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

Francisco Candel, Salvador Petit, Julio Sahuquillo, and José Duato

Department of Computer Engineering
Universitat Politècnica de València
46012 Valencia, Spain
Email: fracanma@inf.upv.es

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.

Keywords: applied modeling and simulation, on-chip memory subsystem, main memory controller, GDDR, cache coherence protocol

---

1. 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 [1] 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 [2] 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 [3] 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 [4] 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 [5], 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× or 3×) performance deviations in the simulated results.

The remainder of this work is organized as follows. Section 2 presents a relevant subset of current GPU simulators. Section 3 describes the Southern Islands architecture and its programming model. In Section 4, the proposed Multi2Sim extensions are described in detail. Section 5 presents the experimental results. Section 6 provides the accuracy improvements achieved by the proposed extensions. Finally, in Section 7 some concluding remarks are drawn.

2. 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 [6, 7] 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 [8] 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 [9], an energy model based on McPAT [10]; 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 [11] with modern OpenCL code.

Barra [12] is a parallel GPU functional simulator. It is based in the UNISIM framework [13] 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 [14, 5] 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 [15], 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

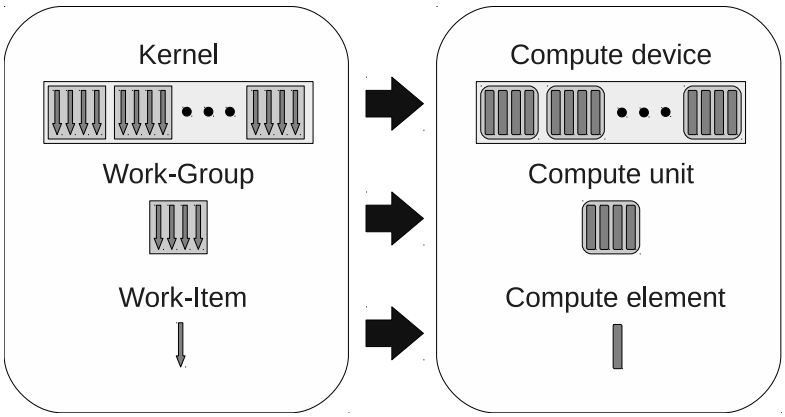

Figure 1: OpenCL mapping between execution and platform models.

CPU-GPU cycle by cycle, ii) it implements the recent AMD GPU core architectures called GCN [16], iii) it includes its own OpenCL and CUDA libraries, and iv) support for the HSA architecture is being developed.

3. 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.

3.1. The OpenCL Programming Model

Two main programming frameworks, CUDA [17] from Nvidia and OpenCL [18] 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 [19]. 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 [20] defines a platform model and an execution model. The platform model is an abstraction of the real machine in which the

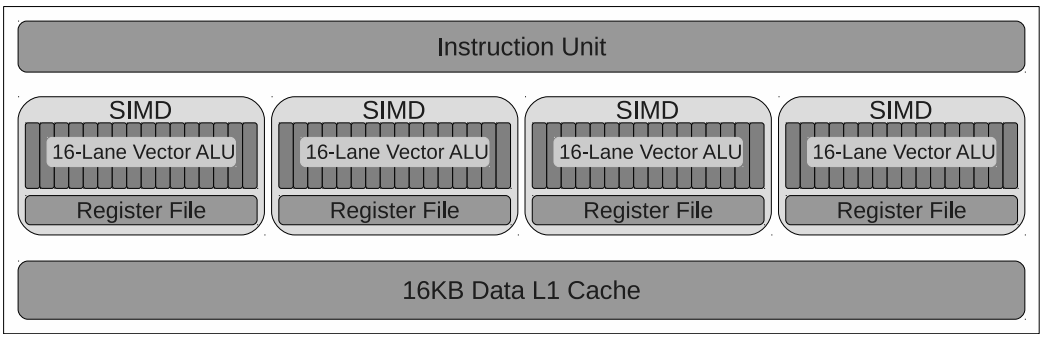

Figure 2: GCN compute unit.

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 1 depicts a block diagram of both models and their mapping.

3.2. 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 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.

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.

3.3. 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

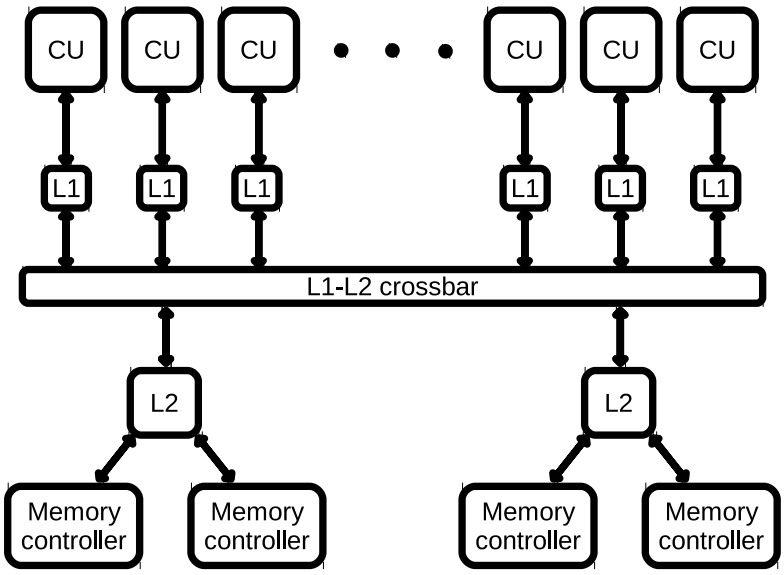

Figure 3: Southern Islands memory hierarchy.

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 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-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.

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 1: Summary of the proposed Multi2Sim extensions.

4. 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 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 5, 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.

4.1. 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 [21].

Some recent works [22] 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*.

4.2. 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 [23]. 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 [24].

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 [25]. 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.

4.3. 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 [26, 27] 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 5, 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.

4.4. 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 [28] 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 [29]. 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 [16]. 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 [30, 16, 31] 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.

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

Table 2 summarizes the main machine parameters. The OpenCL SDK 2.5 benchmarks adapted for Multi2Sim [32] has been used in the evaluation

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 2: Cache-hierarchy and GPU configuration.

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 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

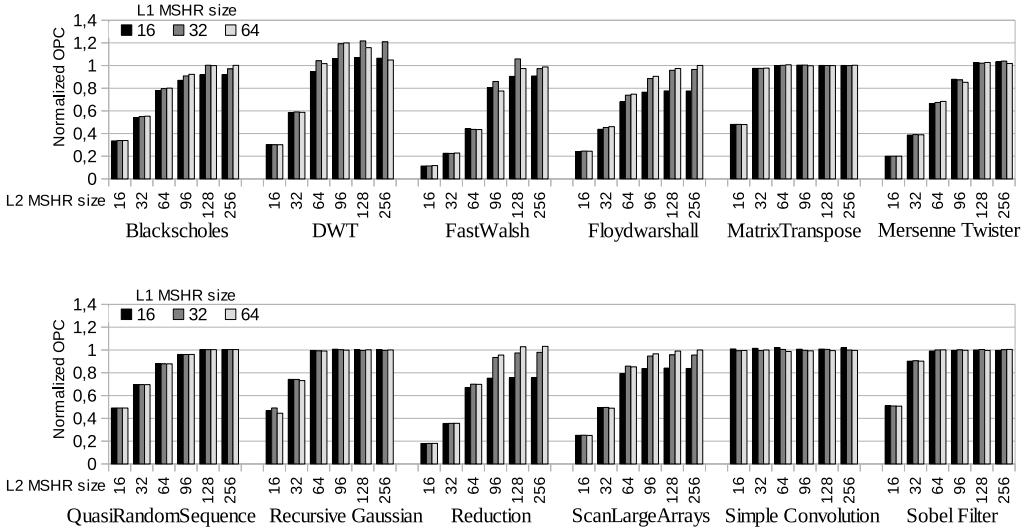

Figure 4: Impact of L1 and L2 MSHR file sizes on performance.

factor.

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

5.1. 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 [21][33] 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 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 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.

5.2. 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 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

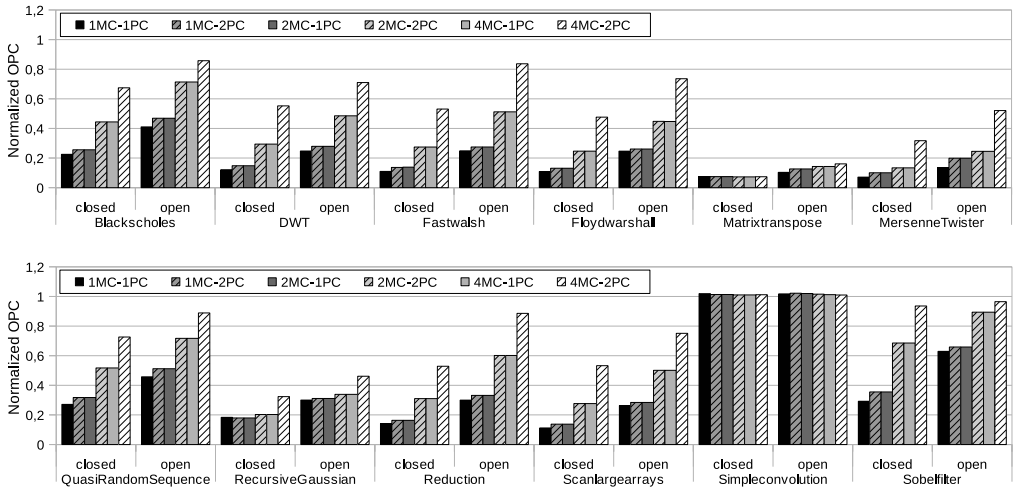

Figure 5: Impact of the number of memory controllers, physical channels, and page policy on performance.

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 [34], 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

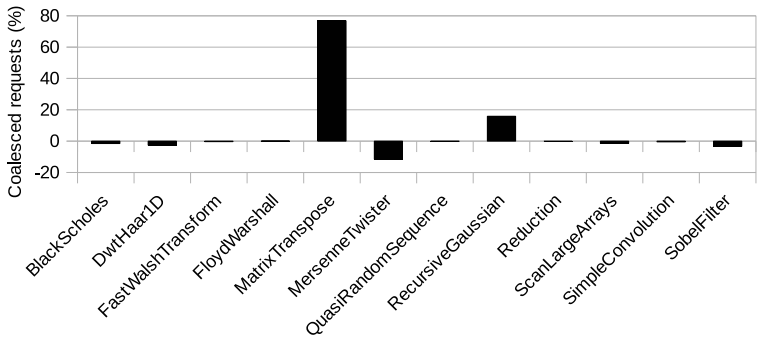

Figure 6: Percentage of combined memory requests by coalescing with respect to merging.

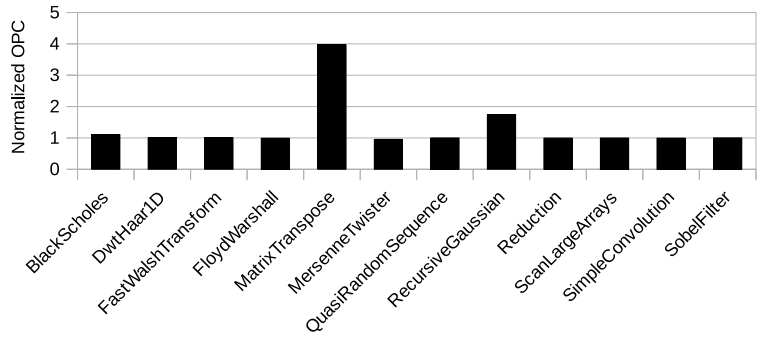

Figure 7: Speedup of coalescing with respect to merging.

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.

5.3. Memory Request Coalescing Mechanisms

This section compares the impact of coalescing versus merging on the obtained performance. Figure 6 and Figure 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,

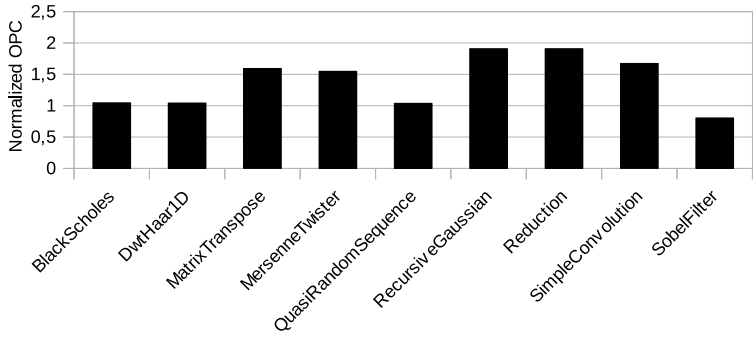

Figure 8: Impact of the coherence protocol: SI over NMOESI.

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.

5.4. Cache Coherence Protocol

In Section 4.4 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 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.

6. 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 [30], 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 [21] and [33].

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.

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 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.

7. 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,

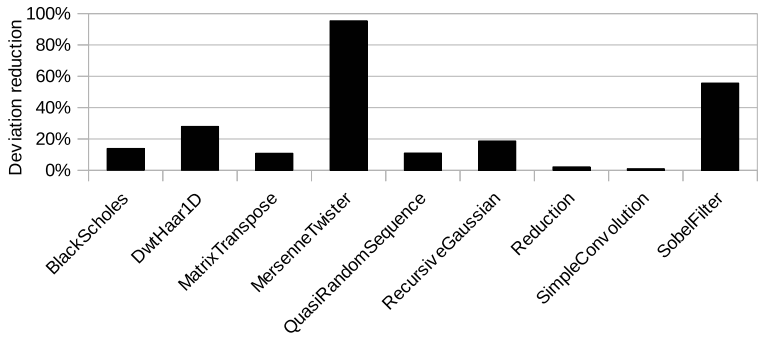

Figure 9: Reduction of execution time deviation between original and alltogether Multi2Sim.

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.

Acknowledgments

This work was supported by the Spanish Ministerio de Economía y Competitividad (MINECO), by FEDER funds under Grant TIN2012-38341-C04-01, and by the Intel Early Career Faculty Honor Program Award.

Authors would like to thank Rafael Ubal, Dana Schaa, and Amir Kavyan Ziabari for this help during the development of this work.

References

[1] S. Huang, S. Xiao, W. Feng, On the energy efficiency of graphics processing units for scientific computing, in: Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, 2009, pp. 1–8. doi:10.1109/IPDPS.2009.5160980.

[2] Top500 Supercomputer Sites, http://top500.org.

[3] Q. Huang, Z. Huang, P. Werstein, M. Purvis, 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.

[4] A. Branover, D. Foley, M. Steinman, Amd fusion apu: Llano, IEEE Micro 32 (2) (2012) 28–37. doi:10.1109/MM.2012.2.
URL `http://dx.doi.org/10.1109/MM.2012.2`

[5] R. Ubal, B. Jang, P. Mistry, D. Schaa, D. Kaeli, Multi2sim: A simulation framework for cpu-gpu computing, in: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12, ACM, New York, NY, USA, 2012, pp. 335–344. doi:10.1145/2370816.2370865.
URL `http://doi.acm.org/10.1145/2370816.2370865`

[6] W. Fung, I. Sham, G. Yuan, T. Aamodt, 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.

[7] A. Bakhoda, G. Yuan, W. Fung, H. Wong, T. Aamodt, Analyzing cuda workloads using a detailed gpu simulator, in: Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, 2009, pp. 163–174. doi:10.1109/ISPASS.2009.4919648.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen,

K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, The gem5 simulator, SIGARCH Comput. Archit. News 39 (2) (2011) 1–7. doi:10.1145/2024716.2024718.
URL http://doi.acm.org/10.1145/2024716.2024718

[9] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, V. J. Reddi, Gpuwattch: Enabling energy optimizations in gpgpus, SIGARCH Comput. Archit. News 41 (3) (2013) 487–498. doi:10.1145/2508148.2485964.
URL http://doi.acm.org/10.1145/2508148.2485964

[10] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, N. Jouppi, 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.

[11] AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK) .
URL http://developer.amd.com/sdks/amdappsdk/

[12] S. Collange, M. Daumas, D. Defour, D. Parello, 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.

[13] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. Penry, O. Temam, N. Vachharajani, Unisim: An open simulation environment and library for complex architecture design and collaborative development, Computer Architecture Letters 6 (2) (2007) 45–48. doi:10.1109/L-CA.2007.12.

[14] R. Ubal, J. Sahuquillo, S. Petit, P. Lopez, 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.
URL http://dx.doi.org/10.1109/SBAC-PAD.2007.17

[15] Heterogeneous System Architecture foundation .
URL http://www.hsafoundation.com/standards/

[16] A. R. G. Technology, AMD Graphics Cores Next (GCN) Architecture White Paper (Jun. 2012).

[17] C. Nvidia, Nvidias next generation cuda compute architecture: Fermi, Comput. Syst 26 (2009) 63–72.

[18] Khronos Group, OpenCL-The open standard for parallel programming of heterogeneous systems.
URL http://www.khronos.org/opencl

[19] J. E. Stone, D. Gohara, G. Shi, OpenCL: A parallel programming standard for heterogeneous computing systems, Computing in science & engineering 12 (1-3) (2010) 66–73.

[20] Khronos Group, The OpenCL Specification (Nov. 2015).
URL https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf

[21] W. Jia, K. A. Shaw, M. Martonosi, MRPB: memory request prioritization for massively parallel processors, in: 20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014, 2014, pp. 272–283. doi:10.1109/HPCA.2014.6835938.
URL http://dx.doi.org/10.1109/HPCA.2014.6835938

[22] F. Candel, S. Petit, J. Sahuquillo, J. Duato, Accurately modeling the gpu memory subsystem, in: High Performance Computing Simulation (HPCS), 2015 International Conference on, 2015, pp. 179–186. doi:10.1109/HPCSim.2015.7237038.

[23] C. J. Lee, O. Mutlu, V. Narasiman, Y. N. Patt, Prefetch-aware dram controllers, in: Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 41, IEEE Computer Society, Washington, DC, USA, 2008, pp. 200–209. doi:10.1109/MICRO.2008.4771791.
URL http://dx.doi.org/10.1109/MICRO.2008.4771791

[24] B. Jacob, S. Ng, D. Wang, Memory Systems: Cache, DRAM, Disk, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[25] P. Rosenfeld, E. Cooper-Balis, B. Jacob, Dramsim2: A cycle accurate memory system simulator, IEEE Comput. Archit. Lett. 10 (1) (2011) 16–19. doi:10.1109/L-CA.2011.4.
URL http://dx.doi.org/10.1109/L-CA.2011.4

[26] Evergreen Family Instruction Set Architecture .
URL http://developer.amd.com/wordpress/media/2012/10/AMD_Evergreen-Family_Instruction_Set_Architecture.pdf

[27] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, D. Ginsburg, OpenCL Programming Guide, 2nd Edition, Addison-Wesley Professional, 2013.

[28] P. Sweazey, 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, IEEE Computer Society Press, Los Alamitos, CA, USA, 1986, pp. 414–423.
URL http://dl.acm.org/citation.cfm?id=17407.17404

[29] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy, Memory consistency and event ordering in scalable shared-memory multiprocessors, SIGARCH Comput. Archit. News 18 (2SI) (1990) 15–26. doi:10.1145/325096.325102.
URL http://doi.acm.org/10.1145/325096.325102

[30] AMD, AMD Accelerated Parallel Processing OpenCL Programming Guide, http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf (Dec. 2013).

[31] Southern Islands Series Instruction Set Architecture .
URL http://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf

[32] AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK) .
URL http://developer.amd.com/sdks/amdappsdk/

[33] C. Nugteren, G.-J. van den Braak, H. Corporaal, H. Bal, 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.

[34] P. Navarro, V. Selfa, J. Sahuquillo, M. E. Gómez, C. G. Requena, 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.