical channels, the figure shows that the performance of the 1MC-2PC configuration matches that of 2MC-1PC while the performance of 2MC-2PC equals that of 4MC-1PC. In principle, increasing memory bandwidth by adding additional memory controllers instead of physical channels provides more access flexibility because memory controllers are logically independent while physical channels connected to the same memory controller work in lockstep, however, it involves more hardware complexity and does not translate to performance benefits in GPU applications. Therefore, results demonstrate that this complexity is not needed when dealing with GPU workloads.

In general, adding more memory controllers or physical channels increase the performance but this increase is reduced as the memory bandwidth ceases to be a performance bottleneck. We found that implementing four or more memory channels does not provide significant performance benefits for most applications.

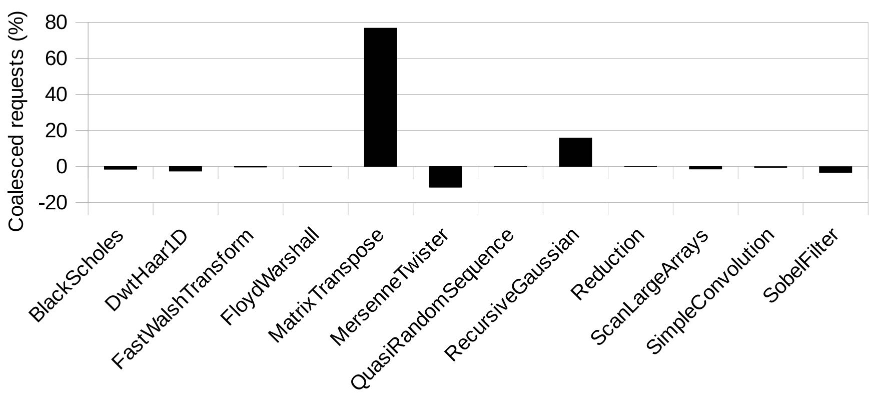


Figure 4.6: Percentage of combined memory requests by coalescing with respect to merging.

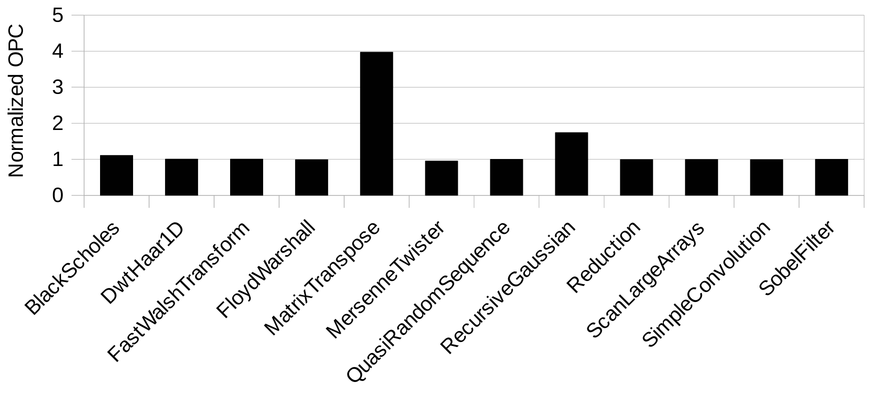


Figure 4.7: Speedup of coalescing with respect to merging.

4.6.3 Memory Request Coalescing Mechanisms

This section compares the impact of coalescing versus merging on the obtained performance. Figure 4.6 and Figure 4.7 show the relative number of combined memory requests and the relative OPC, respectively, of the coalescing approach over merging. It can be observed that the number of combined requests is quite similar in 9 out of 12 benchmarks; however, important differences appear between both approaches in some benchmarks that rise up to about 15% in RecursiveGaussian and 75% in MatrixTranspose. Moreover, these values turn into important differences in performance (OPC), which grows by 3.4× and 1.6×, respectively. This happens because the merging approach sometimes is not able to combine all the memory requests produced by a sequence of subwavefronts that target the same block. This often happens when the memory requests from a subwavefront leave the cache write queue (i.e., access to the cache) before subsequent memory requests enter the queue. This situation cannot occur if a coalesce mechanism is used because the requests are combined before reaching the write queue.

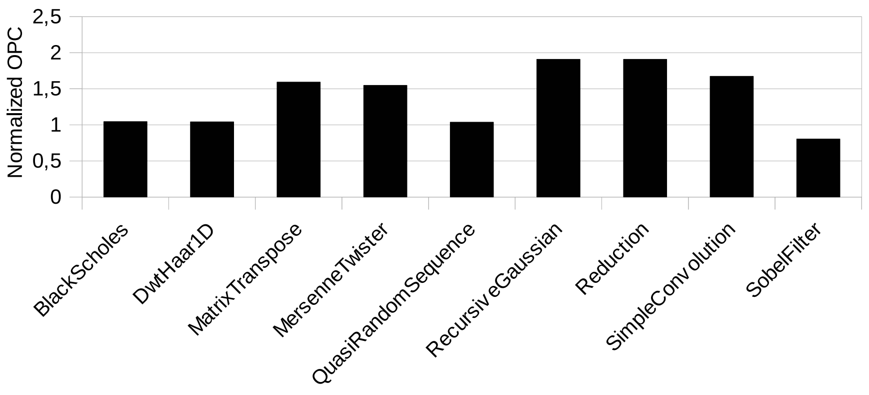


Figure 4.8: Impact of the coherence protocol: SI over NMOESI.

4.6.4 Cache Coherence Protocol

In Section 4.5.4 we discussed two coherence protocols applied to GPUs, NMOESI –with five main states– from the academia that extends the well-known MOESI protocol and SI –with only two main states–, which is much simpler and has been deployed in recent commercial devices.

In this section we compare the performance of both protocols across the studied workloads. Figure 4.8 shows the results. As observed, the SI protocol, in spite of its simplicity, improves the performance over NMOESI by 50% in half of the applications; moreover, in two of them almost doubles the performance of NMOESI. Nevertheless, NMOESI achieves significant benefits in two of the applications.

We looked into the rationale behind these results. We found two main critical aspects related to the details of each protocol implementation that make difficult to find a single cause that explains the performance differences between both protocols.

The first aspect refers to the cache write miss policy. While the SI protocol implements a no-write allocate L1 policy (i.e. the block is not fetched to the L1 cache on a write miss), the Multi2Sim implementation of the NMOESI protocol follows a write allocate policy. Consequently, the SI protocol incurs in a higher number of L1 misses, which does not necessarily yield the system to performance losses since there is a tradeoff among cache space, data locality (e.g. blocks fetched and not reused), and miss penalty.

The second significant aspect is that the L2 cache directory works differently in both protocols. In the NMOESI protocol, when a block is locally written for the first time or replaced in the L1 cache, the L2 directory must be locked to update the coherence information (e.g. the sharer vector). In the SI protocol, this action is not required. Consequently, a cache write miss in the SI protocol usually take less time than in the NMOESI protocol. Moreover, because of the SI protocol does not have to update the directory, a cache write miss can take less time than a write hit in the NMOESI protocol.

To sum up, the internal hardware structures work differently in both protocols which makes misses and hits to take different time depending on the underlying protocol.

4.7 Putting it All Together and Validation

Once the impact of each memory component on performance has been studied in isolation, this section pursues a twofold objective: i) to analyze the combined effect when the components act *all together* simultaneously, and ii) to check how the proposed mechanisms improve the error deviation that the original simulation framework introduces with respect to real hardware. For this purpose, the simulator configuration file was tuned to model the AMD Southern-Islands 7870HD GPU, which is the GPU that we have available. Then, the results of the *all together* model were compared against both the original Multi2Sim simulator and the real AMD GPU.

Regarding the *all together* configuration, we must select for each memory component the configuration that best fits the real hardware. In this regard, the *all together* system has been configured as follows. Coalescing and SI protocol have been chosen instead of merging and NMOESI since they mimic the real GPU hardware. The memory controller, based on official AMD information [4], has been configured to four double-channel memory controllers, one per L2 cache. Finally, the MSHR files for the L1 cache and for the L2 cache have been set to 32 and 96 entries, respectively, since they are realistic values as inferred in [32] and [51].

Notice that the results of the *all together* configuration cannot be compared against those of individual memory components, because the effects of the *all together* system do not match the sum of the effects of the individual components. In fact, we realized that many times the effect of a given component compensates that of another component (e.g. a positive effect versus a negative one) or overlap among them. Therefore, the aforementioned objectives are realized in the same experiment, which shows that our modeled *all together* machine behaves closer to the results obtained in the real hardware than the results provided by the original simulation framework.

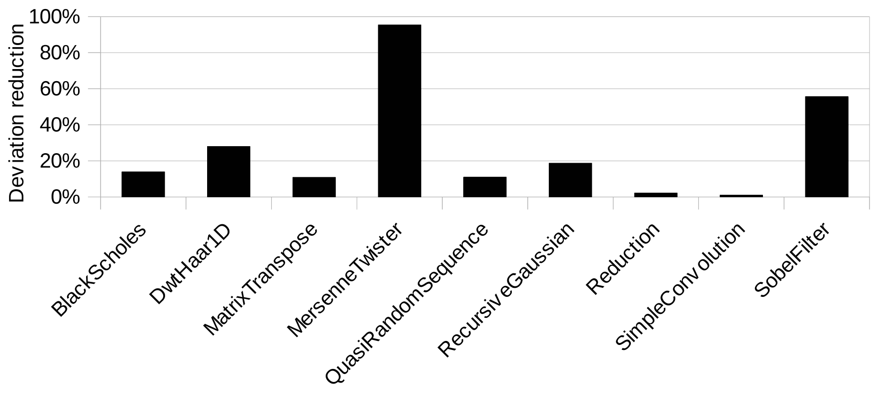


Figure 4.9: Reduction of execution time deviation between original and alltogether Multi2Sim.

For validation purposes we proceeded as follows. We measured the execution time that each benchmark lasts in the AMD Southern-Islands 7870HD GPU, in the original Multi2Sim simulation framework, and in our *all together* model. Then, we analyzed the deviation of the execution time gathered in Multi2Sim from the measured in the real GPU. Finally, we quantified how *all together* improves this deviation, bringing the simulated execution times closer to the real hardware. Figure 4.9 shows the results (in percentage). As observed, with the exception of `Reduction` and `SimpreConvolution`, the *all together* model improves (i.e. reduces) the Multi2Sim deviation in the range between 12% and 96%.

We analyzed the contribution of each component to the *all together* accuracy and found that the SI protocol is the component that most contributes, on average, to the overall accuracy. It shows the major contribution to the accuracy in most of the benchmarks with respect to the original Multi2Sim simulation framework. Examples of benchmarks showing this behavior are MersenneTwister and QuasiRandomSequence, where the contribution of this component represents nearly the total amount of the accuracy achieved by the all together configuration.

4.8 Conclusions

In this work we have shown that accurately modeling the memory subsystem in a current state-of-the-art simulator should be done in order to obtain representative results.

We have identified four main components of the on-chip and off-chip memory hierarchy presenting a significant impact on the performance of current GPUs. The identified components are: i) the MSHR file, ii) the memory controller and GDDR DRAM modules, iii) coalescing mechanisms, and iv) the coherence protocol.

To evaluate the impact of each of them we have extended the state-of-the-art Multi2Sim simulation framework. Below we draw the main conclusions for each studied component. First, modeling the MSHR file can introduce important performance drops over an unbounded MSHR file. For instance, a small file can reduce the performance in a factor of 5×. Second, the number of memory controllers and physical channels can reduce the performance over a fixed memory latency; in addition, the results widely vary depending on the assumed memory controller. For instance, modeling a single memory controller can strangle the performance. Third, coalescing can bring important performance differences over merging in some applications, since the number of L1 accesses can widely vary. Fourth, we have compared two state-of-the-art GPU protocols and we have found that the simple SI protocol, almost doubles the performance in some applications over the much complex NMOESI protocol.

Finally, we have compared the accuracy of the proposed extensions and the original Multi2Sim with respect to the AMD Southern-Islands 7870HD GPU. Experimental results show that our implementation achieves a significant accuracy enhancement over the original simulator.

Chapter 5

Improving GPU Cache Hierarchy Performance with a Fetch and Replacement Cache

- **Authors:** Francisco Candel, Salvador Petit, Alejandro Valero, Julio Sahuquillo
- **Type:** Conference
- **Conference:** 24th International European Conference on Parallel and Distributed Computing (EURO-PAR)
- **Location:** Turin, Italy
- **Year:** 2018
- **DOI:** 10.1007/978-3-319-96983-1_17

5.1 Abstract

In the last few years, GPGPU computing has become one of the most popular computing paradigms in high-performance computers due to its excellent performance to power ratio. The memory requirements of GPGPU applications widely differ from the requirements of CPU counterparts. The amount of memory accesses is several orders of magnitude higher in GPU applications than in CPU applications, and they present disparate access patterns. Because of this fact, large and highly associative Last-Level Caches (LLCs) bring much lower performance gains in GPUs than in CPUs.

This paper presents a novel approach to manage LLC misses that efficiently improves LLC hit ratio, memory-level parallelism, and miss latencies in GPU systems. The proposed approach leverages a small additional Fetch and Replacement Cache (FRC) that stores control and coherence information of incoming blocks until they are fetched from main memory. Then, fetched blocks are swapped with victim blocks to be replaced in the LLC. After that, the eviction of victim blocks is performed from the FRC. This management approach improves performance due to three main reasons: i) the lifetime of blocks being replaced is increased, ii) the main memory path is unclogged on long bursts of LLC misses, and iii) the average L2 miss delaying latency is reduced. Experimental results show that our proposal increases the performance (OPC) over 25% in most of the studied applications, reaching improvements up to 150% in some applications.

5.2 Introduction

In recent years, GPU (Graphics Processing Unit) architectures have acquired a great relevance in the field of high-performance computing. The main reason has been that GPUs are able to accelerate the execution of massively parallel applications, since they provide a much higher level of parallelism than CPU architectures. In addition, GPUs are energetically more efficient [29, 27] for a given performance, than its CPU counterparts. Because of these reasons, many supercomputers in the top 500 list [71] rely on GPUs. For instance, the Piz Daint supercomputer, ranked in third place of the list in November 2017, was built with Nvidia Tesla P100 GPU devices.

GPU architectures are optimized to run applications composed of thousands of logical threads. In order to support the execution of such a high number of threads, the GPU core must be coupled with a memory subsystem able to support a high Memory-Level Parallelism (MLP). GPU memory subsystems are therefore designed to sustain a high memory bandwidth. Because of the poor data temporal locality of GPGPU applications or kernels, on a *very long* burst of L2 accesses many requests can miss, which cause subsequent main memory accesses.

In this scenario, the memory subsystem of GPUs poorly performs. In this paper, we look into the reasons explaining this behavior, and we find that one of the main sources of performance losses of the memory subsystem is the management of L2 cache misses. We find that conventional caches designed to address memory patterns of CPU applications do not properly meet the requirements of GPGPU applications, but they seriously penalize their performance since they can significantly slow down the management of L2 cache requests on long bursts of requests. The previous rationale means that improving the L2 cache management is a key design concern that should be tackled to improve the system performance. This paper proposes a novel L2 cache design aimed at boosting the memory level parallelism by adding a Fetch and Replacement Cache (FRC) that provides additional cache lines that help unclog the memory subsystem. The FRC approach uses these extra resources to prioritize the fetch of incoming L2 cache requests and to delay the eviction of the blocks to be replaced. The proposal has been evaluated considering an AMD GPU based architecture, although the results would also apply in almost all current GPU architectures as they implement a similar memory hierarchy.

The proposal has been modeled in the Multi2Sim simulation framework [72], a state-of-the-art GPU simulator widely used in both the academia and the industry. Experimental results show that FRC improves the Operations Per cycle (OPC) more than 25% in most applications by drastically reducing the Misses Per Kilo-Operation (MPKO) and L2 miss latency.

The remainder of this work is organized as follows. Section 2 describes the architecture of the AMD *Southern Islands* family of GPUs. Section 3 motivates this work by presenting the problems that FRC tackles in current GPU memory subsystems. In Section 4, the proposed approach is described in detail. Section 5 presents the experimental results. Section 6 summarizes related studies about GPU memory subsystems. Finally, in Section 7 some concluding remarks are drawn.

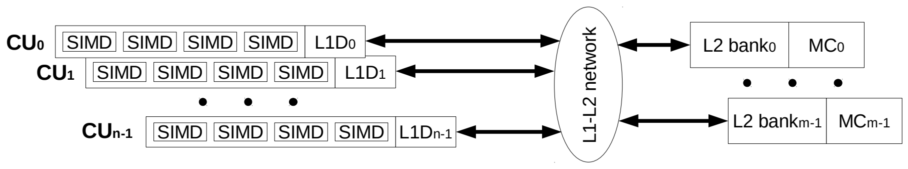


Figure 5.1: Diagram of an AMD Southern Islands GPU.

5.3 Background

This section provides some background about the architecture of modern GPUs. Since this paper focuses on the AMD *Southern Islands* [8] family of GPUs, AMD terminology is used throughout this work.

Figure 6.1 depicts a block diagram of an AMD Southern Islands GPU. This GPU includes up to 32 *Compute Units* (CUs), each one implementing the *Graphics Core Next* (GCN) [68] microarchitecture. Internally, a GCN CU consists of 4 *Single Instruction Multiple Data* (SIMD) arithmetic logic units.

GPU applications or *kernels* are composed of a massive number of threads or *work-items*. These threads are organized in 64-thread bundles, named *wavefronts*, which are allocated to SIMD units. During most of the execution time of a kernel, the GPU ensures that each SIMD unit is assigned tens of wavefronts. In this way, SIMD units can switch among wavefronts in a fine-grain basis, which helps hide memory latencies.

A SIMD unit executes instructions from threads of a wavefront in a lockstep manner. That is, at a given point of the execution time a SIMD unit is performing the same arithmetic instruction in the 64 threads of the same wavefront. Memory reference instructions are also executed following the SIMD paradigm; that is, a wavefront can generate up to 64 memory requests at the same time. To reduce the overall amount of memory requests, those referencing the same 64-byte cache block are *coalesced* into a single memory request, which is issued to the memory subsystem.

As in a conventional processor, the memory subsystem is organized hierarchically. After being coalesced, memory requests access the L1 data cache of the corresponding CU. Those requests that miss the L1 cache are forwarded to a multi-banked L2 cache, acting as Last-Level Cache (LLC). L2 banks contain interleaved block addresses at a granularity of 256 bytes, and each bank is connected to a dual-channel memory controller that manages the corresponding off-chip

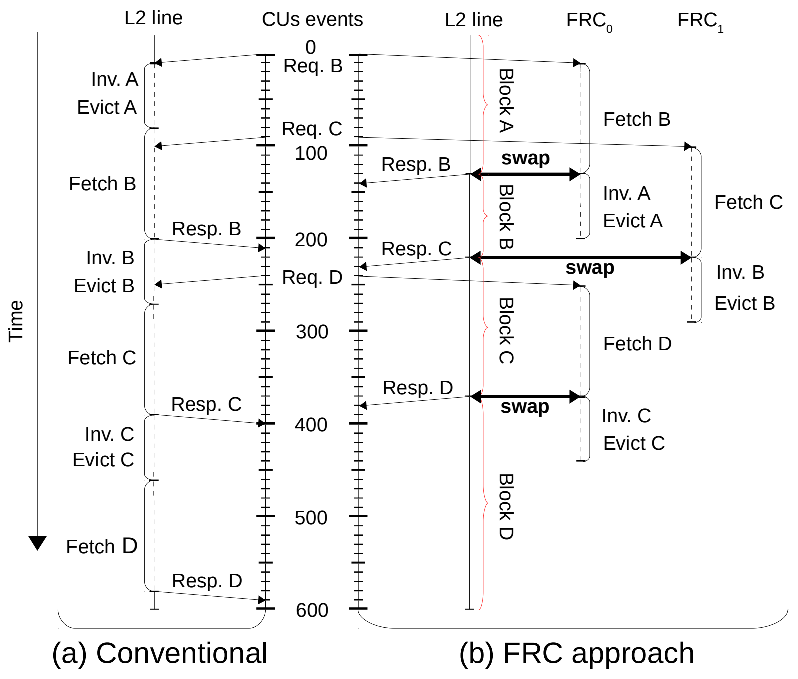


Figure 5.2: Sequence of events involved in three consecutive replacements targeting the same L2 cache line for both the conventional and the proposed approaches.

GDDR5 main memory. This design reduces the number of channel conflicts and increases the memory bandwidth utilization.

5.4 Motivation

The coalesce mechanism reduces the number of requests to the memory subsystem. However, GPGPU applications generate enormous amounts of memory traffic; for instance, a typical GPU can issue thousands of memory requests in a given cycle. These amounts yield conventional cache organizations to significant performance losses. The main reason is that the massive number of threads is executing in parallel causes sudden bursts of memory transactions, which involve a high number of cache replacements. As a consequence, in a relatively short interval of time, a given cache line can suffer a long number (e.g. in the order of tens) of consecutive block replacements, each one involving different actions such as coherence invalidations or accesses to lower levels of the memory hierarchy. Since these actions are serialized at the cache line, the management of cache replacements becomes a major performance bottleneck, which can heavily reduce the MLP and the L2 hit ratio.

To help understand the problem, Figure 6.2 depicts a time diagram with the events involved in three consecutive replacements all targeting the same L2 victim line. The three requests causing these replacements have been labeled as *Req. B*, *C*, and *D*, and have been generated at cycles 0, 90, and 240, respectively, after the requests miss the L1 cache and are forwarded to the L2 cache.

As can be seen in Figure 6.2a, which shows the behavior of a conventional replacement approach, *Req. B* triggers the replacement of the currently stored block (block *A*). From this moment, the victim line is in a transient state (represented by dashed lines), preventing other requests from accessing the line. To manage the replacement, depending on the state of *A*, an invalidation to the L1 cache and an L2 cache eviction must be performed. Once the victim line is freed, the requested incoming block (*B*), must be fetched from main memory and allocated to this line.

While block *B* is being fetched, *Req. C* arrives to L2, which triggers another replacement in the same victim line. However, because of the line is in a transient state, *Req. C* must be enqueued. Thus, *Req. C* cannot be attended until cycle 210, delaying its completion until cycle 400. This serialization also affects *Req. D* at cycle 240.

Moreover, the hit ratio is also reduced, since i) the invalidation and eviction of the contents of a victim line are performed before fetching the requested block and ii) the fetch operation is the longest one involved in a replacement due to the high main memory latencies. As an example, even if a complex protocol allows reading the contents of a cache line while it is in a transient state, a load requesting block *A* would only hit between cycles 0 and 90, and would miss afterwards.

Although theoretically possible, it is very rare that this situation occurs in a conventional CPU processor since there is likely a non-transient line in the same cache set that can be selected as a victim, which avoids the serialization of replacements. In contrast, in GPUs, it is often the case that a burst of misses triggers replacements in all the lines of the same cache set. Therefore, further misses targeting the same set cannot be served from memory, which impacts on the exploited memory parallelism.

A naive solution to this problem is blindly increasing cache associativity so that a set has more available lines. However, this approach incurs in high latencies and energy penalties since associative tag lookups do not scale well with the number of ways. Moreover, although such a solution may alleviate the problem, larger sets can also be blocked provided that bursts of misses affecting the same cache set are large enough.

5.5 FRC Approach

The proposed approach is aimed at increasing MLP and LLC hit ratio. With this aim, we introduce a Fetch and Replacement Cache (FRC) to each L2 cache bank. The FRC provides additional cache lines that allow i) start fetching from memory as soon as an L2 miss rises, increasing MLP, and ii) performing invalidation and eviction actions *after* fetching the requested block, which increases the lifetime of victim blocks and the overall hit ratio.

Figure 6.2b shows how the FRC can help improve the management of consecutive replacements affecting the same line. By cycle 10, when *Req. B* misses in L2, instead of immediately invalidating the victim line, a free FRC entry (FRC_0) is allocated and used to fetch block B. After this block is fetched, the contents of the victim line and FRC_0 are swapped. Then, the invalidation and eviction of block A are performed from FRC_0, which becomes free when the eviction is completed. In this way, fetch actions can be performed as long as there are free FRC entries (e.g. the fetch of block C can start in parallel at cycle 90). To ensure that there are free FRC entries, they are recycled. Thus, after block A has been replaced, FRC_0 is freed, which allows this entry to be used later by *Req. D*.

The swap operation guarantees that the victim line is never in a transient state (note that it is not represented with dashed lines in Figure 6.2b), and that the invalidation and eviction of its contents are performed after the requested block is fetched. Consequently, FRC supports a higher cache level parallelism that allows responding to several requests at the same time. Furthermore, compared to the conventional approach, the lifetime of the victim block becomes longer when FRC is used.

Tags and control bits of blocks in transient state are stored in the FRC. Thus, to reduce tag lookup overhead, FRC is organized as a conventional cache, although its geometry (i.e. associativity and number of sets) can be different from that of the L2 cache. L2 accesses must search the requested block both in the target L2 bank and its associated FRC. A hit in the L2 bank is performed as in the conventional approach, while a hit in FRC for a block being fetched is enqueued until the fetch operation completes.

As shown in Figure 6.4, the FRC approach modifies the classical miss management by adding the events highlighted in gray color. On an L2 miss (both in the L2 bank and the FRC), and if there are free entries in the FRC's set mapped to the missing block, the block is assigned to a FRC 's entry and the access is immediately propagated to the lower memory hierarchy

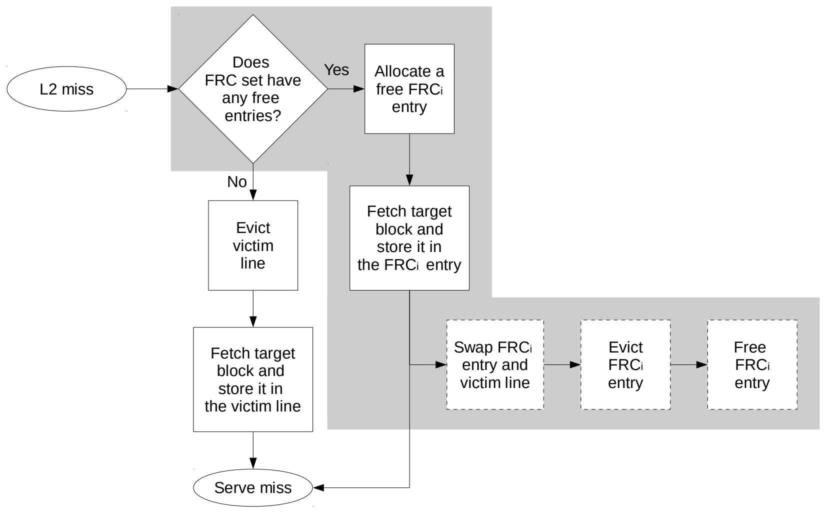

L2 miss
Does FRC set have any free entries?
Yes
No
Allocate a free FRC$_i$ entry
Fetch target block and store it in the FRC$_i$ entry
Evict victim line
Fetch target block and store it in the victim line
Swap FRC$_i$ entry and victim line
Evict FRC$_i$ entry
Free FRC$_i$ entry
Serve miss

Figure 5.3: Block diagram with the steps followed on an L2 miss. Those steps introduced with the FRC are highlighted in gray color.

level (early fetch). Once the fetch has been performed, the miss can be already served. In this way, the victim block eviction is taken out of the critical path. To manage the eviction without leaving L2 cache lines in a transient state, the data stored in the FRC's entry and the victim line are swapped. Thereby, the eviction is done from the FRC's entry. Once the eviction has finished, the FRC's entry is set as free to handle subsequent L2 misses.

Finally, note that in case there is not any free entry in the FRC's set targeted by the missing block, the proposed approach operates like the conventional approach. In addition, FRC does not change the state of blocks stored in the cache, but only modifies the resources they are using. Thus, it does not affect the coherence protocol.

Overall, as experimental results will show, FRC has three main impacts on performance: i) new requests do not wait (or wait much less) for cache block's evictions, which reduces the memory access latency, ii) the lifetime of an L2 block becomes longer, decreasing the number of misses, and iii) a higher MLP is achieved, since FRC allows immediate access to lower memory levels as long as there are free FRC entries.

5.6 Experimental Evaluation

To evaluate the proposal, we have modeled the FRC approach with the Multi2Sim [72] simulation framework. We focus on the Southern Islands GPU architecture from AMD, which is one of the most recent GPU architectures modeled on a detailed simulation framework. In particular, we model the characteristics of an HD7770 GPU [68], including CUs, L1 and L2 caches, memory controllers, and GDDR5 memory [17]. The L2 cache consists of two 16-way 128KB banks, which is our baseline configuration. In addition, to evaluate the impact on performance of cache associativity and capacity, we evaluate two additional conventional L2 caches consisting of two 32-way 256KB banks and two 32-way 512KB banks. Both configurations are compared to the FRC one, which is composed of the baseline configuration plus two additional FRCs (1 per bank). We analyze the sensitivity our proposal to the number of FRC entries, which ranges between 4 and 512. All the evaluated FRC configurations, except the smallest one with 4 entries, are organized with 8-way sets.

Notice that the FRC approach represents a minor area increase over the baseline, since the area occupied by an additional FRC is much smaller than doubling or quadrupling the cache bank capacity, which would present roughly the same cost in area as adding 2048 and 6144 entries, respectively. Nevertheless, we conservatively assume that all the analyzed L2 cache configurations have the same access time.

For evaluation purposes, a subset of the OpenCL SDK 2.5 benchmarks [3] has been used, covering all the possible performance behaviors from the entire benchmark suite. These benchmarks are executed until completion.

5.6.1 Performance Analysis

System performance has been quantified in terms of Operations Per Cycle (OPC), which is analogous to its counterpart IPC used to evaluate CPU processors [17]. This metric accounts for the number of single scalar operations each GPU instruction executes during the workload execution. For instance, if a given vector instruction is internally executed as 64 individual scalar operations, this metric accounts for 64 operations instead of only one instruction.

Figure 6.6 shows the OPC for the studied benchmarks. The red bar on the left side of each plot represents the 2×128KB L2 baseline cache, and the two red bars on the right side represent the 2×256KB L2 cache and the 2×512KB L2 cache, respectively. The black bars show

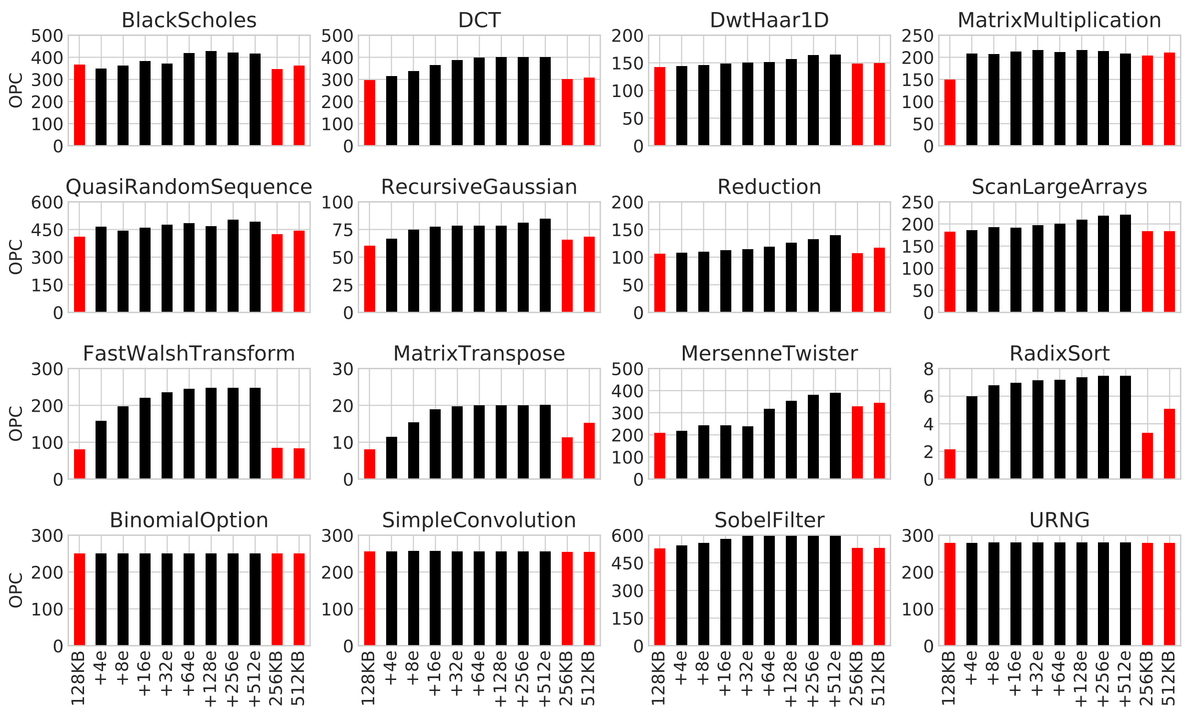


Figure 5.4: Operations Per Cycle (OPC) across the studied applications.

results of the FRC configuration varying the number of entries per FRC ranging from 4 to 512, labeled as *+Ne*, where *N* indicates the number of entries. The proposed approach achieves, across most of the studied applications, OPC improvements higher than 25% compared to the baseline, reaching improvements up to 150% in applications such as `FastWalshTransform` and `MersenneTwister`. In general, it can be observed that almost all the applications achieve their highest OPC with around 32 or 64 entries, which represents by 64× and 32× less area, respectively, than doubling the cache bank size to 256KB. Moreover, in most applications, the performance achieved by FRC is much higher than that obtained by blindly increasing the L2 cache capacity with a higher associativity degree.

Three main behaviors can be appreciated:

- Smooth OPC increase. The OPC of applications exhibiting this behavior, which is the common one, increases in small steps with additional FRC entries until a given saturation point. This is the case of benchmarks such as `FastWalshTransform`, `MersenneTwister`, and `DCT`.

- Sharp OPC increase. Applications presenting this behavior show significant performance increase with just 4 FRC entries, but no remarkable OPC improvement is observed with additional entries. This is the case of `MatrixMultiplication`.

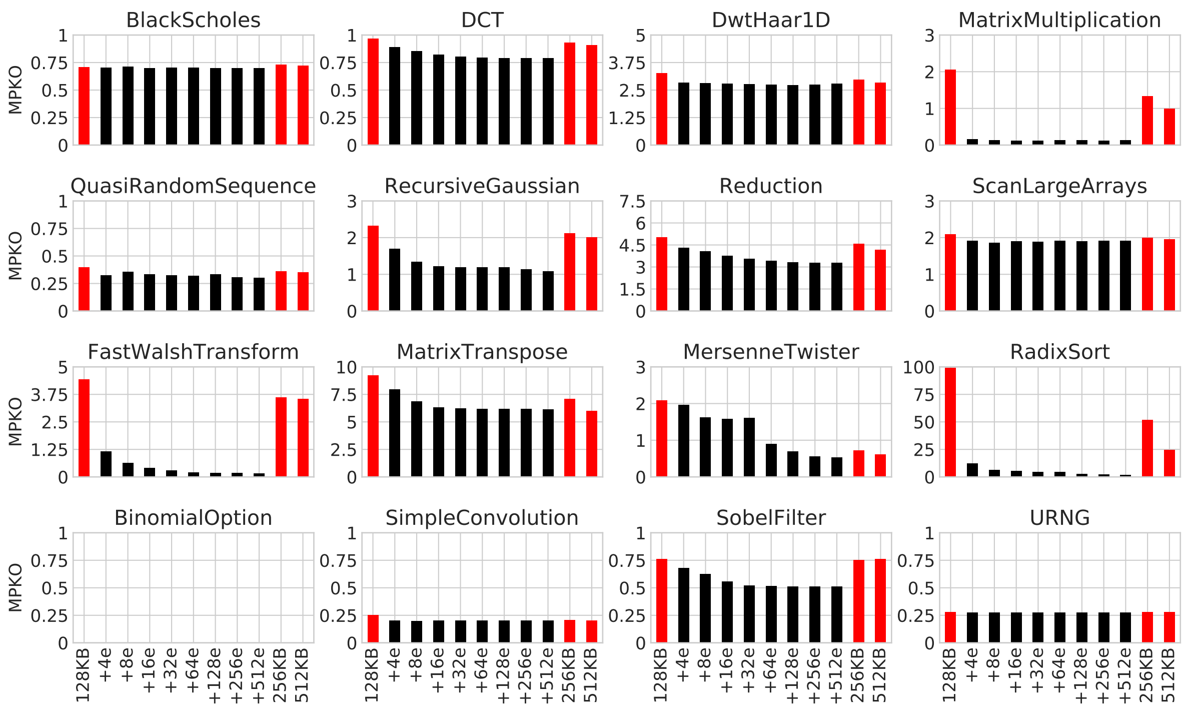


Figure 5.5: Misses Per Kilo-Operation (MPKO) in the L2 cache.

- Similar OPC. Applications in this category experience the same performance across all the studied cache approaches. This is the case of `BinomialOption` and `URNG`, mainly due to their low number of memory accesses as discussed below. Obviously, the OPC of this type of applications is also not affected when enlarging the L2 cache size and associativity.

5.6.2 Analysis of Memory Subsystem Metrics

To provide insights into the OPC trend shown by the studied applications, we analyze the following metrics: number of misses measured in *Misses Per Kilo-Operation* (MPKO), percentage of misses served by FRC additional entries, and the L2 miss latency penalty.

Misses Per Kilo-Operation.

We define the metric MPKO for GPUs with analogous meaning to the MPKI (Misses Per Kilo-Instruction), widely used when studying the cache hierarchy of the CPU counterparts. Figure 6.7 plots the results. It can be observed that the baseline configuration shows high MPKO values, which can be notably reduced by adding FRC entries. This fact confirms the benefits on performance brought by the FRC approach by keeping victim blocks in a non-transient state

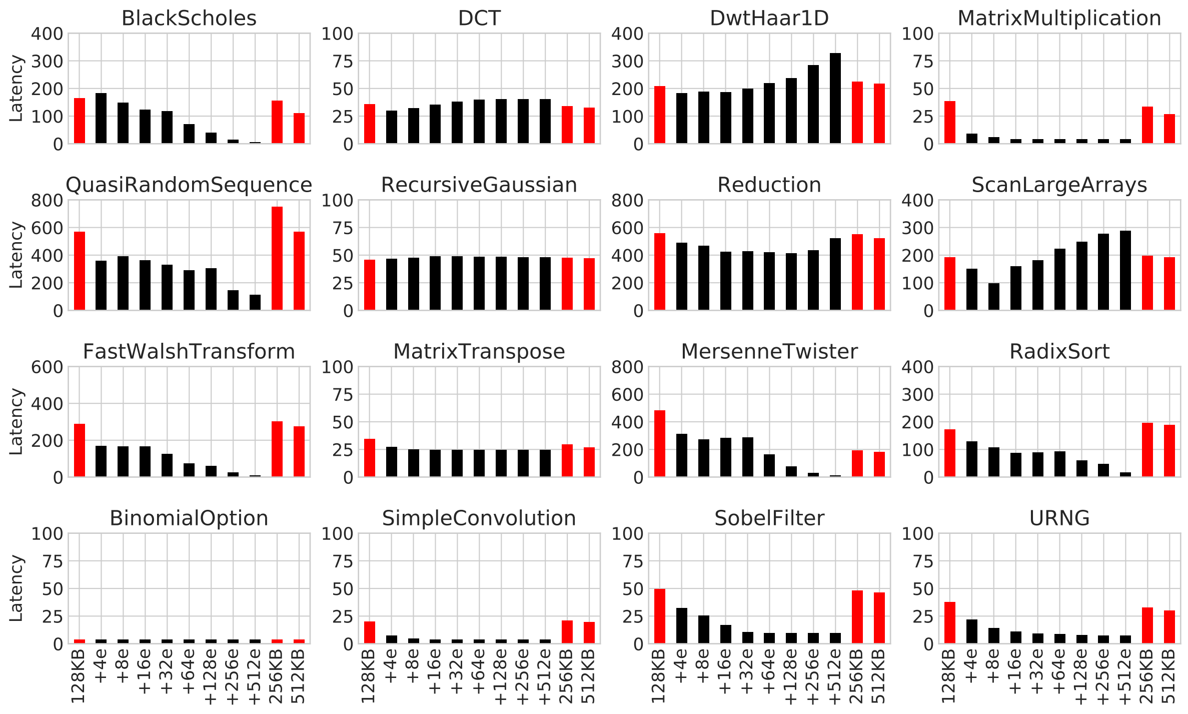


Figure 5.6: Average L2 miss delaying latency quantified in processor cycles.

until fetch actions are completed. As a consequence, the hit ratio is improved compared to the conventional approach.

Overall, a clear inverse correlation between OPC and MPKO can be appreciated. However, in a few applications like `DwtHaar1D` and `Reduction`, a significant MPKO reduction over the baseline with a few FRC entries has a minimal effect on OPC. On the other hand, as observed, `BinomialOption` and `URNG` present a near-zero MPKO, meaning that no OPC gains can be achieved in these applications by acting on the L2 cache. However, there are applications like `BlackScholes`, `DCT`, `QuasiRandomSequence`, and `SobelFilter`, with a relatively low MPKO (below 1.5) in the baseline which improve their OPC with an FRC. In order to explain these behaviors, the MLP and memory latency are analyzed below.

L2 Miss Latency.

L2 cache misses can be handled either by normal cache entries or by FRC entries. Misses handled by FRC entries can be considered as *fast* L2 misses since, as explained in Section 6.5, they are able to access to main memory with a minimum delay. In other words, the more misses handled by FRC entries the better the performance. Figure 6.8 plots the results of the L2 miss latency (excluding the actual main memory access time), quantified in processor cycles.

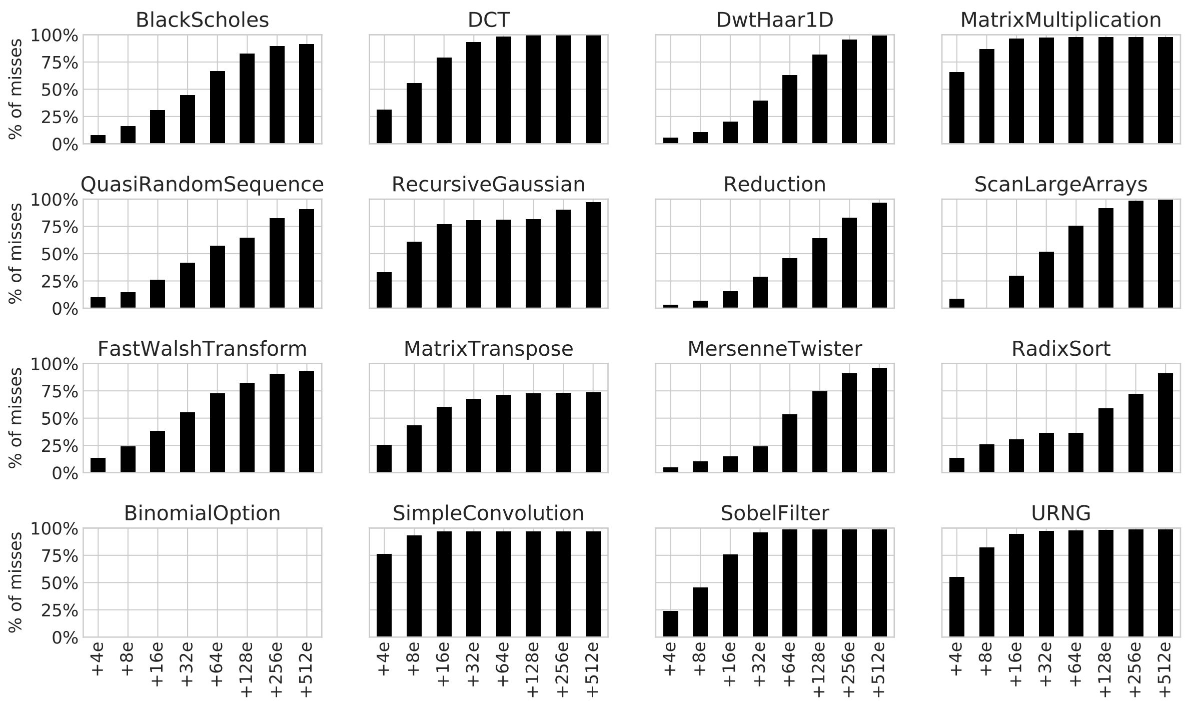


Figure 5.7: Percentage of L2 misses handled by FRC entries.

The use of FRC entries reduce the average L2 miss latency for almost all the applications. As observed, with just 4 FRC entries, latency is largely reduced with respect to the 256KB and 512KB cache configurations. In fact, the largest FRC configuration completely reduces the L2 contention in most benchmarks. Nevertheless, it can be seen that just 4 FRC entries only provide a slight latency improvement in some applications, thus large-sized FRCs are preferred. However, `DwtHaar1D` and `ScanLargeArrays` suffer an increase in latency as the number of FRC entries grows over around 8 entries. This is because the parallelism level is higher than the baseline, which increases the memory contention. Notice that, in spite of this increase, the higher MLP turns into OPC improvements.

Percentage of Misses Served by the FRC.

Since the service of misses is not stalled in case of consecutive replacements over the same victim line, MLP is also improved. Figure 5.7 shows the percentage of misses served by the FRC. As observed, FRC with only 64 entries handles by 75% of misses in most applications. Moreover, this percentage significantly rises, even to almost 100% in some benchmarks, for configurations smaller than the $+512e$ configuration.

The applications `Matrixtranspose` and `BinomialOption` show an unexpected behavior as the percentage of misses handled by FRC entries saturate in a relatively low number of entries, that is, this percentage does not increase even if more entries are added. In other words, the L2 cache misses are mostly handled by the cache itself instead of by FRC entries. This is due to two different reasons. First, the kernel of `Matrixtranspose` presents bursts of accesses targeting the same FRC set. This behavior can be improved by increasing FRC associativity (8-way in these experiments). Second, `BinomialOption` makes important use of the local memory of the CU, which significantly reduces the number of accesses to main memory.

5.7 Related Work

The GPU memory subsystem performance has been widely analyzed in recent years from different angles, including memory scheduling strategies [48, 32, 64], cache bypassing techniques [41, 44], and optimizing the memory subsystem design [40, 25, 47, 74, 63]. This section summarizes prior work in this regard.

Elastic-Cache [40] supports fine-grained L1 cache line management for those kernels with irregular memory access patterns that do not efficiently exploit cache space. Auxiliary tags for fine-grained cache line management are stored in unused shared memory space, which is not fully occupied in many kernels.

Gebhart et al. [25] propose to dynamically adjust the storage partitioning among registers, primary caches, and scratchpads depending on the kernel memory requirements, resulting in a reduction of the on-chip access latencies.

IBOM [47] is an integrated architecture that leverages unused register file entries with lightweight ISA support to enlarge the L1 cache size. With enough cache capacity, a set balancing technique exploits underutilized sets to improve cache usage.

Other works have proposed additional memory structures to improve GPU performance. Wang et al. [74] incorporate a victim cache between L1 and L2 that presents the same capacity and associativity as the L1 cache. Reused blocks are kept in the L1 cache by enabling swap operations with the victim cache. Since a victim cache so large would impact on energy and area, unused entries from the register file and shared memory are proposed as an alternative to holding data that otherwise would remain in the victim cache.

In [63], the authors propose to allocate *TinyCaches* between each lane in a CU and the L1 cache to filter out memory requests to lower memory levels for energy saving purposes. By leveraging intrinsic characteristics of CUDA and OpenCL programming models, these caches are kept non-coherent to avoid incurring additional overheads.

All the above works primarily focus on L1 caches. In contrast, our proposed FRC design targets LLCs where all accesses from L1 are merged and contention greatly limits MLP. Furthermore, the FRC approach can be easily implemented in different memory subsystem architectures, since it does not change the actions required to handle misses, but the locations where these actions are performed (i.e. FRC entries).

5.8 Conclusions

This paper has presented a novel GPU cache subsystem design that leverages a small Fetch and Replacement Cache (FRC) between the Last-Level Cache (LLC) and the main memory. The design provides additional cache lines that allow prioritizing the fetch of incoming LLC cache blocks over the replacement of victim blocks. The proposed design boosts the system performance by increasing the Memory-Level Parallelism (MLP) and enlarging the lifetime of the victimized blocks.

FRC attacks by design three main cache performance related events, which results in a much better L2 cache management: i) it reduces the number of Misses Per Kilo-Operation (MPKO) by keeping victim blocks in cache until fetch actions are completed, ii) it reduces the miss latency by starting the fetch actions from main memory as soon as a miss rises, and iii) it increases the MLP by unclogging new block requests whose victim line is already being replaced.

Experimental results have shown that, compared to a conventional LLC design, FRC increases the Operations Per Cycle (OPC) over 25% in all the applications suffering contention in main memory.

Chapter 6

Efficient Management of Cache Accesses to Boost GPGPU Memory Subsystem Performance

- **Authors:** Francisco Candel, Alejandro Valero, Salvador Petit, Julio Sahuquillo
- **Type:** Journal
- **Journal:** IEEE transactions on Computers (TC)
- **Year:** 2019
- **DOI:** 10.1109/TC.2019.2907591

6.1 Abstract

To support the massive amount of memory accesses that GPGPU applications generate, GPU memory hierarchies are becoming more and more complex, and the Last Level Cache (LLC) size considerably increases each GPU generation. This paper shows that counter-intuitively, enlarging the LLC brings marginal performance gains in most applications. In other words, increasing the LLC size does not scale neither in performance nor energy consumption. We examine how LLC misses are managed in typical GPUs, and we find that in most cases the way LLC misses are managed are precisely the main performance limiter. This paper proposes a novel approach that addresses this shortcoming by leveraging a tiny additional Fetch and Replacement Cache-like structure (FRC) that stores control and coherence information of the incoming blocks until they are fetched from main memory. Then, the fetched blocks are swapped with the victim blocks (i.e., selected to be replaced) in the LLC, and the eviction of such victim blocks is performed from the FRC. This approach improves performance due to three main reasons: i) the lifetime of blocks being replaced is enlarged, ii) the main memory path is unclogged on long bursts of LLC misses, and iii) the average LLC miss latency is reduced. The proposal improves the LLC hit ratio, memory-level parallelism, and reduces the miss latency compared to much larger conventional caches. Moreover, this is achieved with reduced energy consumption and with much less area requirements. Experimental results show that the proposed FRC cache scales in performance with the number of GPU compute units and the LLC size, since, depending on the FRC size, performance improves ranging from 30% to 67% for a modern baseline GPU card, and from 32% to 118% for a larger GPU. In addition, energy consumption is reduced on average from 49% to 57% for the larger GPU. These benefits come with a small area increase (by 7.3%) over the LLC baseline.

6.2 Introduction

Nowadays, GPU (Graphics Processing Unit) architectures have acquired a great relevance in the high-performance computing field. One of the main reasons has been that GPUs are energetically more efficient [27, 29] when running massively parallel applications, since they provide a much higher level of parallelism than their CPU counterparts with a much better performance to power ratio. In fact, many of the current most powerful and energy-efficient supercomputers, ranked in both the Top500 and Green500 lists [71], rely on GPUs.

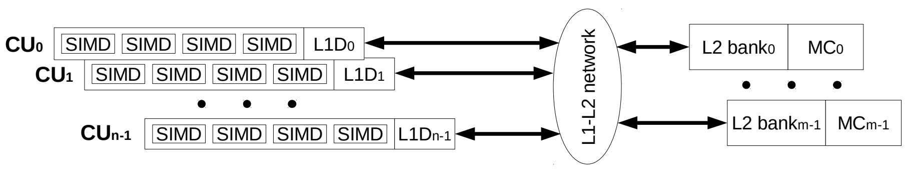


Figure 6.1: Diagram of an AMD Polaris GPU.

GPU architectures are optimized to run applications composed of thousands of logical threads. Given that these applications demand an ever-increasing amount of computational and memory resources, successive GPU architectures include more *multiprocessors* (i.e., compute units) and a larger on-chip memory subsystem. For instance, NVIDIA has continuously enlarged the Last-Level Cache (LLC) size in 2MB on recent architectures (e.g., LLC sizes of Maxwell [53], Pascal [54], and Volta [55] GPUs are 2MB, 4MB, and 6MB, respectively). Coupling GPUs with larger memory subsystems enables a higher Memory-Level Parallelism (MLP). However, because of the poor data temporal locality of GPU applications, upon a *fast and relatively very long* burst of LLC (i.e., L2) accesses it is likely that a significant number of accesses miss in the cache, requiring access to the off-chip memory, which can severely hurt the system performance.

A straightforward solution consists of drastically increasing the L2 cache size with the aim of accommodating the entire working set of the application on chip. Unfortunately, enlarging the L2 cache not only brings much lower performance gains in GPUs than in CPUs [20, 43], but also translates into a high area overhead as well as a huge static energy consumption, which aggravates as transistor size shrinks [34].

In this paper, we look into the reasons explaining the poor performance gains of GPU memory subsystems, and we find that a key aspect is the way L2 cache misses are managed in typical caches. In particular, a typical cache miss management gets clogged on fast, long bursts of cache misses, which increases memory latencies and limits MLP. Moreover, the lifetime of memory blocks is shortened, rising the amount of memory misses even for those applications with low temporal locality and low cache hit ratio.

The previous rationale means that the L2 cache management is a key design concern that should be tackled to improve the GPU performance. This paper proposes an energy-efficient L2 cache design aimed at boosting MLP by adding a tiny Fetch and Replacement Cache-like structure (FRC) that provides additional reusable cache lines that help unclog the memory subsystem. The proposed approach prioritizes the fetch of incoming L2 cache requests and delays the eviction of the blocks, which helps alleviate memory latencies and improve the cache hit ratio.

The proposal has been modeled and evaluated in both AMD Polaris [5] and Vega [7] GPU architectures, although the results would apply to most of the current GPU architectures provided that they implement a similar memory subsystem and organization. For instance, some NVIDIA L2 caches use a replacement algorithm other than LRU and a 32B line size [45]. However, these characteristics are orthogonal to our proposal. In particular, experiments consider two Polaris GPU cards, namely RX540 and RX570, with a different number of compute units and L2 cache sizes to show the scalability of FRC in terms of performance and energy, whereas a Vega64 card is also studied to show how the proposed FRC approach behaves with an enhanced memory subsystem using HBM technology and a higher clock frequency.

The proposal has been modeled using the state-of-the-art Multi2Sim [72] and CACTI [69] simulation frameworks, which are a cycle-accurate GPU simulator and an analytical model for both on-chip and off-chip memories, respectively, both widely used in the academia and the industry. Experimental results show that, compared to a conventional design, FRC improves the average system performance (OPC) of the RX540 between 30% and 67% depending on the FRC size, whereas these percentages rise up to 32% and 118%, respectively, for the larger RX570 GPU. Moreover, in most applications, the performance achieved by adding a small FRC is much higher than simply increasing the L2 cache capacity or associativity. In addition, compared to the conventional approach, energy savings fall in between 49% and 57% for the RX570 GPU. These benefits come with a small L2 cache area increase by 7.3% over the baseline. Finally, in spite of an improved memory subsystem with the Vega64 GPU, the FRC approach still boosts the average OPC from 16% to 54%.

This paper extends the work in [16] in four main ways: i) a hardware implementation for FRC is presented, ii) FRC has been modeled and evaluated on the recent AMD Polaris and Vega GPU architectures, iii) performance scalability has been studied by analyzing how FRC behaves with an increasing number of compute units and L2 cache sizes, and iv) energy consumption and area results are discussed.

The remainder of this work is organized as follows. Section 2 describes the architecture of the AMD Polaris family of GPUs. Section 3 motivates this work. In Section 4, the proposed approach is introduced. Section 5 presents the experimental results. Section 6 summarizes related studies about GPU memory subsystems. Finally, Section 7 summarizes the paper.

6.3 Background

This section provides some background of the architecture of modern GPUs. Since this paper primarily uses the AMD Polaris family of GPUs as a driving example, the AMD terminology is used throughout this work.

Figure 6.1 depicts a block diagram of an AMD Polaris GPU. This GPU includes up to 36 Compute Units (CUs), each one implementing the 4th version of the *Graphics Core Next* (GCN) [68] microarchitecture. Internally, a GCN CU consists of 4 *Single Instruction Multiple Data* (SIMD) arithmetic logic units.

GPU applications or *kernels* are composed of a massive number of threads or *work-items*. These threads are organized in 64-thread bundles, named *wavefronts*, which are allocated to SIMD units. During most of the execution time of a kernel, the GPU ensures that each SIMD unit is assigned tens of wavefronts. In this way, SIMD units can switch among wavefronts in a fine-grain basis, which helps hide memory latencies.

A SIMD unit executes instructions from threads of a wavefront in a lockstep manner. That is, at a given point of the execution time a SIMD unit is performing the same arithmetic instruction in the 64 threads of the same wavefront. Memory reference instructions are also executed following the SIMD paradigm; that is, a wavefront can generate up to 64 memory requests at the same time. To reduce the overall amount of memory requests, those referencing the same 64-byte cache block are *coalesced* into a single memory request, which is issued to the memory subsystem.

As in a conventional processor, the memory subsystem is organized hierarchically. After being coalesced, memory requests access the L1 data cache of the corresponding CU. Those requests that miss the L1 cache are forwarded to a multi-banked L2 cache, acting as the LLC. L2 banks contain interleaved block addresses at a granularity of 256 bytes, and each bank is connected to a dual-channel memory controller (MC) that manages the corresponding off-chip GDDR5 main memory. This design reduces the number of channel conflicts and increases the memory bandwidth utilization.

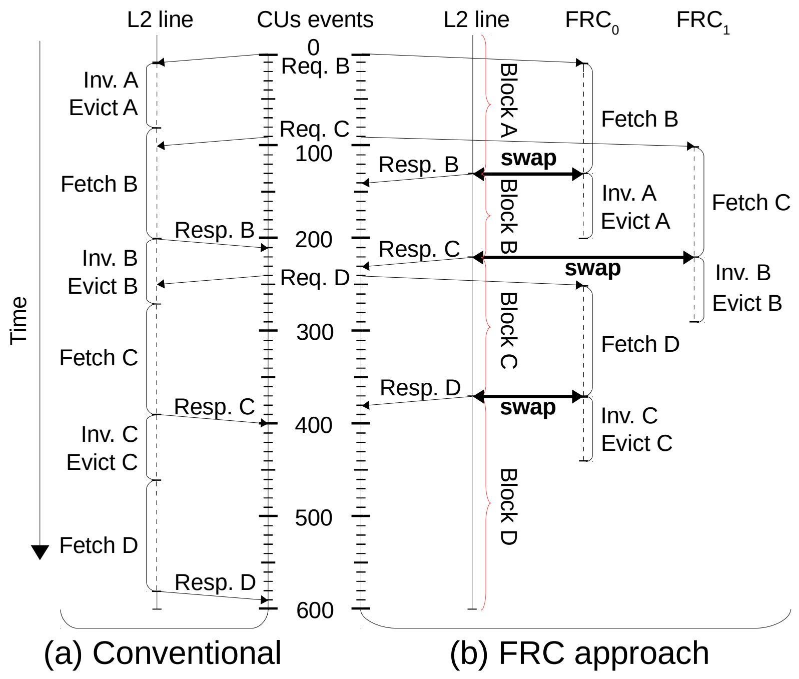


Figure 6.2: Sequence of events involved in three consecutive replacements targeting the same L2 cache line for both the conventional and the proposed approaches.

6.4 Motivation

6.4.1 Conventional Cache Miss Management

The coalesce mechanism reduces the number of requests to the memory subsystem. However, GPGPU applications generate enormous amounts of memory traffic; for instance, a typical GPU can issue thousands of memory requests in a given cycle. These amounts yield conventional cache organizations to significant performance losses. The main reason is that the massive number of threads executing in parallel causes sudden bursts of memory accesses, which involve a high number of cache replacements. As a consequence, in a relatively short interval of time, a relatively high number of cache lines can suffer a long number (e.g., in the order of tens) of consecutive block replacements, each one involving different actions such as coherence invalidations or accesses to lower levels of the memory hierarchy. Since these actions are serialized at each cache line, the management of cache replacements becomes a major performance bottleneck, which can heavily increase memory latencies and reduce the MLP and the L2 hit ratio.

To help understand the problem, Figure 6.2 plots a time diagram with the events involved in three consecutive replacements, all of them targeting the same L2 line. The three requests causing these replacements have been labeled as *Req. B*, *C*, and *D*, and have been generated at cycles 0, 90, and 240, respectively, after the requests miss the L1 cache and are forwarded to the L2 cache.

As can be seen in Figure 6.2a, which shows the behavior of a conventional replacement approach, *Req. B* triggers the replacement of the currently stored block (block *A*). From this point forward, the line storing the victim block is in a transient state (represented in dashed lines), preventing other requests from accessing the line. To manage the replacement, depending on the state of *A*, an invalidation to the L1 cache and an L2 cache eviction should be performed. Once the line is released, the requested incoming block (*B*), must be fetched from main memory and written in that line.

While block *B* is being fetched, *Req. C* arrives to L2, which triggers another replacement in the same line. However, because the line is in a transient state, *Req. C* must be enqueued. Thus, *Req. C* cannot be handled until cycle 210, delaying its completion until cycle 400. This serialization also affects *Req. D* at cycle 240.

Delaying requests increases memory latencies and reduces MLP. Moreover, the hit ratio is also reduced, since i) the invalidation and eviction of the victim block are performed before fetching the requested block, and ii) the fetch operation is the longest one involved in a replacement due to the high main memory latencies. As an example, even if a complex protocol allows reading the contents of a cache line while it is in a transient state, a memory instruction accessing to block *A* would only hit between cycles 0 and 90, and would miss afterward.

6.4.2 A Novel Cache Miss Management Approach

The proposed approach is aimed at improving MLP and LLC hit ratio while reducing miss latencies. With these aims, we implement a Fetch and Replacement Cache (FRC) in each L2 cache bank. The FRC provides additional cache lines that allow to i) start fetching from memory as soon as an L2 miss is detected, which reduces the miss latency and increases the MLP, and ii) delaying invalidation and eviction actions *until* the requested block is fetched, which enlarges the lifetime of victim blocks and the overall hit ratio.

Figure 6.2b shows how the FRC can help improve the management of consecutive replacements in the same line. By cycle 10, when *Req. B* misses in the L2 cache, instead of invalidating the

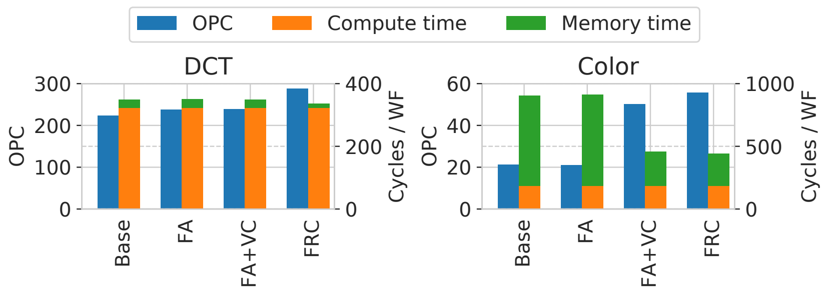


Figure 6.3: Operations Per Cycle (left Y-axis) and average execution cycles split in compute and memory cycles (right Y-axis) for the studied approaches.

victim block (i.e., block A), a free FRC entry (FRC_0) is allocated and used to fetch block B. After this block is fetched, the contents of the line storing block A and FRC_0 are swapped. Then, the invalidation and eviction of block A are performed from FRC_0, which is freed when the eviction is completed. In this way, fetch actions can be immediately start as long as there are free FRC entries (e.g., the fetch of block C can start in parallel at cycle 90). To ensure that there are free FRC entries, they are recycled. Thus, once block A is replaced, FRC_0 is freed, which allows this entry to be used later by *Req. D*. Recycling entries allows FRC to be smaller and thus more efficient than conventional approaches regarding energy consumption and area overhead.

The swap operation guarantees that the line storing the victim block is never in a transient state (note the lack of dashed lines in the plot below the L2 line of Figure 6.2b), and that the invalidation and eviction of the victim block are performed after the requested block is fetched. Consequently, FRC supports a higher cache level parallelism that allows responding to several requests at the same time. Furthermore, compared to the conventional approach, the lifetime of the victim block is enlarged when FRC is used.

Overall, as experimental results will show, the FRC has three main positive impacts on performance: i) reduces the memory access latency, ii) enlarges the lifetime of L2 cache blocks, and iii) exploits a higher MLP.

6.4.3 Potential FRC Performance Benefits

This section explores the potential performance benefits of the FRC approach and where they come from. To this end, the proposal is compared against two approaches, a fully-associative (FA) L2 cache and an FA L2 cache working together with a victim cache (FA+VC). The FA scheme is sized with the same number of entries as our experimental baseline (see Section 6.6 for further experimental details) and it is used to check the benefits coming from reducing the conflict misses. Notice that FA imposes an upper-bound for performance with respect to alternative set mapping strategies [51]. On the other hand, the FA+VC scheme is chosen to compare the potential benefits of a victim cache compared to our approach. In order to explore the potential performance, experiments assume that the additional structures (both VC and FRC) have an unbounded number of entries.

Performance is evaluated for the RX540 GPU in terms of Operations Per Cycle (OPC) and average number of execution cycles per wavefront in a kernel. The OPC is a performance metric analogous to the IPC, which is used to evaluate conventional processors [17]. An *operation* refers to the work performed by an individual thread when executing its corresponding part of a SIMD instruction. For instance, in our experimental platform, a SIMD unit can execute instructions from up to 64 threads, each one performing a scalar operation, which accounts for 64 operations. Regarding the execution cycles, they are split in two main categories referred to as compute cycles and memory cycles. The former indicates the mean time a wavefront is executing instructions and the latter the mean time a wavefront is blocked because it is waiting for a memory access.

For illustrative purposes, a pair of benchmarks showing two common and representative behaviors are presented (see Section 6.6). Figure 6.3 shows the results. As observed, for `DCT`, the FA cache improves performance (i.e., OPC in the left Y-axis) by 6% over the baseline thanks to reducing the number of conflict misses. On the other hand, no performance gains can be observed in `Color`, where long bursts of cache accesses many times exceed the cache capacity, and capacity misses dominate over conflict misses. In this application, the OPC is improved by 137% over the baseline when adding the VC, which helps to reduce capacity misses; however, the VC slightly helps in `DCT` since capacity misses are not as critical as in `Color`. The main reason is that `Color` is a memory-intensive kernel, where memory cycles dominate over compute cycles. On the contrary, in `DCT`, the compute cycles dominate the execution time, hence, little can be done by enlarging the cache capacity with a VC.

To sum up, it can be concluded that to enlarge the L2 cache size and/or to increase its associativity either directly or indirectly (i.e., with an additional memory structure) can improve the performance in some (especially memory-bounded) applications but it cannot in some others (compute-bounded). However, looking at the FRC with the same number of entries as the FA+VC approach but with a different data management, the system performance is signifi-

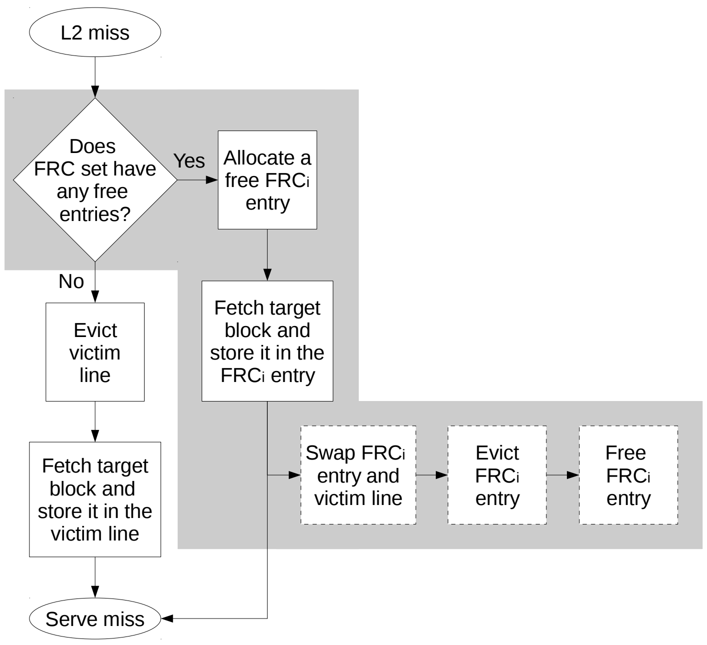

L2 miss
Does FRC set have any free entries?
Yes
Allocate a free FRCi entry
No
Evict victim line
Fetch target block and store it in the FRCi entry
Swap FRCi entry and victim line
Evict FRCi entry
Free FRCi entry
Fetch target block and store it in the victim line
Serve miss

Figure 6.4: Block diagram with the steps followed on an L2 miss. Those steps introduced with the FRC are highlighted in gray color.

cantly boosted in both kernels (by 29% and 163% for `DCT` and `Color`, respectively). The main reason is that the primary aim of FRC is not only to reduce the number of either conflict or capacity misses but to improve the MLP and to further reduce the memory access latency. Notice too that, for the FRC, the average time a wavefront is blocked for memory is smaller with respect to the other approaches so that, taking into account all the wavefronts of the kernel together, it turns into significant performance gains.

6.5 FRC Implementation

Figure 6.4 illustrates a block diagram with the steps involved on an L2 cache miss. The highlighted steps in gray color correspond to the proposed FRC approach. Upon an L2 miss (both in the L2 bank and the FRC), and if there are free entries in the target FRC set, the block is assigned to an FRC entry and the access is forwarded to the lower memory hierarchy level. This process is referred to as an early fetch. Once the early fetch is performed, the missing data can be already delivered to the processor. In this way, the victim block eviction is taken out of the critical path. To manage the eviction without leaving L2 cache lines in a transient state, the data stored in the FRC entry and the line storing the victim block (*victim line*) are swapped. Thereby, the eviction is handled from the FRC entry. Once the eviction finishes, the

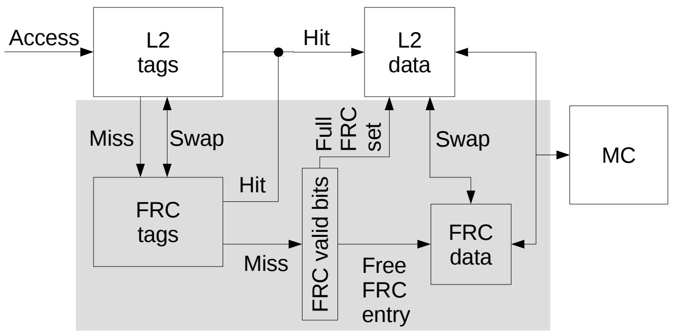


Figure 6.5: FRC hardware block diagram.

FRC entry is freed and enabled to handle subsequent L2 misses. In case that there is no free entry in the target FRC set, the FRC approach works like the conventional approach.

Figure 6.5 depicts a hardware block diagram of the FRC approach. The main focus of this paper is not to deal with the optimal implementation but on providing some insights on the design. A refined design for enhanced performance is beyond the scope of this paper. The FRC is plotted within the gray box, and, similarly to the L2 cache, includes the FRC tag and data arrays. For illustrative purposes, the valid bits are plotted in a different box. The access to the L2 tags and the FRC tags are performed sequentially and this is the way modeled in the experimental results. In practice, however, these structures can be indexed with the target block address in parallel or within the same cycle to avoid any latency penalty.

On an L2 cache access, the tags of the target set are looked up on a first stage. On an successful tag comparison, the requested block is retrieved from the L2 data array on a second stage and the FRC is not used. Otherwise, the FRC tags are looked up. On a miss in both the L2 and FRC tag array, a free block entry in the target FRC set is sought. Depending on whether the FRC set has a free entry or not, the fetched block from main memory is written into the FRC or the L2, respectively. In the former case, once the fetch completes, the L2 victim block is swapped with the FRC block. On the other hand, that is, on a hit in the FRC tag array, the request is served by the L2 data array. In this case, the request waits until the swap operation completes.

Finally, note that the FRC approach does not affect the state of the cache blocks, thus, it does not affect the coherence protocol.

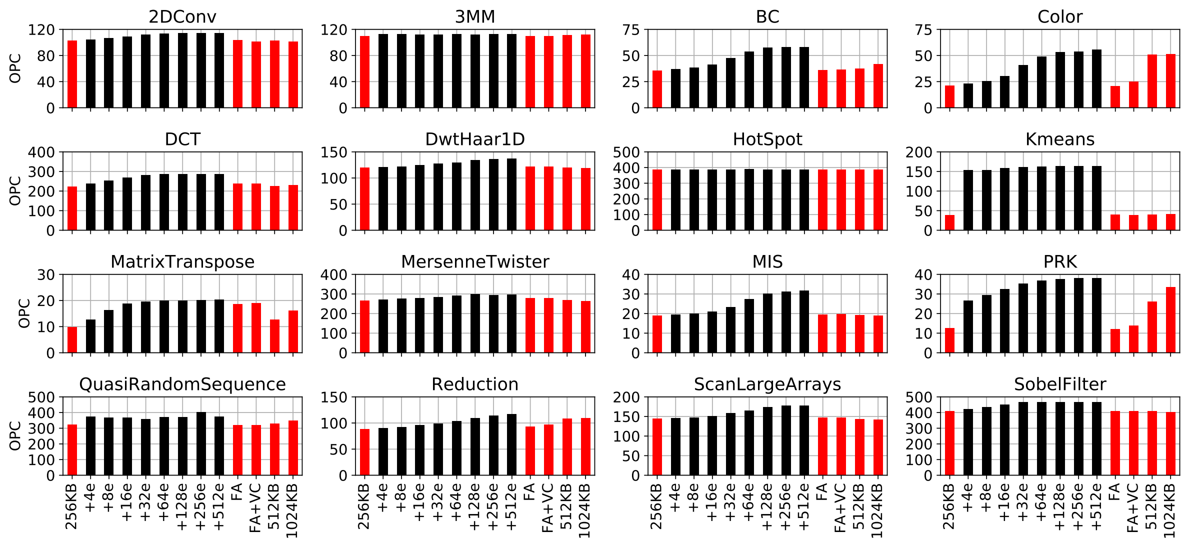


Figure 6.6: Operations Per Cycle (OPC) of the RX540 across the studied applications.

6.6 Experimental Evaluation

The FRC approach has been modeled and evaluated with the Multi2Sim [72] simulation framework. The simulation results include performance metrics and cache memory statistics required to compute the overall energy consumption.

We focus on the AMD Polaris GPU architecture. The RX540 GPU [68] has been modeled, including CUs, L1 and L2 caches, memory controllers, and GDDR5 memory modules [17]. The RX540 consists of 8 CUs, each one implementing the 4th version of the GCN core. The L2 cache is composed of two 32-way 256KB banks, which has been used as the baseline configuration.

The FRC consists of the L2 conventional cache plus two additional FRCs (one per bank). We analyze the performance sensitivity of our approach to the number of FRC entries, ranging from 4 (256B) to 512 (32KB) entries. All the evaluated configurations, except the smallest one, are organized as 8-way set-associative caches[1]. As mentioned above, the FRC is compared to the FA and FA+VC approaches. Experiments assume a fully-associative 32KB VC per bank, which matches the tested maximum FRC size. In addition, two conventional L2 caches consisting of two 32-way 512KB banks and two 32-way 1024KB banks are also evaluated. All the memory structures implement 64B lines.

[1]Higher associativity has been explored for enhanced performance. However, the marginal performance gains do not compensate the extra energy and area consumption. Therefore, all the presented results assume 8-way associativity.

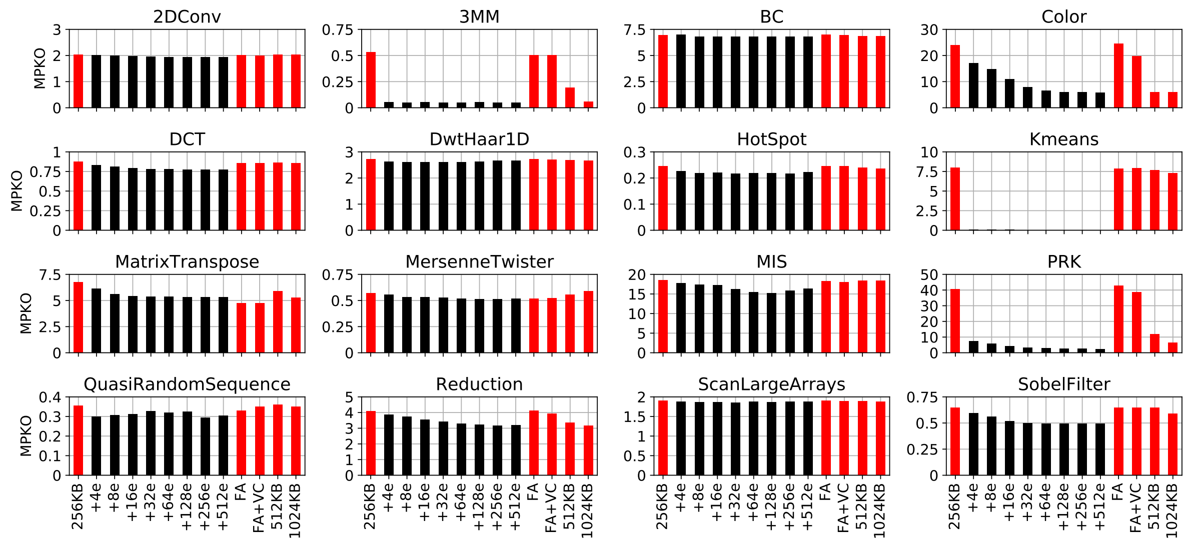


Figure 6.7: Misses Per Kilo-Operation (MPKO) in the L2 cache of the RX540.

To study the FRC performance scalability, we modeled the RX570 GPU with the same Polaris architecture as the RX540 GPU, but including up to 32 CUs and 8 L2 cache banks.

CACTI [69] has been used to estimate timing, energy, and area of the proposal. We have assumed a 32nm technology node and a 1.2GHz clock frequency. All the FRC caches are small enough to fit their access time within 1 clock cycle, whereas a swap operation is estimated to take 3 cycles to complete. Despite their fully-associative geometry, for comparison purposes, we conservatively assume the access time of the VCs to be the same as the FRCs. Regarding the L2 cache, it has been modeled with a 10-cycle access latency regardless of the cache geometry and capacity. The reader is referred to Section 6.6.4 for further experimental details about energy consumption.

Results have been obtained for 29 benchmarks from the OpenCL SDK [3], Rodinia [19], Pannotia [18], and PolyBench [57] benchmark suites. For illustrative purposes, a subset of 16 benchmarks are shown. All the benchmarks are run until completion.

6.6.1 System Performance Analysis

Figure 6.6 shows the OPC achieved by the RX540 GPU for the studied benchmarks. The red bar on the left side of each plot represents the 2×256KB L2 baseline cache, and the four red bars on the right side represent the 2×256KB FA L2 cache, the FA L2 cache with a 2×32KB VC (FA+VC), the 2×512KB L2 cache, and the 2×1024KB L2 cache, respectively. The black bars show results of the proposal varying the number of entries per FRC from 4 to 512, labeled as *+Ne*, where *N* indicates the number of entries.

The proposed approach achieves, across most of the studied applications (14 out of 16), OPC improvements higher than 10% over the baseline, reaching improvements up to 200% in applications such as `Kmeans` and `PRK`. It can be observed that OPC improves, in general, as the number of entries increases up to 64 or 128, where it saturates in most applications. In most of the benchmarks, the proposal performs better than blindly increasing the L2 cache associativity and capacity. Enlarging the cache capacity enhances the performance over a higher number of ways in benchmarks like `Color`, `PRK`, and `Reduction`.

According to the FRC performance, three behaviors can be appreciated:

- Smooth OPC increase. The OPC of applications exhibiting this behavior, which is the common case, gradually increases with additional FRC entries until a given saturation point, which is achieved with a small FRC of 64 or 128 entries. Examples are `DCT`, `MatrixTranspose`, and `ScanLargeArrays`.
- Sharp OPC increase. This behavior show a significant performance increase with just 4 FRC entries, but no remarkable OPC improvement is observed with additional entries. This is the case of `Kmeans`.
- Similar OPC. Applications in this category experience the same performance across all the studied cache approaches. This is the case of `3MM` and `HotSpot`, mainly due to their relatively low number of memory accesses, as shown in Section 6.6.2. Of course, the OPC of this type of applications is neither affected when either enlarging the cache associativity or capacity.

Overall, FRC boosts the OPC between 30% (*+4e*) and 67% (*+512e*) on average compared to the baseline. These values drop to 20% and 27% for the 512KB and 1024KB caches, respectively, and they are less than 10% for the FA schemes.

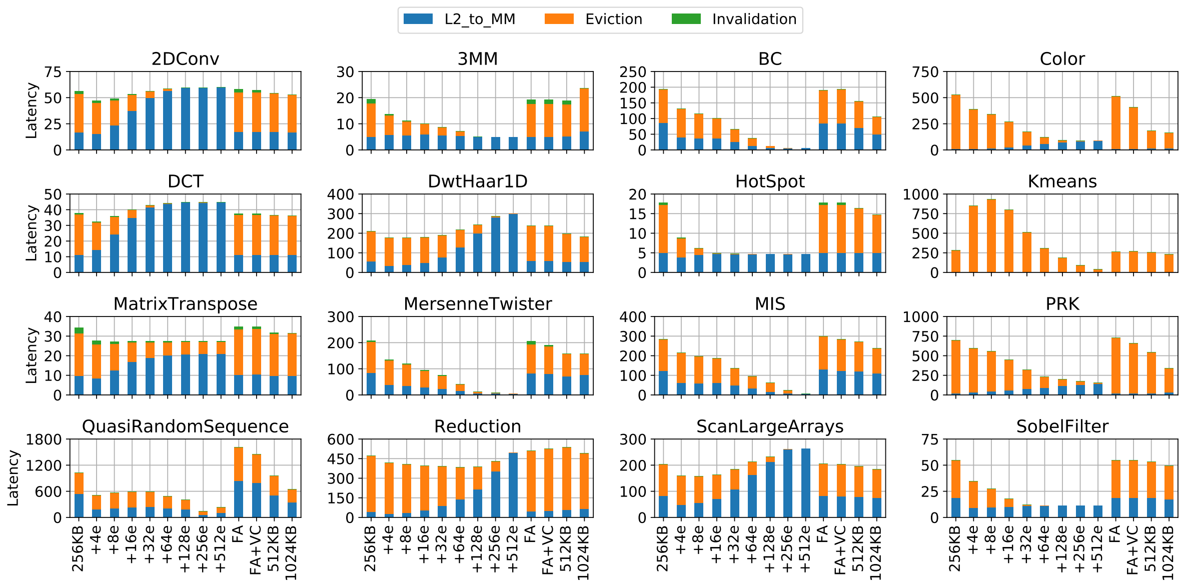


Figure 6.8: Average L2 miss latency (excluding main memory access time) in processor cycles for the RX540.

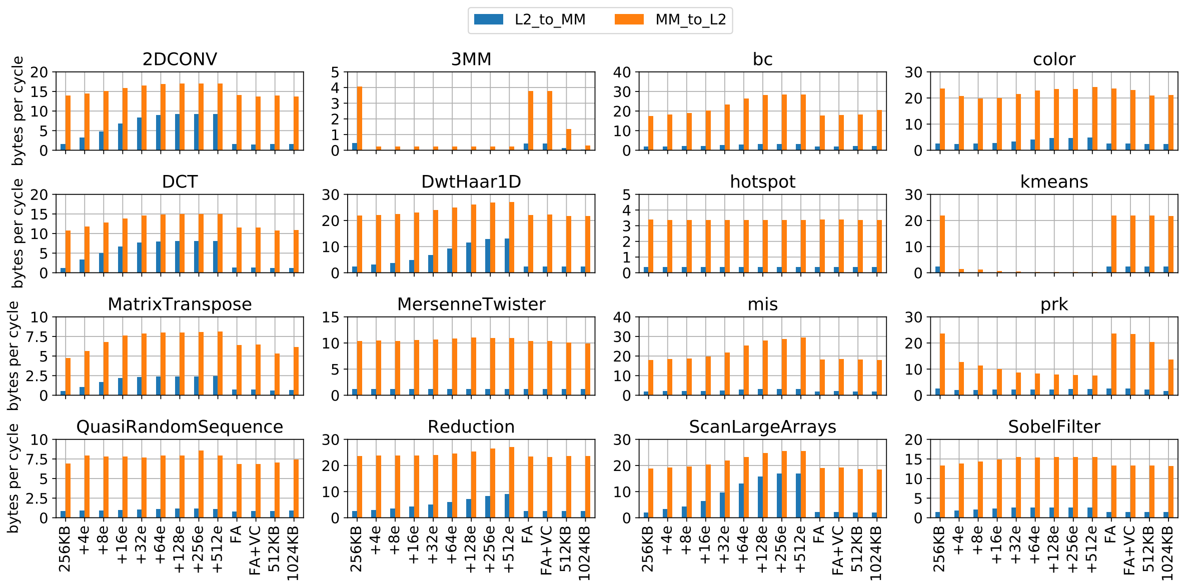


Figure 6.9: Traffic from the L2 to main memory and vice versa on the RX540.

6.6.2 Memory Subsystem Performance Analysis

To provide insights into the OPC trend shown by the RX540 GPU, this section evaluates the memory hierarchy performance.

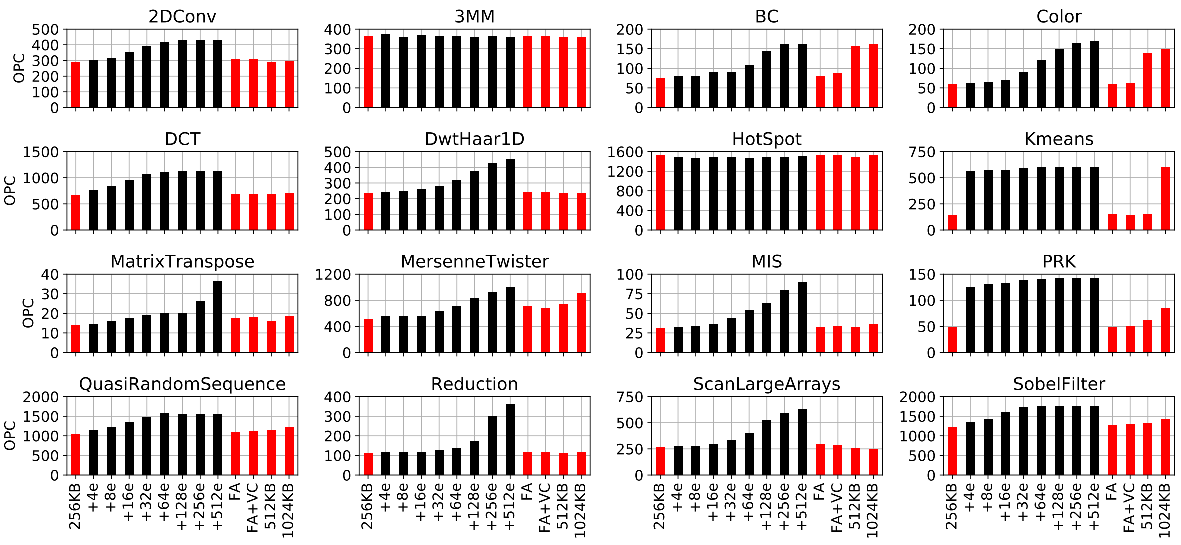


Figure 6.10: Operations Per Cycle (OPC) of the RX570 across the studied applications.

Misses Per Kilo-Operation

Misses per Kilo-Operations (MPKO) can be defined with analogous meaning to the MPKI (Misses Per Kilo-Instruction), widely used to study the cache hierarchy of the CPU counterparts. Figure 6.7 depicts the results. By adding FRC entries, the MPKO is reduced on average between 23% (*+4e*) and 31% (*+512e*) compared to the baseline approach. Moreover, the MPKO can be completely or mostly eliminated in applications like `Kmeans` and `PRK`. Note that in both benchmarks, FA+VC provides significantly lower MPKO reductions than FRC. This is because FRC does not only enlarge the lifetime of victim blocks (as a victim cache does) but also because it keeps them in a non-transient state. As a consequence, the number of hits improves over the conventional approaches.

Overall, an inverse correlation between OPC and MPKO can be appreciated. For kernels with a near-zero MPKO, (e.g., below 1) one might expect that increasing the number of hits would not improve the OPC. Examples are `3MM` and `HotSpot`. However, there are applications with an MPKO lower than 1 like `DCT`, `QuasiRandomSequence`, `MersenneTwister`, and `SobelFilter`, which improve their OPC with FRC. In order to explain this behavior, memory latency and bandwidth consumption are analyzed below.

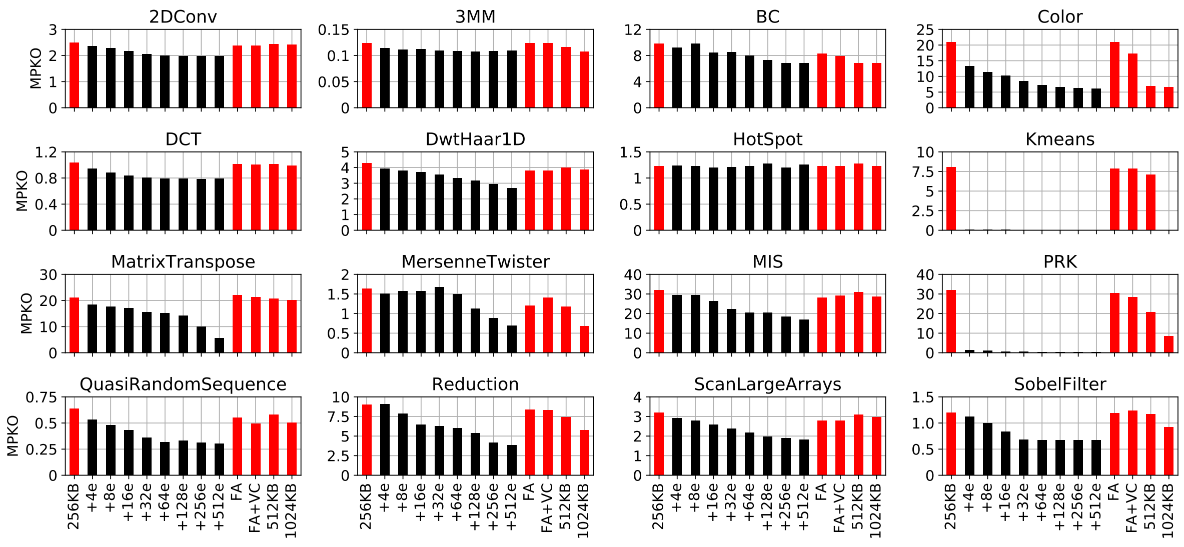


Figure 6.11: Misses Per Kilo-Operation (MPKO) in the L2 cache of the RX570.

Memory Latency and Bandwidth Consumption

L2 cache misses can be handled by either normal cache entries or FRC entries, however, FRC handles misses *faster* than the L2 cache since, part of the main memory latency is hidden by moving eviction and invalidation actions out of the critical path. In other words, the higher the number of misses handled by FRC the better the performance.

Figure 6.8 plots the average L2 miss latency results (excluding the actual DRAM access time without contention), quantified in clock cycles. The miss latency is split in three main components according to its causes (see Figure 6.2): invalidations, evictions, and fetches (L2_to_MM). The former category is due to invalidating and writing back (if necessary) the L1 copies of the blocks that are evicted from L2. The eviction category accounts for the latency due to evicting L2 blocks and writing back their data to main memory. Finally, the fetch latency refers to the time fetching target blocks but excluding the actual DRAM access time. That is, its value is only affected by main memory contention.

The use of FRC entries reduces the average L2 miss latency in some applications. Especially, the latency caused by evictions is removed in most applications with 512 FRC entries. The figure shows that the FRC approach also improves performance due to latency reduction. In particular, `BC`, `MersenneTwister`, and `QuasiRandomSequence`, which do not benefit from MPKO improvement, present a significant reduction in memory latencies ranging from 84% to 93% over the baseline. In contrast, in some applications like `2DConv`, `DCT`, `DwtHaar1D`,

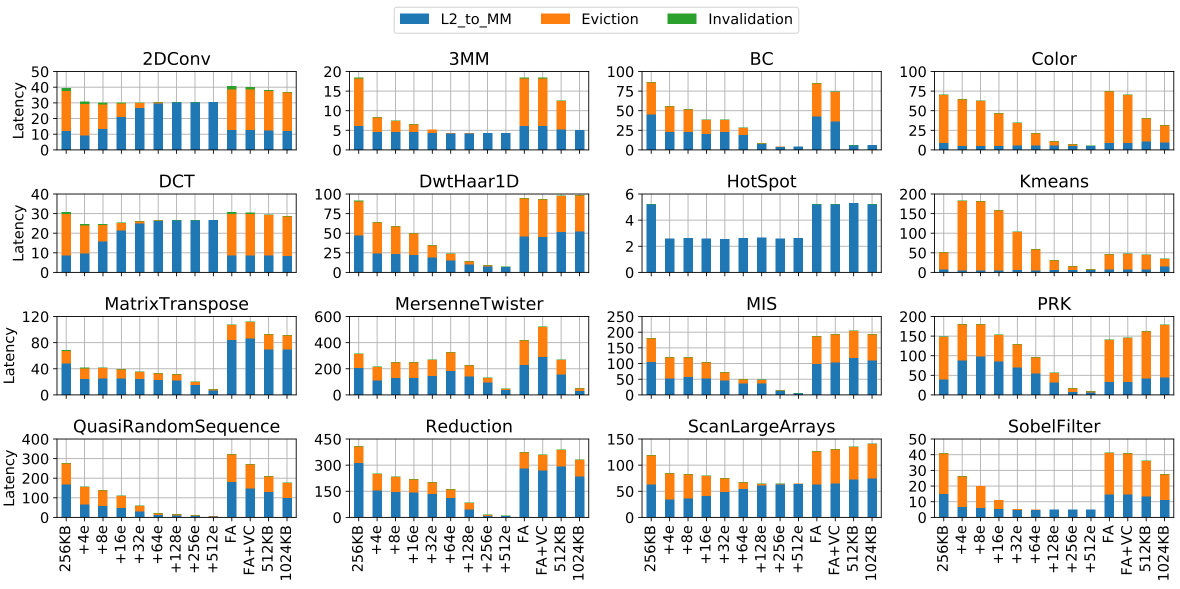


Figure 6.12: Average L2 miss latency (excluding main memory access time) in processor cycles for the RX570.

`Reduction`, or `ScanLargeArrays` the miss latency increases, in spite of removing the eviction latency.

To provide insights on this increase, Figure 6.9 shows the traffic in bytes per cycle from the L2 cache to main memory and vice versa. It can be seen that the traffic rises in these applications with the number of FRC entries. This is because adding more entries enables a higher MLP. In turn, such an MLP increases memory contention and L2 to memory latency, but also improves OPC since the memory latency growth can be partially hidden by the GPU massive parallelism, while the higher MLP enhances the system throughput.

6.6.3 Impact on Performance of Increasing the Number of Compute Units

The previous sections have focused on the performance of the RX540 GPU consisting of 8 CUs and 2 L2 cache banks. FRC, however, is expected to behave better compared to the other approaches with additional computational power and memory subsystem capabilities, since a higher L2 contention is expected. This section studies the FRC performance in the RX570 GPU, which implements 4× more compute units (32 CUs) and L2 cache banks (8 banks).

Figure 6.10 presents the OPC results. As expected, this GPU outperforms the smaller RX540 GPU. Results also show that, compared to the same baseline, OPC improvements of the proposal are higher than those achieved by the RX540 GPU. In most benchmarks, OPC improvements range from 40% to 300%, whereas these percentages fall down to 10% and 200%, respectively, for the RX540 GPU. To sum up, these results point out that the FRC approach performs even better with a larger L2 and potentially higher memory level parallelism.

Overall, FRC improves OPC between 32% (*+4e*) and 118% (*+512e*) on average with respect to the baseline cache. These percentages are by 22% and 50% for the 512KB and 1024KB caches, respectively. Note that all these configurations also improve the mean OPC achieved by RX540. However, as the next section will show, such a performance boost comes at the cost of a greater energy consumption and area overhead. For the FA schemes, the OPC improvement remains below 10%. To find out the reason why the RX570 GPU gets a higher improvement than the RX540, the MPKO and memory latencies for this GPU have been also studied. Figure 6.11 shows the MPKO results. Although the total cache capacity increases with the number of CUs, the MPKO rises in a high number (11 out of 16) of kernels with respect to the RX540 GPU because of the higher level of parallelism. Despite this fact, MPKO values of the FRC approach are similar to those of the RX540 GPU, with average MPKO reductions from 20% (*+4e*) to 48% (*+512e*) over the baseline.

Memory latencies are reduced in the RX570 GPU, as shown in Figure 6.12, because the memory traffic is distributed among more memory controllers. As a consequence, this GPU brings higher OPC improvements. Moreover, in the RX570, the FRC approach almost eliminates the eviction related latency and, in general, it is able to drop latency close to zero across most of the applications (remember that this latency does not include the main memory module access time). Therefore, memory contention is not an issue in the RX570, which enables further throughput improvements thanks to the MLP increase achieved by the proposal.

6.6.4 Energy Consumption

This section presents the methodology used to estimate both static and dynamic energy. Then, energy results are discussed for the larger RX570 GPU. We restrict the study to such a GPU because its size helps understand the impact of our proposal in high-performance computing.

Methodology

The three main FRC operations are accesses, fetches, and swaps. The FRC is accessed on every L2 tag miss, which triggers an FRC tag look-up. Upon a hit, the requested block is read from the L2 data array. A fetch operation causes a write operation of the fetched block from main memory to either the FRC or the L2 cache, depending on whether the target FRC set has free

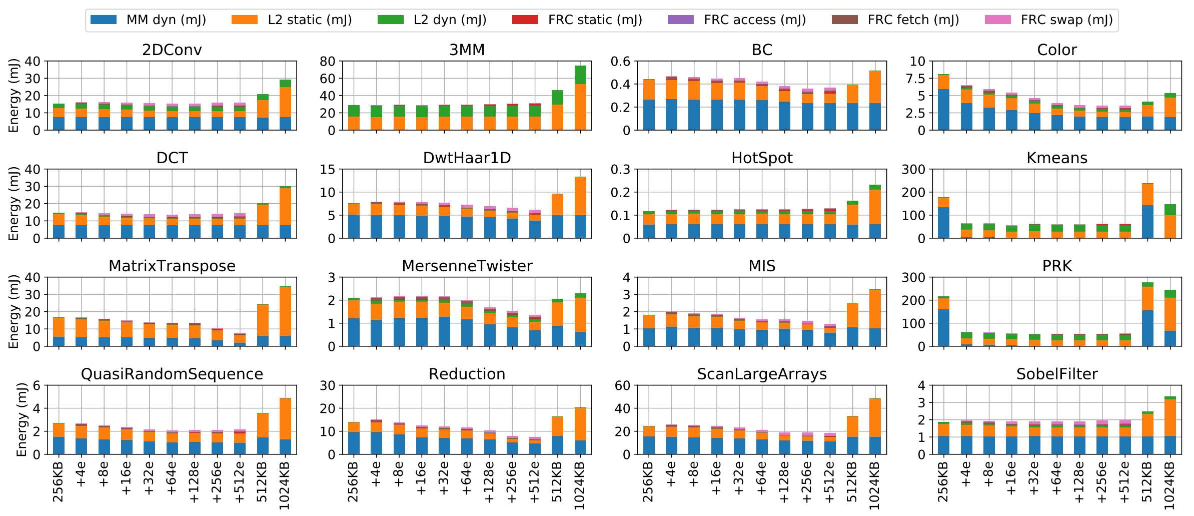


Figure 6.13: Energy consumption of the RX570 including the L2 cache banks and the main memory.

entries to allocate the incoming block or not, respectively. Note that such a write operation involves writing both the tag and data arrays. Finally, a swap operation is performed after an FRC fetch, and involves four steps: i) reading the victim block from L2, ii) reading the fetched block from the FRC, iii) writing the FRC block to L2, and iv) writing the L2 block to the FRC.

CACTI has been used to quantify the dynamic energy of each type of operation. Then, these numbers are multiplied by the number of times that each operation occurs during the entire kernel execution. Static energy overheads of L2 and FRC caches are quantified considering the execution time. Execution related events have been gathered from Multi2Sim simulations.

Results

Figure 6.13 plots the total energy consumption (in mJ) of the baseline, FRC, 2× sized, and 4× sized L2 caches. The L2 and FRC energy contributions are split into static and dynamic energy. The FRC dynamic energy is in turn divided into access, fetch, and swap expenses. In addition, the dynamic energy of a 2GB GDDR main memory (MM) module is also studied.

Compared to the energy consumption of the L2 and FRC caches, the consumption of the main memory represents a significant fraction of the overall consumption in most benchmarks and cache configurations. By reducing the number of accesses to this device, the FRC approach reduces such expenses over the conventional schemes. Some benchmarks showing this behavior are `Color`, `MersenneTwister`, and `Reduction`. Moreover, this energy contribution is mostly eliminated in `Kmeans` and `PRK`.

Focusing exclusively on the L2 and FRC caches, the static expenses dominate over dynamic expenses in most applications. This is mainly due to dynamic energy is consumed only upon a cache access, whereas static energy is consumed along time regardless of the cache is being accessed or not. In addition, the accesses to the tag and data arrays of both the L2 and FRC caches are serialized, meaning that the data array is only accessed in case of tag hit, which helps mitigate dynamic energy.

As expected, static energy increases with the L2 cache size. In comparison, the much smaller and less associative FRCs present low static energy consumption. In fact, FRC configurations present much lower static energy than the baseline in some applications like `DwtHaar1D` and `MatrixTranspose`. This is because the FRC approach highly improves the system performance in such kernels (see Section 6.6.3); thus, the number of execution cycles and static energy are significantly reduced over the baseline. In addition, FRC configurations with 256KB L2 caches consume much less static energy per cycle than conventional schemes with 512KB and 1024KB caches.

Compared to the baseline approach, those kernels with a heavy use of FRC entries like `2DCONV` and `MersenneTwister` increase the dynamic consumption with the number of FRC entries, especially due to swaps, which translate to up to 4 different cache operations as mentioned above. Nevertheless, notice that, despite FRCs consuming more dynamic energy than the baseline, the total consumption is very similar (e.g., `2DCONV`) or even reduced in some kernels (e.g., `MersenneTwister`) thanks to energy savings in both the dynamic main memory and static L2 energy. Furthermore, the total energy of FRC caches does not surpass that of 512KB and 1024KB caches.

Overall, the FRC approach obtains energy savings from 49% (*+4e*) to 57% (*+512e*) on average with respect to the baseline cache. Compared to the 512KB L2 cache, these percentages grow up to 62% and 67%, respectively.

6.6.5 Area Estimation

This section analyzes the area requirements of the proposed FRC approach. The area numbers include not only the tag and data arrays of the modeled caches, but also the cache controller peripherals (e.g., comparators, decoders, multiplexers, and sense amplifiers). Figure 6.14 shows the area (in mm^2) of the studied cache configurations. The presented numbers are for the

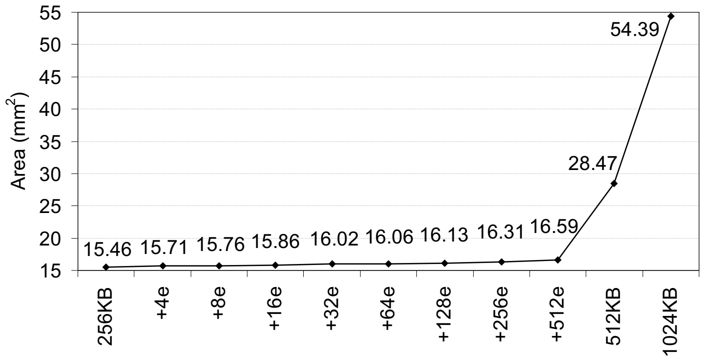


Figure 6.14: Area (in mm^2) of the cache configurations for the RX570 GPU.

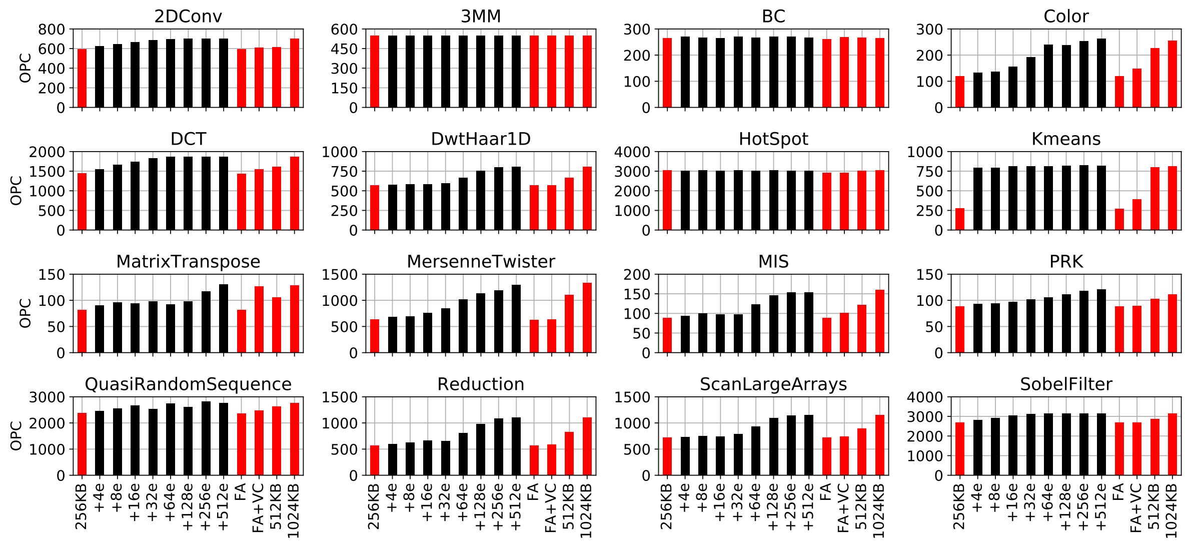


Figure 6.15: Operations Per Cycle (OPC) of the Vega64 across the studied applications.

RX570 GPU and refer to the 8 L2 cache banks plus the coupled FRC caches with each bank (if any).

The area overhead of the FRC schemes ranges from 1.6% (+4e) to 7.3% (+512e) compared to the baseline L2 cache without FRC. Nevertheless, these overheads are largely reduced compared to those of the much larger 512KB and 1024KB caches, whose cache capacities would translate into 4096 and 8192 FRC entries, respectively, per bank. The area overheads of these caches are up to 84.2% and 251.9%, respectively, over the baseline scheme.

6.6.6 Impact on Performance of Improved Memory Subsystem and Clock Frequency

In the Vega architecture [7], the most recent GPU generation from AMD to date, the GDDR5 main memory modules from the Polaris architecture are replaced with HBM modules in the GPU package. The HBM technology offers a higher memory bandwidth compared to GDDR5. In addition, the recent trend in new GPU generations is not only to improve the memory subsystem but also the GPU clock frequency.

This section evaluates the performance behavior of the FRC approach under a Vega64 GPU, which consists of 64 CUs, 16 L2 cache banks, and a clock frequency of 1.5GHz. This study not only evaluates scalability under additional computational power and memory subsystem capabilities (see Section 6.6.3) but also an improved memory subsystem and a higher clock frequency has been considered.

Figure 6.15 shows the OPC results for the Vega64 GPU. The Vega64 presents better OPC values with respect to the RX540 and RX570 across all the studied benchmarks thanks to the improved computational and memory capabilities and higher memory bandwidth. This fact does not prevent the FRC from boosting the OPC over the baseline cache in most applications. Although the average OPC improvements are not as high as those from the RX540 and RX570 GPUs, the FRC still boosts the OPC from 16% (*+4e*) to 54% (*+512e*) compared to the 256KB L2 cache. In this study, the 4× sized cache also reaches an average OPC improvement of 54% over the baseline. However, such a performance would be achieved with a greater energy consumption and area as discussed above.

6.7 Related Work

Prior work focusing on the GPU memory subsystem can be classified into works aimed at primarily improving either system performance or energy consumption, which are summarized in this section.

6.7.1 System Performance

The GPU memory subsystem performance has been widely analyzed in recent years from different perspectives including cache bypassing techniques [41, 44, 56], and optimization techniques of the memory subsystem design [40, 25, 47, 74, 32]. This section summarizes prior work in this regard.

Elastic-Cache [40] supports fine-grained L1 cache line management for kernels with irregular memory access patterns that do not efficiently exploit cache space. Auxiliary tags for fine-grained cache line management are stored in unused shared memory space, which is not fully occupied by many kernels. Gebhart et al. [25] propose to dynamically adjust the storage partitioning among registers, primary caches, and scratchpads depending on the kernel memory requirements, resulting in a reduction of the on-chip access latencies. IBOM [47] is an integrated architecture that leverages unused register file entries with lightweight ISA support to enlarge the L1 cache size. With enough cache capacity, a set balancing technique exploits underutilized sets to improve cache usage.

Other works have proposed additional memory structures to improve GPU performance. Taylor and Chang [46] investigate the effectiveness of adding a victim buffer to the L1 cache, and show that victim buffers with a relatively low number of lines obtain the same performance as doubling the L1 cache size. Wang et al. [74] incorporate a victim cache between L1 and L2 that presents the same capacity and associativity as the L1 cache. Reused blocks are kept in the L1 cache by enabling swap operations with the victim cache. Since a victim cache so large would impact on energy and area, unused entries from the register file and shared memory are proposed as an alternative to holding data that otherwise would remain in the victim cache. MRPB [32] is a memory-request priorization buffer that allows reordering and bypassing memory requests before they access the L1 cache. After being captured by the MRPB buffer, memory requests are released into the cache in a cache-friendly order to reduce cache thrashing and stalls.

Other research work has focused on memory and wavefront scheduling strategies [48, 64, 60].

6.7.2 Energy Consumption

Research addressing energy consumption in on-chip GPU caches has been done from different points of view including adaptive cache management techniques such as bypassing, thread throttling, indexing schemes, fine-grained fetching, and power-gating techniques [70, 20, 37, 59, 75], using alternative memory technologies to SRAM for on-chip storage [34, 62], and the proposal of additional on-chip memory structures [63].

Tian et al. [70] prevent streaming one-time-use blocks into the L1 cache with a dynamic bypass prediction technique. The proposed technique saves energy by avoiding useless cache insertions and evictions. Chen et al. [20] propose to protect the memory hierarchy from contention with a bypass policy based on reuse distance. Besides, this policy is combined with a thread throttling technique that dynamically controls the active number of threads in order to mitigate the contention and resource congestion. Reducing both the memory hierarchy contention and congestion translates into energy savings with respect to a conventional approach. IACM [37] is an integrated architecture combining Chen's bypassing and thread throttling techniques with an L1 cache indexing scheme. IACM dynamically determines the cache indexing bits that can mitigate cache thrashing and contention based on the runtime information of GPU kernels. LAMAR [59] is a technique that facilitates a fine-grained control of DRAM data fetches for those blocks with low spatial and temporal locality, reducing the energy-hungry traffic between on- and off-chip memory. This technique is combined with a bloom-filter predictor to adjust the fetching granularity at runtime. Wang et al. [75] mitigate the leakage energy consumption by putting both L1 and L2 caches in a state-retentive sleep mode when there are no ready threads to be scheduled and no memory requests, respectively. The effectiveness of the mechanism lies in the fact that the power on/off latencies are completely hidden.

Alternative high-density and low-leakage memory technologies have been used to implement energy-efficient GPU memory subsystems. Jing et al. [34] implement the GPU register file, shared memory, and L1 cache with eDRAM technology. The refresh penalty introduced by eDRAM is mitigated with the proposal of refresh mechanisms assisted by the compiler. Samavatian et al. [62] use STT-RAM technology to implement L2 caches. The main shortcomings of STT-RAM are the high energy and latency of write operations, which are addressed by reducing the data retention time thanks to the kernel data behavior.

Finally, additional memory structures have been also used for energy efficiency. In [63], the authors propose to allocate *TinyCaches* between each lane in a CU and the L1 cache to filter out memory requests to lower memory levels and save on-chip energy. By leveraging intrinsic characteristics of GPU programming models, these caches are kept non-coherent to avoid incurring additional overheads.

6.8 Conclusions

This paper has shown that the way Last-Level Cache (LLC) misses are handled in typical GPUs acts as a major performance limiter. To deal with this shortcoming, this work has presented a novel GPU cache subsystem design that leverages a tiny Fetch and Replacement Cache-like structure (FRC) between the LLC and the main memory. The design provides additional cache lines that allow prioritizing the fetch of incoming LLC cache blocks over the replacement of victim blocks. The proposed design boosts the system performance by increasing the MLP, improving the lifetime of the victimized blocks and removing eviction latencies from the critical path. Moreover, the small size of the FRC provides additional benefits regarding energy consumption and area compared to merely enlarging the LLC size.

The FRC attacks by design three main cache performance related issues, which results in a much better LLC cache management: i) it reduces the number of MPKO by keeping victim blocks in cache until fetch actions are completed, ii) it reduces the miss latency by starting the fetch actions from main memory as soon as a cache miss rises, and iii) it increases the MLP by unclogging new block requests whose target line is already being replaced.

Experimental results have shown that, compared to a conventional LLC design, the FRC increases the average OPC by 67%. In addition, the proposal also presents a high scalability, since it provides more performance benefits in a larger GPU, whose average OPC grows up to 118% over the baseline. Moreover, compared to a GPU using the recent HBM technology to implement the main memory modules, FRC improves the average OPC up to 54% over the conventional design. Such benefits come from a reduction of MPKO due to a higher availability of the contents of victim blocks as well as a reduction of miss latencies due to removing unnecessary serializations and eviction penalties from the critical path. We also found that in some kernels, latency increases because of the higher MLP, which causes additional contention accessing main memory. Nevertheless, this latency increase is not enough to constrain the performance improvements given by the MLP growth.

Results have also shown that the energy overhead of adding a small FRC with just tens of entries is largely compensated by its effectiveness, reaching energy savings up to 57% compared to the conventional design. These savings come with less than a 7.3% of LLC area increase.

Finally, evaluating the FRC approach considering also private L1 caches is planned as for future work.

Chapter 7

Results Discussion

This chapter summarizes the main results obtained during the work developed in this thesis. Since this thesis has focused on three main axes: i) characterization of GPGPU applications from the memory subsystem perspective, ii) GPU simulator framework improvements and validation, iii) the LLC miss management approach proposed in this dissertation, the chapter has been organized in three sections, each one aimed at discussing the main results of each axis.

7.1 Characterization of GPGPU Applications

The first objective of this thesis was to perform a characterization study of GPGPU applications from the memory subsystem perspective in order to find out possible sources of performance losses and performance bottlenecks. As a preliminary step, and due to high amount of traffic that typical memory coherence protocols introduce in the cache hierarchy, a memory protocol from a commercial GPU, namely SI (from the AMD Southern Islands GPU architecture), was modeled and compared against a GPU protocol from the academia (NMOESI) and the well-known MOESI protocol as baseline.

Once the SI protocol was modeled in Multi2Sim, the thesis work focused on characterizing the impact of the memory hierarchy of the GPU on the performance of the studied applications. The characterization study confirmed that the number of in-flight memory accesses in GPUs are several orders of magnitude larger than in CPUs. Because of this fact, it is of paramount importance that GPU memory protocols are capable of supporting a large amount of parallel memory accesses. For this reason, it is necessary to minimize the overhead of the protocols in the utilization of the interconnection network and also increase the parallelism of resources in the memory hierarchy. On the other hand, the results obtained in the characterization study allow us to draw three main concluding remarks that can be used to improve the management of memory accesses in current GPUs. These conclusions are:

- In contrast to CPU memory systems, under the NMOESI protocol, implementing a very high number of Miss Status Holding Registers (MSHR) in the cache can severely harm the performance. The main reason is that supporting more in-flight misses can result in an increase of the congestion of the memory system, which can turn into performance losses.
- Very high latencies (e.g. over two thousands cycles) cannot be hidden by the massive parallelism of current GPUs. In other words, there is a limit on the memory latency that, if surpassed, can cause serious drops in the system performance. Therefore, the protocol logic should prevent the latency from increasing above that limit.
- Specific coherence protocols are needed for GPUs, as also stated in other contemporary works. Moreover, our results show that both the NMOESI protocol and the SI protocol improve the system performance over the MOESI protocol by a factor of 4×.

7.2 Simulator Framework Improvements and Validation

The results of the characterization study also showed that modeling in detail certain elements and techniques of the memory hierarchy has a high impact on the results of the GPU simulations. The study revealed that these components are not so critical for performance when simulating CPUs; however, since the focus of this dissertation is on GPUs, a significant effort was made in the implementation of a realistic model of the GPU memory hierarchy in Multi2Sim. The improved code can be used both in GPU research as well as in research on heterogeneous systems composed of CPUs and GPUs.

The components modeled in higher detail are the following:

- MSHR file: the size of the MSHR file (e.g. its number of registers) limits the memory access parallelism in both CPUs and GPUs. However, it is on GPUs where this component has a much stronger impact since GPUs perform much more memory accesses per unit of time.

- Non-blocking writes: this technique prevents memory write instructions from blocking the instruction queue of memory accesses while write operations execute. Thus, this mechanism allows a higher memory access parallelism.

- Coalescing of memory accesses in the SIMD units: Multi2Sim coalesces memory requests in the memory subsystem. Nevertheless, GPUs perform a coalesce operation in the SIMD units before issuing the memory access in order to avoid congestion and unnecessary memory accesses.

- Memory controller and main memory: Multi2Sim does not simulate in detail the behavior of the memory controller and the GDDR memory banks. Instead, it adds a constant delay to the main memory accesses. To increase the detail of the simulation of these components, we incorporated the DRAMSim2 simulator in Multi2Sim.

- Coherence protocol: the coherence protocol used in GPUs from AMD (SI) is very different from the protocol originally implemented in Multi2Sim (NMOESI). Thus, we added support in Multi2Sim for the former protocol.

In addition, a validation study in which the new extended simulation framework was validated versus a commercial GPU (AMD Southern Islands 7870HD) was carried out. The validation study was conducted during the stay of the PhD student at the Northeastern University in Boston, USA, in collaboration with the research group led by Professor David R. Kaeli, which is in charge of the integration of GPU models in Multi2Sim.

The analysis of the experimental results revealed the following five main findings:

- Limiting the MSHR file size introduces important performance drops with respect to assuming an unbounded MSHR file.
- Coalescing in the SIMD units can bring important performance differences in some applications, since the number of L1 accesses can widely vary.
- Non-blocking stores improve the performance across all the studied benchmarks. This improvement is as much as 60% in some applications.
- The number of memory controllers and physical GDDR channels can reduce the performance in contrast to assuming a fixed main memory latency. In addition, the results widely vary depending on the memory controller configuration.
- The performance achieved under the SI protocol almost doubles the performance obtained with the NMOESI protocol in some applications.

The validation study showed that our simulator improvements significantly increase the accuracy of the original simulator with respect to the real system. This fact proves the relevance of accurately modeling realistically these hardware mechanisms because, otherwise, important performance deviations would arise so losing representativeness.

7.3 Proposed LLC Miss Management Approach

Once the modeling and characterization of the system was carried out, a proposal was devised to improve GPU performance. The proposed approach aims to improve the utilization (in terms of higher parallelism) of the resources of the memory hierarchy.

In the numerous studies performed during the dissertation, it was found that one of the factors that serializes the memory access and limits the memory level parallelism is the number of available directory entries. Typically, there is a directory entry for each cache line, so a straightforward solution to support higher parallelism is to increase the cache size, especially the last level cache, whose size has increased from hundreds of kilobytes to several megabytes in the last few years.

This increase, however, implies a significant growth of the area and energy consumption in the memory hierarchy of the GPU, which has a negative impact in the cost of the system that includes the GPU, regardless of whether it is a high-performance system or a mobile device. To deal with this drawback without reducing the parallelism of the hierarchy (or even improving it in some cases), the Fetch and Replacement Cache (FRC) approach was proposed. This approach is based on increasing the number of directory entries while keeping unaffected the number of cache lines.

The extra directory entries manage incoming memory requests immediately without evicting or locking the contents of any cache line. In this way, the FRC approach allows a given load request to progress before evicting the corresponding victim block. Once the missing block is fetched from memory and written into an FRC entry, the missing block and the victim block are swapped. This approach makes the *life cycle* of the victim block longer. The eviction of the victim block is finally performed from the FRC entry, which is subsequently freed until it receives the next access. Since the number of FRC entries is fixed and limited, in case all the entries are occupied, the incoming accesses can be handled as the cache typically does, but in this case without the advantages mentioned above.

Experimental results have shown that the FRC attacks three main issues related to cache performance, which results in much better last level cache management:

- The number of Misses per Kilo-Operation (MKPO) is reduced by keeping the victim block in the cache until the fetch of the missing block is completed.
- The miss latency is shortened by starting the fetch from main memory as soon as the cache miss rises.
- The memory level parallelism is improved by unclogging new block requests whose target line is already being replaced.

The proposed approach has been tested against the AMD GPU Southern Islands, Polaris and Vega architectures. The results have shown that the proposal increases, on average, performance by 67%. These benefits mainly come from a reduction of the MPKO due to a longer availability of the contents of the victim blocks, as well as a reduction of the miss latencies, due to removing unnecessary serializations and eviction penalties from the critical path. Finally, compared to a conventional design, the proposed FRC approach with an area increase smaller than 7.3% achieves energy savings up to 57%.

Note that the FRC is orthogonal to other cache techniques, like victim caches and write buffers. Victim caches could be used in conjunction with the FRC but, in general, they will not be effective to handle the potential memory-level parallelism of GPU applications mainly because of their small size. In other words, the miss management problems solved with the FRC proposal cannot be addressed with a conventional victim cache. A much larger victim cache could help but this naive solution does not scale and its implementation would suffer severe performance degradation and power issues. With respect to write buffers, they can hide writeback latencies, but the FRC provides additional benefits regarding hit ratio, fetch latencies, and memory-level parallelism.

Chapter 8

Conclusions

The main results obtained during the development of this PhD thesis have been published in several conferences and journals. This chapter summarizes the published works classifying them in three main categories according to the axes discussed in the previous chapter. Below, the conclusions and an enumeration of the scientific publications are given for each category. Finally, the last section of this chapter presents our plans for future work.

8.1 Characterization of GPGPU Applications

We characterized the behavior of the GPU applications according to the size of the Miss Status Holding Register (MSHR) file implemented in the L2 cache. We analyzed how the number of MSHRs impacts on typical memory performance metrics and on the overall system performance under two distinct GPU coherence protocols: NMOESI and SI (Southern Islands), which introduce different coherence traffic patterns. The study provided two key findings that can help improve the performance of GPU coherence protocols. First, there is a strong correlation between system performance and memory subsystem latency regardless of the protocol. Second, system performance varies with the number of supported cache misses (i.e. the supported Memory Level Parallelism or MLP). However, counterintuitively, supporting more cache misses does not always brings enhanced performance but it can turn into performance drops that vary depending on the coherence protocol.

The main results of the work carried out regarding the characterization of GPGPU applications gave rise to the following publications:

- Francisco Candel, Salvador Petit, Julio Sahuquillo, José Duato. **Impact of Memory Level Parallelism on the Performance of GPU Coherence Protocols**. In *Proceedings of the 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (PDP), pages 305-308, Heraklion, Greece, 2016.
  The content of this publication can be found in Chapter 2.

- Francisco Candel, Salvador Petit, Julio Sahuquillo, José Duato. **Impacto del Paralelismo de Memoria en los Protocolos de Coherencia para GPUs**. In *Actas de las XXVI Jornadas de Paralelismo* (JP), pages 233-242, Córdoba, Spain, 2015.

8.2 Simulator Framework Improvements and Validation

The memory hierarchy of the GPU is a critical research topic, since its design goals widely differ from those of conventional CPU memory hierarchies. This work is focused on accurately modeling the entire GPU memory subsystem. Researchers often require advanced microarchitectural simulators with detailed models of the memory subsystem to explore novel designs to better support GPGPU computing as well as to improve the performance of GPU and heterogeneous CPU-GPU systems. Nevertheless, due to the vertiginous trend at which current

GPU architectures evolve, simulation accuracy of existing state-of-the-art simulators suffers. In our first published research work on this topic, we identified three main architectural GPU features that should be modeled with more detail to improve the simulator accuracy: i) L1 cache MSHRs, ii) coalescing vector memory requests, and iii) non-blocking store instructions. Experimental results show that if the simulation framework does not implement these features, performance deviations can rise in some applications up to 70%, 75%, and 60%, respectively.

In our second work on this topic, published in the FGCS journal, we extended our first study and covered the modeling of both off-chip and on-chip memory subsystem components. In addition, we validated our models against a real commercial device, the AMD 7870HD GPU, which is based on the AMD Southern Islands GPU architecture. Regarding the on-chip memory hierarchy, we modeled and analyzed memory request coalescing mechanisms, cache coherence protocols, as well as both L1 and L2 MSHRs. With respect to the off-chip memory hierarchy, the work focused on the memory controller and the GDDR memory, which were modeled with the DRAMSim2 simulator. The experimental results showed that not modeling these components causes important deviations, which can vary the results provided by the simulation framework up to a factor of three.

The work regarding the simulator framework enhancements and validation gave out to the following international publications:

- Francisco Candel, Salvador Petit, Julio Sahuquillo, José Duato. **Accurately Modeling the GPU Memory Subsystem**. In *Proceedins of the 13th International Conference on High Performance Computing & Simulation* (HPCS), pages 179-186. Amsterdam, Netherlands, 2015.
  This publication is presented in Chapter 3.

- Francisco Candel, Salvador Petit, Julio Sahuquillo, José Duato. **Accurately Modeling the On-chip and Off-chip GPU Memory Susbsystem**. *Future Generation Computer Systems* (FGCS), volume 82, pages 510-519, 2018.
  The content of this publication can be found in Chapter 4.

In addition, other papers directly related to this topic have been published in the following international summer school and domestic conference:

- Francisco Candel, Salvador Petit, Julio Sahuquillo, José Duato. **Improving the Accuracy of GPU Memory Subsystem Models**. In *Proceedings of the 12th Interna-*

tional Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES), pages 151-154, Fiuggi, Italy, 2016.

- Francisco Candel, Salvador Petit, Julio Sahuquillo, José Duato. **Modelando de Forma Precisa el Subsistema de Memoria de una GPU**. In *Actas de las XXVII Jornadas de Paralelismo* (JP), pages 481-488, Salamanca, Spain, 2016.

8.3 Proposed LLC Miss Management Approach

In this dissertation we have devised a novel approach that leverages a tiny additional Fetch and Replacement Cache (FRC) to store control and coherence information of incoming cache blocks until they are fetched from main memory. Our first study on this topic shows that the FRC improves the system performance due to three main reasons: i) the lifetime of blocks being replaced is increased, ii) the main memory path is unclogged on long bursts of LLC misses, and iii) the average L2 miss latency is reduced. Experimental results show that our proposal improves the system performance (i.e. OPC) over 25% in most of the studied applications, reaching improvements up to 150% in some applications.

The first study was extended in five main ways: i) a more detailed hardware implementation of the FRC was presented and discussed, ii) additional applications from benchmark suites like Rodinia, Polibench, and Pannotia were also analyzed, iii) the newest AMD Arctic Islands (also known as Polaris) and Vega GPU architectures were tested, iv) performance scalability was explored by analyzing how the FRC proposal behaves with an increasing number of CUs and L2 cache sizes, and v) energy and area consumption costs were estimated and included. The extension shows that the proposed FRC cache scales in performance with the number of GPU compute units and the LLC size. Depending on the FRC size, performance improvements range from 30% to 67% for a modern baseline GPU card, and from 32% to 118% for a larger GPU. In addition, energy consumption is reduced on average from 49% to 57% for the larger GPU. These benefits come with a small area increase (by 7.3%) over the LLC baseline.

The main results of the work regarding our proposed LLC miss management approach gave out to the following international publications:

- Francisco Candel, Salvador Petit, Alejandro Valero, Julio Sahuquillo. **Improving GPU Cache Hierarchy Performance with a Fetch and Replacement Cache**. In *Proceedings of the 24th International European Conference on Parallel and Distributed Computing*

(Euro-Par), pages 235-248, Turin, Italy, 2018.
The contents of this publication are presented in Chapter 5.

- Francisco Candel, Salvador Petit, Alejandro Valero, Julio Sahuquillo. **An Energy-Efficient and Scalable Cache Approach to Boost GPGPU Throughput**. *IEEE Transactions on Computers* (TC), to appear in (DOI: 10.1109/TC.2019.2907591).
  This publication can be found in Chapter 6.

In addition, other related paper has been published in the following domestic conference:

- Francisco Candel, David Baselga, Alejandro Valero, Salvador Petit, Julio Sahuquillo. **Mejora de las Prestaciones de las GPU con una Cache para Búsquedas y Reemplazos**. In *Actas de las XXIX Jornadas de Paralelismo* (JP), pages 161-169, Teruel, Spain, 2018.

8.4 Other Indirectly Related Work

In addition to the publications mentioned above performed by the author of this dissertation, both included in previous chapters as well as the directly related and published in domestic conferences (not included), the PhD candidate has co-authored two international papers dealing with transistor aging in the GPU register file. Below the complete references of these publications are presented:

- Francisco Candel, Alejandro Valero, Salvador Petit, Darío Suárez Gracia, Julio Sahuquillo. **Exploiting Data Compression to Mitigate Aging in GPU Register Files**. *Proceedings of the 29th International Symposium on Computer Architecture and High Performance Computing* (SBAC-PAD 2017), Campinas, Brazil, October 17-20, 2017.

- Alejandro Valero, Francisco Candel, Darío Suárez Gracia, Salvador Petit, Julio Sahuquillo. **An Aging-Aware GPU Register File Design Based on Data Redundancy.** *IEEE Transactions on Computers*, Vol. 68, Issue 1, pp. 4-20, 2019.

8.5 Future Work

As for future work, we plan to extend the FRC to the L1 data cache. Since the FRC approach has demonstrated that it can increase effectively the memory parallelism of the L2 cache, it also can potentially help improve the performance of the L1 cache too, allowing increasing the parallelism of the memory subsystem even more.

Nevertheless, L1 improvements could not directly translate into an increase of the overall system performance because the benefits of these improvements may be hidden by other components of the memory subsystem, like the L2 cache. Moreover, there are important differences between both cache levels that should be considered. On the one hand, the L1 cache capacity is much smaller than that of the L2 cache; thus, the size of the FRC should be proportionally lowered. On the other hand, the traffic patterns of the L1 cache accesses are different from L2, since L1 data caches are private for each CU. Therefore, the devised FRC organization and miss management mechanism for L1 caches would probably differ from those presented in this dissertation.

Bibliography

[1] *Heterogeneous System Architecture foundation* (cit. on pp. 10, 29, 47).

[2] AMD. *Evergreen Family Instruction Set Architecture* (cit. on pp. 7, 35, 54).

[3] AMD. *AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK)* (cit. on pp. 10, 17, 29, 37, 46, 57, 74, 97).

[4] AMD. *AMD Accelerated Parallel Processing OpenCL Programming Guide.* Dec. 2013 (cit. on pp. 56, 63).

[5] AMD. *Dissecting Polaris Architecture* (cit. on pp. 4, 88).

[6] AMD. *Graphics Core Next Architecture, Generation 3* (cit. on p. 4).

[7] AMD. *Radeon's next-generation Vega architecture.* 2017 (cit. on pp. 4, 88, 109).

[8] AMD. *Southern Islands Series Instruction Set Architecture* (cit. on pp. 15, 16, 56, 70).

[9] AMD. *"Vega" Instruction Set Architecture* (cit. on p. 4).

[10] D. August et al. "UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development". In: *Computer Architecture Letters* 6.2 (2007), pp. 45–48. ISSN: 1556-6056. DOI: `10.1109/L-CA.2007.12` (cit. on pp. 29, 47).

[11] A. Bakhoda et al. “Analyzing CUDA workloads using a detailed GPU simulator”. In: *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 2009, pp. 163–174. DOI: `10.1109/ISPASS.2009.4919648` (cit. on pp. 10, 28, 46).

[12] Nathan Binkert et al. “The Gem5 Simulator”. In: *SIGARCH Comput. Archit. News* 39.2 (2011), pp. 1–7. ISSN: 0163-5964. DOI: `10.1145/2024716.2024718` (cit. on pp. 29, 46).

[13] D. Boggs et al. “The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology.” In: *Intel Technology Journal* 8.1 (2004) (cit. on p. 38).

[14] A. Branover, D. Foley, and M. Steinman. “AMD Fusion APU: Llano”. In: *IEEE Micro* 32.2 (2012), pp. 28–37. ISSN: 0272-1732. DOI: `10.1109/MM.2012.2` (cit. on p. 45).

[15] F. Candel et al. “Accurately modeling the GPU memory subsystem”. In: *2015 International Conference on High Performance Computing Simulation (HPCS)*. 2015, pp. 179–186. DOI: `10.1109/HPCSim.2015.7237038` (cit. on pp. 17, 51).

[16] F. Candel et al. “Improving the GPU Cache Hierarchy Performance with a Fetch and Replacement Cache”. In: *Proceedings of the 24th International European Conference on Parallel and Distributed Computing*. 2018, pp. 235–248 (cit. on p. 88).

[17] Francisco Candel et al. “Accurately Modeling the On-chip and Off-chip GPU Memory Subsystem”. In: *Elsevier Future Generation Computer Systems* 82 (2018), pp. 510–519. ISSN: 0167-739X. DOI: `https://doi.org/10.1016/j.future.2017.02.012` (cit. on pp. 74, 75, 93, 96).

[18] S. Che et al. “Pannotia: Understanding Irregular GPGPU Graph Applications”. In: *Proceedings of the IEEE International Symposium on Workload Characterization*. 2013, pp. 185–195 (cit. on p. 97).

[19] S. Che et al. “Rodinia: A Benchmark Suite for Heterogeneous Computing”. In: *Proceedings of the IEEE International Symposium on Workload Characterization*. 2009, pp. 44–54 (cit. on p. 97).

[20] X. Chen et al. “Adaptive Cache Management for Energy-Efficient GPU Computing”. In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 2014, pp. 343–355. DOI: `10.1109/MICRO.2014.11` (cit. on pp. 87, 111).

[21] Hanjin Chu. *AMD heterogeneous Uniform Memory Access.* `http://events.csdn.net/AMD/GPUSat%20-%20hUMA_june-public.pdf` (cit. on p. 14).

[22] S. Collange et al. "Barra: A Parallel Functional Simulator for GPGPU". In: *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on.* 2010, pp. 351–360. DOI: `10.1109/MASCOTS.2010.43` (cit. on pp. 10, 29, 47).

[23] W.W.L. Fung et al. "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow". In: *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on.* 2007, pp. 407–420. DOI: `10.1109/MICRO.2007.30` (cit. on pp. 10, 28, 46).

[24] Benedict Gaster et al. *Heterogeneous Computing with OpenCL.* 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 0123877660, 9780123877666 (cit. on p. 36).

[25] M. Gebhart et al. "Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture.* 2012, pp. 96–106. DOI: `10.1109/MICRO.2012.18` (cit. on pp. 77, 78, 110).

[26] Kourosh Gharachorloo et al. "Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors". In: *SIGARCH Comput. Archit. News* 18.2SI (May 1990), pp. 15–26. ISSN: 0163-5964. DOI: `10.1145/325096.325102` (cit. on pp. 8, 55).

[27] A. Glenis and S. Petridis. "Performance and Energy Characterization of High-performance Low-cost Cornerness Detection on GPUs and Multicores". In: *Proceedings of the 5th International Conference on Information, Intelligence, Systems and Applications.* 2014, pp. 181–186. DOI: `10.1109/IISA.2014.6878727` (cit. on pp. 1, 68, 86).

[28] Q. Huang et al. "GPU as a General Purpose Computing Resource". In: *2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies.* 2008, pp. 151–158. DOI: `10.1109/PDCAT.2008.38` (cit. on p. 45).

[29] S. Huang, S. Xiao, and W. Feng. "On the energy efficiency of graphics processing units for scientific computing". In: *2009 IEEE International Symposium on Parallel Distributed Processing.* 2009, pp. 1–8. DOI: `10.1109/IPDPS.2009.5160980` (cit. on pp. 1, 44, 68, 86).

[30] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 0123797519, 9780123797513 (cit. on pp. 9, 53).

[31] JEDEC. *High Bandwidth Memory 2 (HBM2) DRAM Standard.* Dec. 2018. URL: `https://www.jedec.org/news/pressreleases/jedec-updates-groundbreaking-high-bandwidth-memory-hbm-standard-0` (cit. on p. 6).

[32] W. Jia, K. A. Shaw, and M. Martonosi. "MRPB: Memory request prioritization for massively parallel processors". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA).* 2014, pp. 272–283. DOI: `10.1109/HPCA.2014.6835938` (cit. on pp. 22, 51, 58, 63, 77, 110, 111).

[33] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. "Characterizing and Improving the Use of Demand-fetched Caches in GPUs". In: *Proceedings of the 26th ACM International Conference on Supercomputing.* ICS '12. San Servolo Island, Venice, Italy, 2012, pp. 15–24. ISBN: 978-1-4503-1316-2 (cit. on p. 22).

[34] N. Jing et al. "Energy-Efficient eDRAM-Based On-Chip Storage Architecture for GPGPUs". In: *IEEE Transactions on Computers* 65.1 (2016), pp. 122–135 (cit. on pp. 87, 111, 112).

[35] Khronos Group. *The OpenCL Specification.* Nov. 2015 (cit. on p. 48).

[36] Khronos Group, Khronos OpenCL Working Group, et al. *OpenCL-The open standard for parallel programming of heterogeneous systems* (cit. on pp. 4, 48).

[37] K. Y. Kim, J. Park, and W. Baek. "IACM: Integrated Adaptive Cache Management for High-Performance and Energy-Efficient GPGPU Computing". In: *Proceedings of the IEEE 34th International Conference on Computer Design.* 2016, pp. 380–383 (cit. on p. 111).

[38] C. J. Lee et al. "Prefetch-Aware DRAM Controllers". In: *2008 41st IEEE/ACM International Symposium on Microarchitecture.* 2008, pp. 200–209. DOI: `10.1109/MICRO.2008.4771791` (cit. on pp. 9, 52).

[39] Jingwen Leng et al. "GPUWattch: Enabling Energy Optimizations in GPGPUs". In: *SIGARCH Comput. Archit. News* 41.3 (June 2013), pp. 487–498. ISSN: 0163-5964. DOI: `10.1145/2508148.2485964` (cit. on pp. 29, 46).

[40] B. Li et al. “Elastic-Cache: GPU Cache Architecture for Efficient Fine- and Coarse-Grained Cache-Line Management”. In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. 2017, pp. 82–91 (cit. on pp. 77, 110).

[41] C. Li et al. “Locality-driven Dynamic GPU Cache Bypassing”. In: *Proceedings of the 29th International ACM Conference on Supercomputing*. 2015, pp. 67–77 (cit. on pp. 77, 110).

[42] Sheng Li et al. “McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures”. In: *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. 2009, pp. 469–480 (cit. on pp. 29, 46).

[43] Y. Liang et al. “An Efficient Compiler Framework for Cache Bypassing on GPUs”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.10 (2015), pp. 1677–1690. ISSN: 0278-0070. DOI: `10.1109/TCAD.2015.2424962` (cit. on p. 87).

[44] Y. Liang et al. “Optimizing Cache Bypassing and Warp Scheduling for GPUs”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.8 (2018), pp. 1560–1573 (cit. on pp. 77, 110).

[45] X. Mei and X. Chu. “Dissecting GPU Memory Hierarchy Through Microbenchmarking”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.1 (2017), pp. 72–86 (cit. on p. 88).

[46] E. M.Taylor and D. W.Chang. “Studying Victim Caches in GPUs”. In: *Proceedings of the 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. 2018, pp. 394–398 (cit. on p. 111).

[47] S. Mu et al. “IBOM: An Integrated and Balanced On-Chip Memory for High Performance GPGPUs”. In: *IEEE Transactions on Parallel and Distributed Systems* 29.3 (2018), pp. 586–599 (cit. on pp. 77, 78, 110).

[48] S. Mu et al. “Orchestrating Cache Management and Memory Scheduling for GPGPU Applications”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.8 (2014), pp. 1803–1814 (cit. on pp. 22, 77, 111).

[49] Aaftab Munshi et al. *OpenCL Programming Guide*. 2.7. Addison-Wesley Professional, 2013 (cit. on pp. 7, 31, 35, 54).

[50] Paula Navarro et al. “Row Tables: Design Choices to Exploit Bank Locality in Multiprogram Workloads”. In: *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015.* 2015, pp. 22–26 (cit. on p. 60).

[51] C. Nugteren et al. “A detailed GPU cache model based on reuse distance theory”. In: *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on.* 2014, pp. 37–48. DOI: `10.1109/HPCA.2014.6835955` (cit. on pp. 38, 58, 63, 93).

[52] C Nvidia. “NVIDIA’s Next Generation CUDA Compute Architecture: FERMI”. In: *Comput. Syst* 26 (2009), pp. 63–72 (cit. on pp. 4, 48).

[53] NVIDIA Corporation. *NVIDIA Maxwell Architecture.* Jan. 2014. URL: `https://developer.nvidia.com/maxwell-compute-architecture/` (cit. on pp. 2, 87).

[54] NVIDIA Corporation. *NVIDIA Pascal Architecture.* Jan. 2016. URL: `https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/` (cit. on pp. 2, 87).

[55] NVIDIA Corporation. *NVIDIA Volta Architecture.* Jan. 2018. URL: `https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/` (cit. on pp. 2, 87).

[56] Yunho Oh et al. “APRES: Improving Cache Efficiency by Exploiting Load Characteristics on GPUs”. In: *Proceedings of the 43rd International Symposium on Computer Architecture.* 2016, pp. 191–203 (cit. on p. 110).

[57] L.-N. Pouchet. *Polybench: The Polyhedral Benchmark Suite.* `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/` (cit. on p. 97).

[58] Jason Power et al. “Heterogeneous System Coherence for Integrated CPU-GPU Systems”. In: *MICRO, 2013.* MICRO-46. Davis, California, 2013, pp. 457–467. ISBN: 978-1-4503-2638-4. DOI: `10.1145/2540708.2540747` (cit. on p. 22).

[59] M. Rhu et al. “A Locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures”. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture.* 2013, pp. 86–98 (cit. on pp. 111, 112).

[60] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. "Cache-Conscious Wavefront Scheduling". In: *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 2012, pp. 72–83 (cit. on p. 111).

[61] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. "DRAMSim2: A Cycle Accurate Memory System Simulator". In: *IEEE Comput. Archit. Lett.* 10.1 (Jan. 2011), pp. 16–19. ISSN: 1556-6056. DOI: `10.1109/L-CA.2011.4` (cit. on p. 53).

[62] Mohammad Hossein Samavatian et al. "Architecting the Last-Level Cache for GPUs Using STT-RAM Technology". In: *ACM Transactions on Design Automation of Electronic Systems* 20.4 (2015), 55:1–55:24 (cit. on pp. 111, 112).

[63] A. Sankaranarayanan et al. "An Energy Efficient GPGPU Memory Hierarchy with Tiny Incoherent Caches". In: *Proceedings of the International Symposium on Low Power Electronics and Design*. 2013, pp. 9–14 (cit. on pp. 77, 78, 111, 112).

[64] A. Sethia, D. A. Jamshidi, and S. Mahlke. "Mascar: Speeding up GPU Warps by Reducing Memory Pitstops". In: *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture*. 2015, pp. 174–185 (cit. on pp. 77, 111).

[65] I. Singh et al. "Cache coherence for GPU architectures". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 2013, pp. 578–590. DOI: `10.1109/HPCA.2013.6522351` (cit. on p. 22).

[66] John E Stone, David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems". In: *Computing in science & engineering* 12.1-3 (2010), pp. 66–73 (cit. on pp. 4, 48).

[67] P. Sweazey and A. J. Smith. "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus". In: *Proceedings of the 13th Annual International Symposium on Computer Architecture*. ISCA '86. Tokyo, Japan: IEEE Computer Society Press, 1986, pp. 414–423. ISBN: 0-8186-0719-X (cit. on pp. 8, 55).

[68] AMD Radeon Graphics Technology. *AMD Graphics Cores Next (GCN) Architecture White Paper*. June 2012 (cit. on pp. 4, 15, 29, 47, 56, 70, 74, 89, 96).

[69] S. Thoziyoor et al. "A Comprehensive Memory Modeling Tool and its Application to the Design and Analysis of Future Memory Hierarchies". In: *Proceedings of the 35th Annual*

International Symposium on Computer Architecture. Beijing, China, 2008, pp. 51–62. DOI: 10.1109/ISCA.2008.16 (cit. on pp. 88, 97).

[70] Yingying Tian et al. "Adaptive GPU Cache Bypassing". In: *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*. 2015, pp. 25–35 (cit. on p. 111).

[71] Top500.org. Top500 Supercomputer Sites, http://top500.org (cit. on pp. 1, 26, 45, 68, 86).

[72] Rafael Ubal et al. "Multi2Sim: A Simulation Framework for CPU-GPU Computing". In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. PACT '12. Minneapolis, Minnesota, USA: ACM, 2012, pp. 335–344. ISBN: 978-1-4503-1182-3. DOI: 10.1145/2370816.2370865 (cit. on pp. 10, 17, 29, 46, 47, 69, 74, 88, 96).

[73] Rafael Ubal et al. "Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors". In: *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*. 2007, pp. 62 –68. DOI: 10.1109/SBAC-PAD.2007.17 (cit. on pp. 10, 29, 47).

[74] Jianfei Wang et al. "Incorporating Selective Victim Cache into GPGPU for High-performance Computing". In: *Wiley Concurrency and Computation: Practice and Experience* 29.24 (2017), pp. 1–11 (cit. on pp. 77, 78, 110, 111).

[75] Y. Wang, S. Roy, and N. Ranganathan. "Run-Time Power-Gating in Caches of GPUs for Leakage Energy Savings". In: *Proceedings of the Design, Automation Test in Europe Conference Exhibition*. 2012, pp. 300–303 (cit. on pp. 111, 112).