Heterogeneous CPU/ (GP) GPU Memory Hierarchy Analysis and Optimization

Josué Vladimir Quiroga Esparza

Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

Advisor:

Ramon Canal Corretger

Universitat Politècnica de Catalunya

A thesis submitted in fulfillment of the requirements for the degree of

Master in Innovation and Research in Informatics (MIRI-HPC)

July 10th, 2015

To my parents, José and Patricia, for unconditionally provide their love and support.

Abstract

Heterogeneous systems, more specifically CPU – GPGPU platforms, have gained a lot of attention due to the excellent speedups GPUs can achieve with such little amount of energy consumption. Anyhow, not everything is such a good story, the complex programming models to get the maximum exploitation of the devices and data movement overheads are some of the constraints or challenges in order to get the benefits from GP-GPU computing.

On the other hand, architects from big processor manufacturers like Intel and AMD have integrated the CPU's and GPU's on the same chip thanks to the "Moore's Law" but the logical integration has not been as easy as putting them physically together side-by-side on the same die. Fusing these two different kind of cores, each one with its own memory hierarchy: one with higher memory bandwidth due to the throughput, on GPU's for example, and the CPU's with multi-level, higher capacity caches using protocols to provide strong consistency models for the programmer less scalable due to the coherency-related traffic.

With this, the Heterogeneous System Architecture (HSA) has been developed by the HSA Foundation founded ARM, AMD, Qualcomm and many other companies to reduce latency between devices connectivity, and make this system more compatible from a programmer's perspective(CUDA or OpenCL), without doing copies on disjoint memories, giving as result a Unified Virtual Memory. Because of the nature of these two separated memory systems, the heterogeneous Uniform Memory Access (hUMA) was created by AMD to share the system's virtual memory space. The GPU can access CPU memory addresses directly, allowing both reading and writing data that the CPU is also accessing sharing page tables so devices can exchange data by sharing pointers.

Great improvements can be achieved by the architecture integration on-chip, but memory wall is always present and a big constraint for systems with a lot of memory bandwidth demands as GPU does. Memory Controllers are the main character in scene to coordinate and schedule all the request of the processor to go to main memory, off chip, taking into account the technology latencies, refreshes, etc. It has too many constraints and to many scheduling possibilities that are impossible to have a general formula to schedule a processor requests to main memory so the flavors vary from processor to processor.

In this master thesis, we propose a scheduling re-ordering based on a hysteresis detector to give some fairness and speedup to the memory request threads taking advantage of the bank level parallelism at the memory system organization. First we introduce the evolution of the CPU and GPU processors until their integration in systems and processor using GPU as a general purpose

processor. Later we take a closer look to a Memory Controller giving the general perspective and functional elements with a state-of-the-art memory controllers for multicore processors. Given this we show our proposal system for re-ordering with the hysteresis detection and re-ordering logic. Then, the methodology about the simulation infrastructure and benchmarks used is described. The analysis of a baseline processor without memory unification, a fusion processor with virtual memory unification and this same fusion processor with the proposal scheduling for bank parallelism awareness. Conclusions derive from the result at the analysis are stated and the future work.

Index

List of Figures

List of Tables

1 Introduction

Fifty years have passed for the famous technical article at the "Electronics" magazine where Gordon Moore did the prediction for future computers with an extrapolation he did, that every year the numbers of elements on chip were duplicating [1]. He estimated chips going from 60 elements to 60,000 in ten years. This prediction was treated as a law ten years later, in 1975, thanks to Professor Carver Andress Mead from Caltech who popularized the term "Moore's law".

The integration of more and more elements on the same chip helped to make a faster and complex processor. Adding co-processors like a floating point unit (x87) was one of the most relevant examples, but the last decade we have integrated several of these complex cores doing the multicore era a reality. This transition from single processor to multicore gave a lot of challenges to processor architects in various ways: communication, programmability, memory system, technology, etc.

Evolution is continuing with the integration of multiple kinds of processors [2] (MIC's, GPU, DSP, etc.) on-chip. Making a new step in this evolution, Heterogeneous Chip Multiprocessors allows the usage of different processor architectures to better match the execution of the different application needs and to address wider spectrum of system loads with high efficiency. This new era has not been invented in one day. Big computing systems have been taking advantage of different kind of processors, accelerators (like the x87 for historical example), to better execute the tasks that were running, getting better performance than just having the general purpose processors. This kind of accelerators came in various flavors. Some using ASIC's with very detailed specifications (e.g. frequency of operation) and more recently using FPGA taking advantage of the possibility to describe multiple types of hardware on the same device, reconfiguring it as our system is evolving with newer components. Systems like these are called heterogeneous systems nowadays and this term has become very popular and as the name has become popular, a specific kind of coprocessor too: the Graphic Processor Unit (GPU). CPU's and GPU's have an important role in heterogeneous systems since a few years ago thanks to ability of GPU's to do heavy multithreaded workloads. The complex (requires a level of programming expertise to achieve the best performance), programming models like CUDA and OpenCL have not limited the popularity of them. Supercomputers around the world are widely using this heterogeneous CPU-GPU systems, one big example is the Titan Supercomputer in Oakridge Tennessee, number two at the Top 500 list (Jun 2015), scientists are getting use to program their formulas and models to fit in these systems making the GPU a General Purpose (GP) GPU, rather than a graphic processor.

But having this kind of systems working together on a same chip gives a lot of challenges. In the memory hierarchy, for example, with coherence and consistency between caches of both CPU and GPU. The off-chip accesses for main memory are one of the most important bottlenecks[3], more precisely the scheduling of these accesses are an important challenge due to the diversity of the throughput that each kind of processor, CPU and GPU, and bandwidth is always limited due to the number of pins to communication purposes. There is not a scheduling technique that can give a best fit for all the kind of varieties of configurations that exists on a chip that's why through this work we analyze a heterogeneous processor memory controller to identify bottlenecks and give a rescheduling proposal.

1.1 CPU Evolution

1.1.1 Single Core Era

Historically, the first processors were very simple arithmetic units that interacted to memory registers, also very limited in space, which worked at Kilohertz frequencies to compute simple operations faster than a human could, but with a limited set of instructions and a complex way to program through the assembly language.

As the prediction of Moore stated, more and more devices were integrated to add capabilities to the processing unit, separating the stages of processing (pipeline), adding more operations and interactions with registers to extend the ISA so the user could have better performance on their programs. This meant the development of the firsts programming languages. These processors start being very complex with a lot of extensions and big memory systems.

1.1.2 Multicore Era

Single core processors evolved very well thanks that processor architects could make a lot of improvements on the microarchitecture but the programmers were still demanding more computational power due to the complex applications that emerged so arranges of multiple processor where made to create clusters and "supercomputers" that made all the array of processors work as a big one. This opened the door to create the modern Multicore processor by the early 2000's and now they are widely used: supercomputing, workstations, personal computers, mobile devices, etc.

1.2 GPU Evolution

The primary ancestors of GPUs are graphics accelerators. Before the mid-90's PC graphics scaling was almost nonexistent. The Video Graphics Array (VGA) controller was a simply memory controller and a display generator connected to a DRAM. But later in that decade, three-dimensional (3D) capable accelerators started to appear and VGA controllers began to incorporate these functions including hardware for triangle setup, rasterization (dicing triangles into individual pixels), texture mapping and shading (applying "decals" or patterns to pixels and blending colors).

Continuing the years and evolution, at the beginning of this millennial (2000's) this chip that started as a graphics controller, incorporated almost every detail of the traditional graphics pipeline and therefore it deserved a new term: the GPU, to denote that the graphics device had become a processor.

1.2.1 GPGPU

This early graphics hardware was configurable but not programmable. The NVIDIA GeForce 3 took the first step toward true general shader programmability [4]. Mapping general computations to a GPU in this era was quite awkward. Nevertheless, intrepid researchers demonstrated a handful of useful applications with painstaking efforts. This field was called "GPGPU" for general-purpose computing on GPUs.

1.2.2 GPU Computing and CUDA

When researchers start solving nongraphic problems on a GPU was a huge challenge and a claim for a new programming model to take advantage of the massive amount of parallel processing power (e.g. large amount of floating-point units) and the Compute Unified Device Architecture (CUDA) appeared. CUDA is a scalable parallel programing model and a software platform for the GPU and other parallel processors that allows the programmer to bypass the graphics API and graphics interfaces of the GPU and simply program in C or C++. A brief explanation of the programming flow of CUDA can be seen in Annex A.

1.3 Heterogeneous CPU – GPU Systems

A system with a CPU and a GPU for heterogeneous multiprocessing takes advantage from these two different processors: The CPU, as a host, executes the application and it offloads massive parallel tasks to the GPU. This interaction from host to device and vice versa takes place thanks to copies of data and instructions from CPU to GPU. Then, the GPU computes data and copies the results back to the host. This is, in short, how most these kind of architectures are implemented taking the best from both worlds. In modern supercomputer systems, is possible to see the trend of heterogeneous configurations.

The two most important lists for supercomputers are Top 500 and Green 500. We list the top 10 from each of them (as of November 2014).

Modern Intel MIC Xeon Phi and NVIDIA's GPUs are the most popular devices used. At the Top500 list, we can see that only 3 of the supercomputers ranked are using GPUs: Titan, Piz Daint & Cray CS-Storm. In contrast, at the Green500 list we can see that 8 of the top 10 systems are using GPUs.

Let's remember that the Green500 list ranks each system's computing power in relation to its energy consumption, better said, those computers at this rank are the most efficient in electricity consumption per FLOP. Thus GPUs are seen as a very efficient device to gain FLOPs over the energy consumption. And going back to the Top500 list, Titan supercomputer is in a decent second place in the rank. To explain the configuration of each of the systems using GPUs in those lists would take a lot of chapters, and it is also not the target on this work. Nevertheless, Annex A describes how an AMD CPU and an NVIDIA GPU are interconnected in the Titan Supercomputer.

Green 500					
Ranking Nov 2014	**Name**	**Country**	**System**	**Power (KW)**	**RPEAK (MFLOP/W)**
1	**L-CSC**	**Germany**	ASUS ESC4000 FDR/G2S, Intel Xeon E5-2690v2 10C 3GHz, Infiniband FDR, AMD FirePro S9150	57.15	5,271.81
2	**Suiren**	**Japan**	ExaScaler 32U256SC Cluster, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, PEZY-SC	37.83	4,945.63
3	**TSUBAME-KFC**	**Japan**	LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, ***NVIDIA K20x***	35.39	4,447.58
4	**Storm1**	**US**	Cray CS-Storm, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, ***NVIDIA K40m***	44.54	3,962.73
5	**Wilkes**	**UK**	Dell T620 Cluster, Intel Xeon E5-2630v2 6C 2.600GHz, Infiniband FDR, ***NVIDIA K20***	52.62	3,631.70
6	**iDataPlex DX360M4**	**France**	Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband, ***NVIDIA K20x***	54.60	3,543.32
7	**HA-PACS TCA**	**Japan**	Cray CS300 Cluster, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband QDR, ***NVIDIA K20x***	78.77	3,517.84
8	**Cartesius Accelerator Island**	**Netherlands**	Bullx B515 cluster, Intel Xeon E5-2450v2 8C 2.5GHz, InfiniBand 4× FDR, ***NVIDIA K40m***	44.40	3,459.46
9	**Piz Daint**	**Switzerland**	Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , ***NVIDIA K20x***	1753.66	3,185.91
10	**Romeo**	**France**	Bull R421-E3 Cluster, Intel Xeon E5-2650v2 8C 2.600GHz, Infiniband FDR, ***NVIDIA K20x***	81.41	3,131.06

Table 1-1: Top ten supercomputers at Green500 list

TOP 500					
Ranking Nov 2014	**Name**	**Country**	**System**	**Power (KW)**	**RPEAK (TFLOP/S)**
1	**Tianhe-2 (MilkyWay-2)**	**China**	TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, ***Intel Xeon Phi*** 31S1P	17,808	54,902.4
2	**Titan**	**US**	Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, ***NVIDIA K20x***	8,209	27,112.5
3	**Sequoia**	**US**	BlueGene/Q, Power BQC 16C 1.60 GHz, Custom	7,890	20,132.7
4	**K Computer**	**Japan**	PARC64 VIIIfx 2.0GHz, Tofu interconnect	12,660	11,280.4
5	**Mira**	**US**	BlueGene/Q, Power BQC 16C 1.60GHz, Custom	3,945	10,066.3
6	**Piz Daint**	**Switzerland**	Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , ***NVIDIA K20x***	2,325	7,788.9
7	**Stampede**	**US**	PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, ***Intel Xeon Phi*** SE10P	4,510	8,520.1
8	**JUQUEEN**	**Germany**	BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect	2,301	5,872.0
9	**Vulcan**	**US**	BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect	1,972	5,033.2
10	**Cray CS-Storm**	**US**	Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, ***NVIDIA K40***	1,499	6,131.8

Table 1-2: Top ten supercomputers of Top500 list

1.4 Heterogeneous CPU – GPU Processors

At the beginning of this decade, it started to be common to see processors with the graphic processing unit integrated on the same die as the CPU [5]. This GPU has been used for graphic purposes mainly. Intel and AMD have been the main vendors to see this trend in the latest chips but we are making more emphasis at the AMD Advance Processing Units as they have been designed as general purpose GPUs.

On Intel's side, "HD Graphics" integrated the graphics inside the same chip with the multicore processor to get more energy efficiency for mobile devices. This integration and the advance of graphics with 2k and 4k video, force the creation of the latest version called "Iris Pro Graphics" since 2013 for more rendering support. Lately this integration of GPUs just has helped to video and graphics, and not for general computing.

The term Advance Processing Unit [5] was created to name the new series of microprocessor from AMD designed to act as a CPU and a graphics accelerator GPU on a single chip. The first design release was Fusion on 2011 and since then, with the newest Kaveri design from 2014, they have included more functionality. One of the key features for these chips, as we mention at the beginning, was to develop a microprocessor to act as a CPU and a GPU on a single die, bur not only putting them together physically, also unifying their memory system with heterogeneous Uniform Memory Access (hUMA) to avoid the overhead of doing copies between the two separated memories. Getting this done was not an easy task. The vision started in 2012 with the creation of the HSA Foundation, with the collaboration of AMD, ARM, Qualcomm, Texas Instruments and many other companies, some specs and guidelines where developed by this foundation. An overview of hUMA can be found in the Annex A.

1.4.1 HSA Foundation

Talking about the Heterogeneous processors of AMD, it is impossible not to touch the topic about of the Heterogeneous System Architecture (HAS) Foundation.

It was founded by big companies: AMD, ARM, Qualcomm, Texas instruments, Mediatek, Samsung and Imagination and it is a Non-Profit industry standards body focused on making it dramatically easier to program heterogeneous computing devices. The consortium comprises various software vendors, IP providers, and academic institutions and develops royalty-free standards and open-source software.

The HSA Foundation seeks to create applications that seamlessly blend scalar processing on the CPU, parallel processing on the GPU, and optimized processing on the DSP via high bandwidth shared memory access enabling greater application performance at low power consumption. The HSA Foundation is defining key interfaces for parallel computation utilizing CPUs, GPUs, DSPs, and other programmable and fixed-function devices, thus supporting a diverse set of high-level programming languages and creating the next generation in general-purpose computing.

The final version 1.0 for Platform System architecture has been out since January 2015, and the features that AMD achieved with Kaveri architecture are stated here like the unified virtual memory and cache coherency.

1.5 Motivation and Objectives

The review of the integration on-chip for mobile devices last ten years and also the advances on programming models for faster and more tightly connected processors for supercomputers are converging on dynamic processors where the communication to the external world is always limited in bandwidth due to the pin connections available. It is, thus, important to provide an analysis of the memory system for integrated CPU – GPU processors on die for heavily parallel applications. More specifically the memory controller for these future processors that will have a multicore CPU and general purpose GPUs, stressing the off-chip petitions for main memory.

The Objectives of this work starts with the analysis of a baseline processor where we are integrating both CPUs and GPUs using the same physical memory but not shared, this means that we will have two different memory controllers addressing two different parts of the physical memory doing necessaries copies of data and instructions between both types of processor as the actual processing model of CUDA does. Later, we use the unified memory access design, so they start sharing the same memory controller and see the same memory space. This design also establishes the elimination of unnecessary copies of data and instruction like the normal processing model of CUDA.

The results of both interactions, with separated memories and unified memory will give us the advantage of the elimination of copies between host and device and how the numbers of petitions to a memory controller behave.

The behavior will give us an idea to propose a reordering in the scheduling model that the memory controller is using. We will implement it and compare the results of the executions with the ones before doing the re-scheduling proposal. At the end we want to compare the evolution of the 3 different memory systems to show the actual pros and cons so we can relate a future work.

In summary, the main objectives are:

1. An analysis of the memory system, more specifically the memory controller orchestration on a baseline system to show the actual programming model flow for heavily parallel programming (CUDA) selected from the Rodinia suit.

2. Obtain a unified memory system where CPUs and GPUs can address the same memory system with the usage of the unified memory access and control all petitions on one memory controller.

3. Compare this last unified memory system without copies of data between devices and the baseline with copies to get some discussion towards a proposal in the reordering of the scheduling policies available to better fit the behavior of the heterogeneous processor.

4. Implement the reordering proposal on the memory controller and execute the selected benchmarks to compare the results checking the output against the memory controller before the implementation

5. Analyze the pros and cons obtained in the reordering proposal and do an evolution comparison from the baseline memory system with copies, the unified memory system without copies and the modified scheduling implementation proposed by us in order to visualize the future work.

2 The Memory Controller

At the beginning, processors had the memory controller off-chip. Neither was part of the memory component, it was an ASIC located in between the processor and memory. The integration on-chip of different modules on the processor die, made the memory controller part of these modules. But off-chip or on-die, the paper that the memory controller has stayed almost intact: manage the movement of data into and out of processors to DRAM devices while ensuring protocol compliance, accounting for DRAM-device-specific electrical characteristics, timing characteristics, and, depending on the specific system, even error detection and correction.

2.1 Memory system components

To better understand the role of the memory controller we have to give a brief overview of the memory system components and related latencies. We mentioned at the beginning, the memory controller was located between the memory modules, DRAM, and the CPU, GPU or other components on chip that need communication to the memory (Figure 1 illustrates the concept).

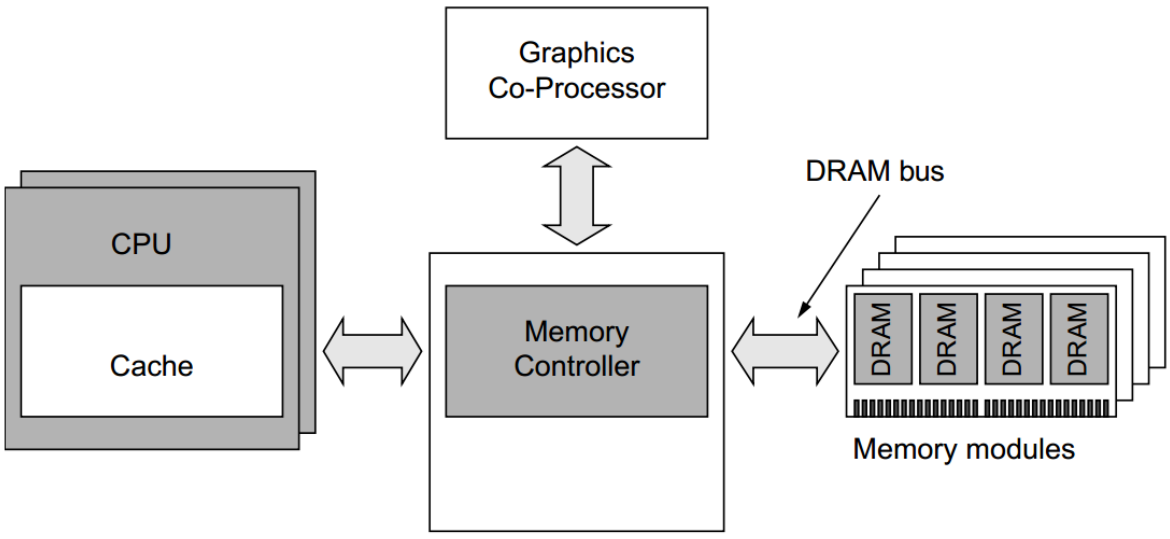


Figure 1: Memory Controller principal connection [7]

The memory controller connection to the memory module is a DRAM bus sends or receives: Data, Address/Command and Chip selection. (See Figure 2)

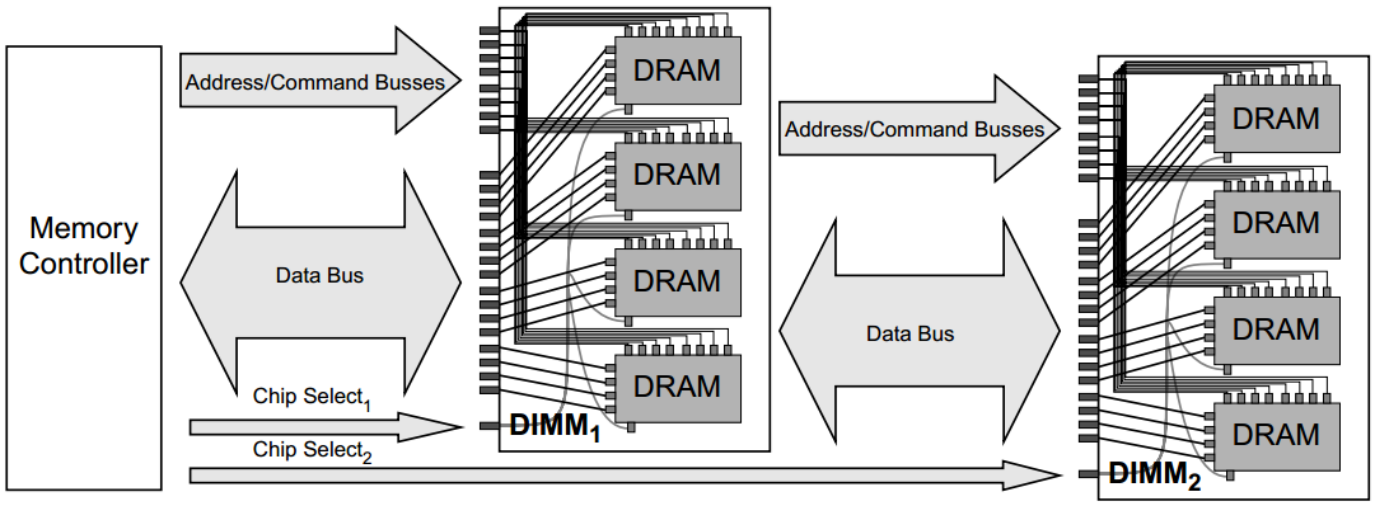


Figure 2: Memory Controller Busses [7]

Memory components of DRAM are DIMMs, Ranks, Banks and Memory Arrays as shown in Figure 3

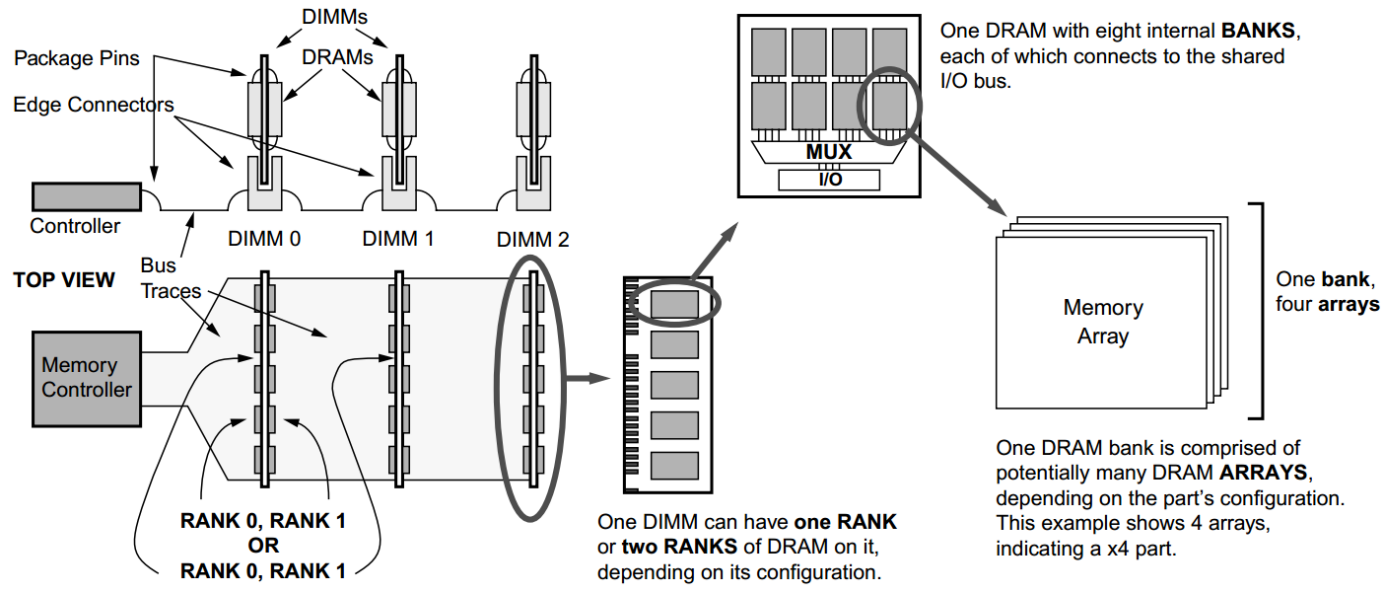


Figure 3: Memory components at a DRAM [7]

2.2 Memory Controller

The design of an optimal memory controller must consist of system-level considerations that ensure fairness in arbitration for access between the different agents (i.e. CPUs and GPUs in our case) that read and store data in the same memory system.

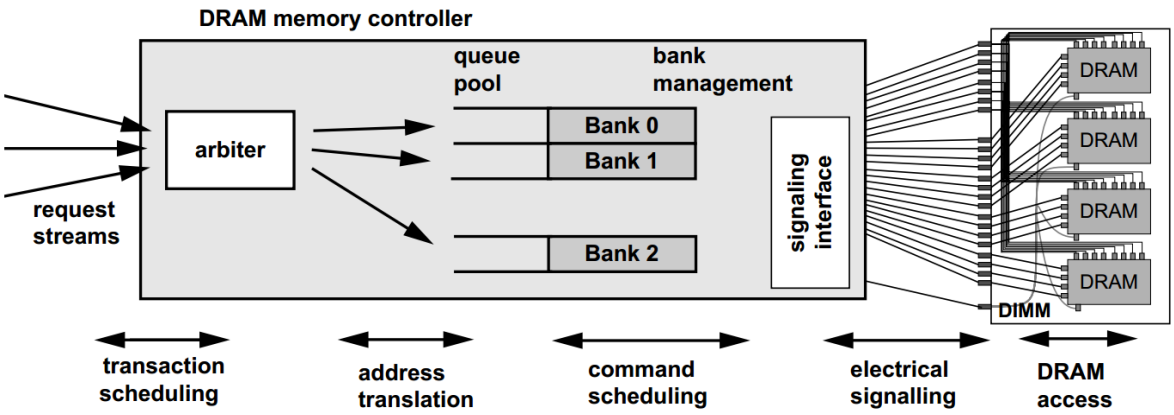


Figure 4: Abstract DRAM Memory Controller [7]

Figure 4 illustrates some basics of an abstract DRAM memory controller. Starting from the left of the image, the memory controller receives the requests from one or multiple microprocessors (e.g. multicore) and it provides the arbitration to determine which request agent will be able to place its request into the memory controller. From a certain perspective, the request arbitration logic may be considered as part of the system controller rather than the memory controller. However, as the cost of memory access continues to increase relative to the cost of data computation in modern processors; efforts in performance optimizations are combining transaction scheduling and command scheduling policies and while examining them in a collective context rather than separate optimizations. Once a transaction wins arbitration it enters into the memory controller, it is mapped to a memory address location and converted to a sequence of DRAM commands.

The sequence of commands is placed in queues in the memory controller. The queues may be arranged as a generic queue pool, where the controller will select from pending commands to execute, or the queues may be arranged so that there is one queue per bank or per rank of memory. Then, depending on the DRAM command scheduling policy, commands are scheduled to the DRAM devices through the electrical signaling interface. Although the electrical signaling interface may be one of the most critical components in modern, high data rate memory systems, the challenges of signaling are not examined.

2.2.1 Row-Buffer-Management

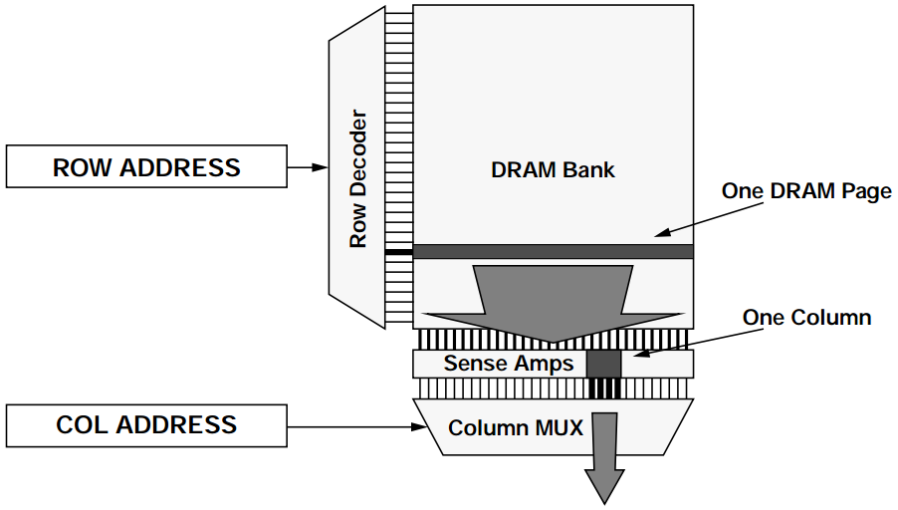


Figure 5: Location of Sense Amplifiers at a DRAM Bank [7]

In DRAM devices, the arrays of sense amplifiers can also act as buffers that provide temporary data storage. Policies that manage the operation of sense amplifiers are referred to as row-buffer-management policies. The two primary row-buffer-management policies are the open-page policy and the close-page policy, and depending on the system, different row-buffer-management policies can be used to optimize performance or minimize power consumption of the DRAM memory system. In cases where the memory access sequence possesses a high degree of temporal and spatial locality, memory system architects and design engineers can take advantage of the locality by directing temporally and spatially adjacent memory accesses to the same row of memory. The open-page row-buffer-management policy is designed to favor memory accesses to the same row of memory by keeping sense amplifiers open and holding a row of data for ready access. In a DRAM controller that implements the open-page policy, once a row of data is brought to the array of sense amplifiers in a bank of DRAM cells, different columns of the same row can be accessed again with the minimal latency. In contrast to the open-page row-buffer-management policy, the close-page row-buffer-management policy is designed to favor accesses to random locations in memory and optimally supports memory request patterns with low degrees of access locality. The close-page policy is typically deployed in memory systems designed for large processor count, multiprocessor systems or specialty embedded systems. The reason that an open-page policy is typically deployed in memory systems of low processor count platforms while a close-page policy is typically deployed in memory systems of larger processor count platforms is that in large systems, the intermixing of memory request sequences from multiple, concurrent, threaded contexts reduces the locality of the resulting memory-access sequence.

2.2.2 Parallelism in Memory System Organization: Banks

Consecutive memory accesses can proceed in parallel to different banks of a given rank subject to the availability of the shared address, command, and data busses. In contemporary DRAM devices, scheduling consecutive DRAM read accesses to different banks within a given rank is, in general, more efficient than scheduling consecutive read accesses to different ranks. Read requests tend to have higher spatial locality than write requests due to the existence of write-back caches. Moreover, the number of column-read commands that immediately follow column-write commands can be minimized in advanced memory controllers by deferring individual write requests and instead group schedule them as a sequence of consecutive write commands.

In performance-optimized, open-page memory systems, adjacent cache line addresses are striped across different channels so that streaming bandwidth can be sustained across multiple channels and then mapped into the same row, same bank, and same rank. This is assuming a uniform memory system where all channels have identical configurations in terms of banks, ranks, rows, and columns.

Similar to the baseline address mapping scheme for open-page memory systems, consecutive cache line addresses are mapped to different channels in a close-page memory system. However, unlike open-page memory systems, mapping cache lines with sequentially consecutive addresses to the same bank, same rank, and same channel of memory will result in sequences of bank conflicts and greatly reduce available memory bandwidth. To minimize the chances of bank conflict, adjacent lines are mapped to different channels, then to different banks, and then to different ranks in close-page memory systems.

3 State-of-the-Art

The adoption of multicore processors last few years, gives a lot of multiple flavors of memory access scheduling. Memory and system throughput maximization are the main goals constrained by fairness on the execution threads. From the variety of works out there we are presenting two memory access scheduling for chip multiprocessors with parallelism-aware and fairness for stall time both important for understanding our proposal explained in the next section.

3.1 First-Ready First Come-First Served

The First-Ready First Come-First Served (FR-FCFS) [8] has been shown to be the best performing one overall in single-threaded systems. The DRAM command prioritization policies employed by the FRFCFS algorithm are unfair to different threads due to two reasons.

First, the column-first policy gives higher priority to threads that have high row-buffer locality: If a thread generates a stream of requests that access different columns in the same row, another thread that needs to access a different row in the same bank will not be serviced until the first thread's column accesses are complete. Second, the oldest-first policy implicitly gives higher priority to threads that can generate memory requests at a faster rate than others. Requests from less memory-intensive threads are not serviced until all earlier-arriving requests from more memory-intensive threads are serviced. Therefore, less memory-intensive threads suffer relatively larger increases in memory-related stalls.

3.2 Stall-Time Fair Memory Access Scheduling

Stall-Time Fair Memory (STFM) [9] scheduler estimates two values for each thread: T_{shared} and T_{alone}. The processor increases a counter when it cannot commit instructions due to an L2-cache miss. This counter is communicated to the memory scheduler. Assuming for now that the STFM scheduler knows each thread's slowdown $S = T_{shared}/T_{alone}$, it uses the following policy to determine the next command to be scheduled:

1. Determine Unfairness: From among all threads that have at least one ready request in the request buffer, the scheduler determines the thread with highest slowdown (S_{max}) and the thread with lowest slowdown (S_{min}).

2. Could be two:

    a. Apply FR-FCFS-Rule: If the ratio $S_{max}/S_{min} \leq \alpha$, then the acceptable level of unfairness is not exceeded and, in order to optimize throughput, the next DRAM command is selected according to the FR-FCFS priority rules described before.

    b. Apply Fairness-Rule: If the ratio $S_{max}/S_{min} > \alpha$, then STFM decreases unfairness by prioritizing requests of thread T_{max} with largest slowdown S_{max}. In particular, DRAM commands are prioritized in the following order:

        i) Tmax-first: Ready commands from requests issued by T_{max} over any command from requests issued by other threads.

        ii) Column-first: Ready column accesses over ready row accesses.

        iii) Oldest-first: Ready commands from older requests over those from younger requests.

In other words, STFM uses either the baseline FR-FCFS policy (if the level of unfairness across threads with ready requests is acceptable), or a fair FR-FCFS policy in which requests from the most slowed-down thread receive highest priority.

The threshold α that denotes the maximum tolerable unfairness can be set by the system software via a privileged instruction in the instruction set architecture. If the system software does not need hardware-enforced fairness at the DRAM controller it can simply supply a very large α value.

Second, to support different treatment of threads based on their importance, we add the notion of thread weights to our mechanism. The system software conveys the weight of each thread to STFM. Threads with equal weights should still be slowed down equally. To support this notion of thread weights and to prioritize threads with larger weights, STFM scales the slowdown value computed for the thread by the thread's non-negative weight such that the weighted slowdown is $S=1+(S-1)*\text{Weight}$. That is, threads with higher weights are interpreted to be slowed down more and thus they are prioritized by STFM. For example, for a thread with weight 10, a measured slowdown of 1.1 is interpreted as a slowdown of 2 whereas the same measured slowdown is interpreted as 1.1 for a thread with weight 1. Note that even after this modification, it is the ratio S_{max}/S_{min} that determines whether or not the fairness-rule is applied. Measured slowdowns of equal-weight threads will be scaled equally and thus those threads will be treated equally by the scheduler.

3.3 Parallelism-Aware Batch Scheduling

Parallelism-Aware Batch Scheduling (PAR-BS) [10] consists of two components. The first component is a request batching (BS), or simply batching, component that groups a number of outstanding DRAM requests into a batch and ensures that all requests belonging to the current batch are serviced before the next batch is formed. Batching not only ensures fairness but also provides a convenient granularity (i.e., a batch) within which possibly thread-unfair but high-performance DRAM command scheduling optimizations can be performed. The second component, parallelism-aware within-batch scheduling (PAR) aims to reduce the average stall time of threads within a batch (and hence increase CMP throughput) by trying to service each thread's requests in parallel in DRAM banks.

The idea of batching is to consecutively group outstanding requests in the memory request buffer into larger units called batches. The DRAM scheduler avoids request re-ordering across batches by prioritizing requests belonging to the current batch over other requests. Once all requests of a batch are serviced (i.e., when the batch is finished), a new batch is formed consisting of outstanding requests in the memory request buffer that were not included in the last batch. The batching component (BS) of PAR-BS works as follows. Each request in the memory request buffer has an associated bit indicating whether the request belongs to the current batch. If the request belongs to the current batch, this bit is set, and we call the request marked

PAR-BS always prioritizes marked requests (i.e., requests belonging to the current batch) over non-marked requests in a given bank. On the other hand, PAR-BS neither wastes bandwidth nor unnecessarily delays requests: if there are no marked requests to a given bank, outstanding non-marked requests are scheduled to that bank. To select among two marked or two non-marked requests, any existing or new DRAM scheduling algorithm (e.g., FR-FCFS) can be employed.

Batching naturally provides a convenient granularity within which a scheduler can optimize scheduling decisions to obtain high performance. There are two main objectives that this optimization should strive for. It should simultaneously maximize:

1. Row-Buffer Locality.

2. Intra-thread Bank-Parallelism within a batch.

The first objective is important because if a high row-hit rate is maintained within a batch, bank accesses incur smaller latencies on average, which increases the throughput of the DRAM system. The second objective is similarly important because scheduling multiple requests from a thread to

different banks in parallel effectively reduces that thread's experienced stall-time. Unfortunately, it is generally difficult to simultaneously achieve these objectives.

Batch-Scheduling algorithm uses the request prioritization rules shown:

1. BS-Marked requests first: Marked ready requests are prioritized over requests that are not marked.

2. RH-Row-hit first: Row-hit requests are prioritized over row-conflict/closed requests.

3. RANK-Higher rank first: Requests from threads with higher rank are prioritized over requests from lower ranked threads.

4. FCFS-Oldest first: Older requests are prioritized over younger requests.

Batching guarantees the absence of short-term or long-term starvation: every thread can make progress in every batch, regardless of the memory access patterns of other threads. Also enables the use of highly efficient within-batch scheduling policies (such as PAR). Without batches (or any similar notion of groups of requests in time), devising a parallelism-aware scheduler is difficult as it is unclear within what context bank-parallelism should be optimized.

4 Hysteresis-Batch Memory Access Scheduling

To propose a new scheduling algorithm for a future Heterogeneous CPU-GPGPU Architecture on-chip we have to take a lot of constrains to maximize the throughput of memory with a good level of fairness in the system inside the chip. These constraints start at the physical level. Processors communicate to the external world through a limited number of pins. These pins cannot be added just because the willing of having them, totally the contrary. Electrical constrains for signal integrity due the high frequencies of operation rates is an example that restrict the distance between pin to pin and, by consequence, the number of pins barely increases from technology to technology. Given this situation, the easy path of adding multiple channels with more bandwidth was not part of our proposal and we just focus to give a proposal of a scheduling technique

A heterogeneous CPU-GPGPU memory controller scheduling can be understood as a request re-ordering mechanism for a multicore chip: a system where the number of requestors is more than two, with different frequencies and various threads. Multicore chips are well known so that's why we took the decision to explore the scheduling algorithms in this area (presented in the state-of-the-art)

Due to the high level of parallelism that GPGPUs can achieve, future heterogeneous CPU-GPGPU processors are going to need specific throughput and fairness policies tailored for their heterogeneous systems.

4.1 Proposal

CPU and GPGPU interaction is known as separated devices, the CPU acts like a host orchestrating the data movements to the GPGPU so it can compute the data delivered. The GPGPU then has a lot of computation over the data received. When finished, the GPGPU returns the results to the CPU. This activity pattern is interesting information to the memory controller at the time of scheduling the request. If we know the requestor is possible to understand which agent is the responsible of the numerous petitions we are receiving.

To detect the most active component of the requests we propose a saturation counter of two-bits where we can define 4 different states as Figure 6 shows:

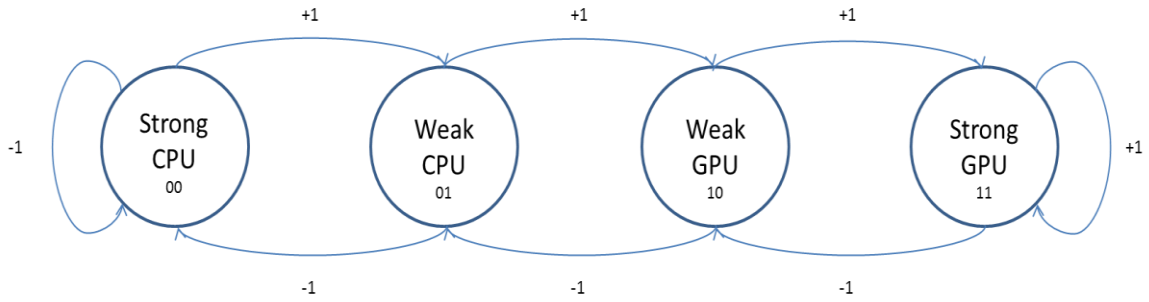


Figure 6: Two-bit saturation counter with the 4 different states.

This saturation counter is going to snoop the input at the memory controller, looking at all requests in order of arrival. Depending on the requester the saturation counter is going to give a behavior status delimiting if we are in a CPU or GPU intensive region. Inside of each region we can have a strong or weak behavior. This means that if the majority of the requests are from the CPU, for example, it would be at the strong region. But if we start a transition where the GPU requests start to be more often we will be in a weak CPU region. If the GPU requests are then more common, we will be transition first to the Weak GPU and then to the Strong GPU when the vast majority of the request seen at the input are from the GPU device.

The outstanding number of request that the CPU or GPU can have is fixed to avoid starvation. We decided to take advantage of the Batch Scheduling in the regions where the Hysteresis detector is in strong CPU or GPU so it can give priority to threads of this agent to enable the bank level parallelism. When in the weak regions, better start ranking threads indistinctly the agent so we can get the best parallelism of the CPU-GPU interactions. Given this, the Hysteresis-Batch scheduling algorithm is as follows:

1. HBS-Marked-requests-first: Hysteresis marked ready requests are prioritized over requests that are not marked.

2. RH-Row-hit-first: Row-hit requests are prioritized over row conflict/closed requests.

3. RANK-Higher-rank-first: Requests from threads with higher rank are prioritized over requests from lower-ranked threads.

4. FCDS-Oldest First: Older requests are prioritized over younger requests.

5 Methodology

To do the proper analysis and evaluate the different configurations of the heterogeneous CPU – GPU processor, the help of a simulation tool was needed. Basically our needs, related to the objectives of this study, were the following:

- A tool where the description of the CPU – GPU memory hierarchy could be easily implemented and changed.

- A tool able to run important benchmarks used for the evaluation of state-of-the-art heterogeneous systems.

These characteristics are generally different from the classic and conventional simulation tools. CPU and GP-GPGU simulators are generally two different tools used to work independently, and the options of stable and open tools to use are limited.

We performed and analysis and comparison of the available simulators for Heterogeneous Architectures: Barra-Sim [11], FusionSim [12], Multi2Sim [13] and gem5-gpu [14].

Feature	Barra-Sim	FusionSim	Multi2Sim	gem5-gpu
CPU Simulation	x	x	x	x
GPU Simulation	x	x	x	x
Cache Coherence	x	x	x	x
Memory Hierarchy	x	x	x	x
CUDA Support	x	x	x	x
OpenCL Support			x	x
OpenACC Support			x	x

Table 5-1: Comparison of characteristics across different simulators

Due to the limited support for languages that execute across heterogeneous platforms, we discard Barra-Sim and FusionSim. The option of using Multi2Sim was first evaluated.

The HSA consortium founded by many important companies mentioned in last chapter, some academic institutions are part of this Foundation too. One of them, Northeastern University is the developer of Multi2Sim so this encourages us to first evaluate its usage for our study.

Multi2Sim is a framework for the CPU – GPU simulation for heterogeneous computing written on C, so the ease of use and development was available. When we first started looking for the representative benchmarks for the evaluation of the heterogeneous processor we found the Rodinia

Suit. We decided to take it into the project, the bad news were that Muti2Sim was not yet capable of implement this suite of benchmarks.

Rodinia Benchmarks Suite [15] for heterogeneous computing was cited in numerous works (+600 at Google scholar) and multiple versions of the suite are available, enforcing its stability, development and usage in the community. Gem5-gpu confirmed the usage of the Rodinia Benchmarks with a couple of papers evaluating heterogeneous configurations recently [16][17][18]. The decision of changing the simulator was made and the usage of the tool started. A description of gem5-gpu simulator and its memory system descriptions are described in the next subchapter. Also the Rodinia Benchmarks selected from the suite are described

5.1 Gem5-gpu

Gem5-gpu is a new simulator that models tightly integrated CPU-GPU systems. It builds on gem5, a modular full-system CPU simulator, and GPGPU-Sim, a detailed GPGPU simulator. Gem5-gpu routes most memory accesses through Ruby, which is a highly configurable memory system in gem5. By doing this, it is able to simulate many system configurations, ranging from a system with coherent caches and a single virtual address space across the CPU and GPU to a system that maintains separate GPU and CPU physical address spaces. Gem5-gpu can run most unmodified CUDA 3.2 source code. Applications can launch non-blocking kernels, allowing the CPU and GPU to execute simultaneously. It is open source and available at gem5-gpu.cs.wisc.edu.

GPGPU-Sim models the compute architecture of modern NVIDIA graphics cards. GPGPU-Sim executes applications compiled to PTX (NVIDIA's intermediate instruction set) or disassembled native GPU machine code. GPGPU-Sim models the functional and timing portions of the compute pipeline including the thread scheduling logic, highly-banked register file, special function units, and memory system.

In the other hand, gem5 simulator is a very well-known infrastructure by the community and is a modular, system modeling tool developed by numerous universities and industry research labs. Include multiple CPU, memory systems and ISA models.

It provides two execution modes:

1. System call emulation, which can run the user level binaries using emulated system calls.

2. Full system, which models all necessary devices to boot and run, unmodified operating systems.

5.1.1 Architecture

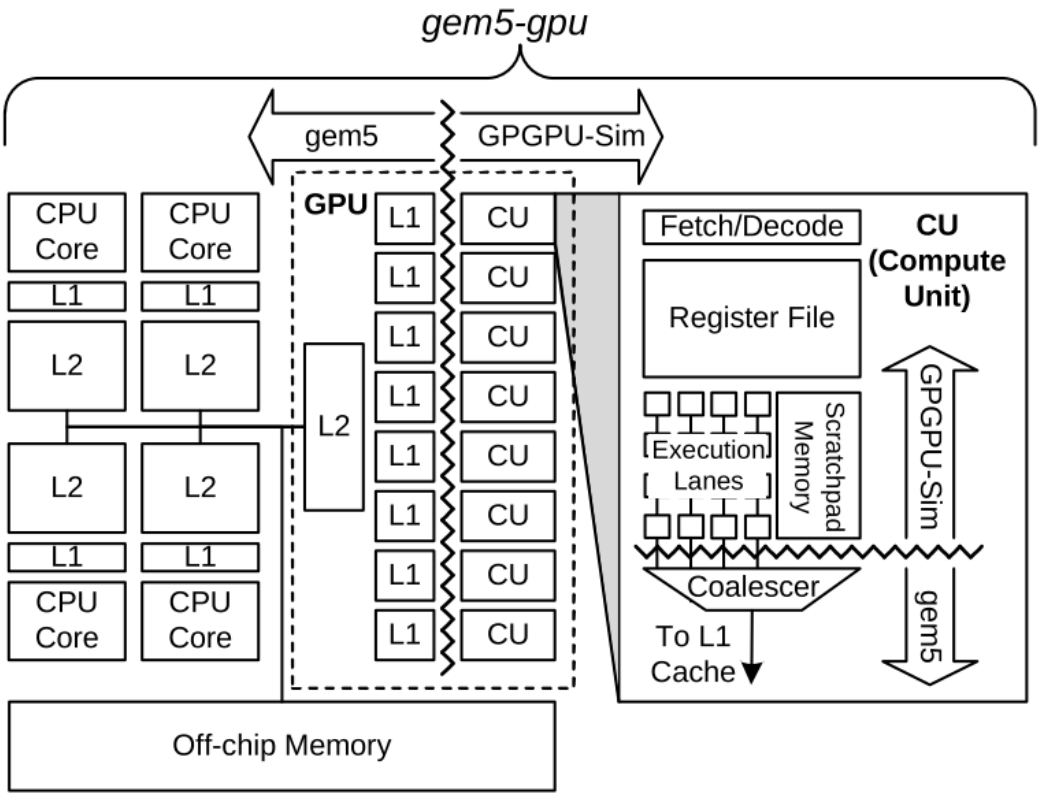


Figure 7: Overview of gem5-gpu architecture [14]

Figure 7 shows a four core CPU and an eight Compute Unit (CU) GPU integrated on the same chip. The number of CPUs, CUs, and topology connecting them is fully configurable, for the project we create a similar description: 4 CPU cores and 16 CU GPU. Two on-chip topologies that gem5-gpu provides out of the box are a shared and a split memory hierarchy (i.e., integrated and discrete GPUs, respectively). Many CUs make up the GPU, each of which has fetch/decode logic, a large register file, and many (usually 32 or 64) execution lanes. When accessing global memory, each lane sends its address to the coalescer, which merges memory accesses to the same cache block. The GPU may also contain a cache hierarchy that stores data from recent global memory accesses.

5.1.1.1 Software Architecture

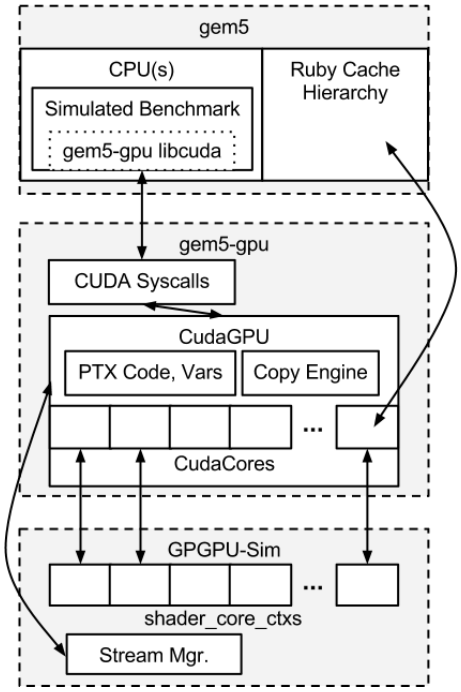


Figure 8: Current gem5-gpu software architecture (Source: gem5-gpu)

Figure 8 is a diagram of the gem5-gpu software architecture used for this, some descriptions to the important elements shown is:

- CudaGPU: Acts as the gem5 structure for organizing the GPU hardware as:
    - Contains the CudaCore(s).
    - Contains the logical Copy Engine (CE) to move data from CPU to GPU address spaces (when you described a “split” virtual memory).
    - Handles PTX code, variables and check pointing.
    - Handles the begin/end of the CUDA kernel.
    - Manages GPU memory space page table.
- CudaCore(s): Is a wrapper for GPGPU-Sim “shader_cores_ctx”. Sends instruction, global and constant memory request (data and instruction memory accesses) to Ruby cache hierarchy.

- CUDA Syscalls: Originally, gem5 supports decoding of non-ISA-specified instructions (pseudo-instructions) within a simulated benchmark. A pseudo-instruction called “m5_gpu” is introduce, which currently allows the CPU to trap and hand control over the CUDA syscalls. These instructions are built into libcuda in the simulated benchmark binary.

    - Once trapped into CUDA Syscalls, the appropriate CUDA call Is executed, which may interface with the GPU for managing memory or kernel handling. PTX code handling or the copy engine.

5.1.2 Memory System

Gem5-gpu uses Ruby to model both the functional and timing of most memory accesses. The load-store pipeline is modeled in gem5, including the coalescing, virtual address translation, and cache arbitration logic. By using the port interface in gem5, gem5-gpu has the flexibility to vary the number of execution lanes, number of CUs, cache hierarchy, etc. and incorporate other GPU models in the future. Currently, GPGPU-Sim issues only general-purpose memory instructions to gem5, including accesses to global. The GPGPU-Sim models memory operations to scratchpad and parameter memory.

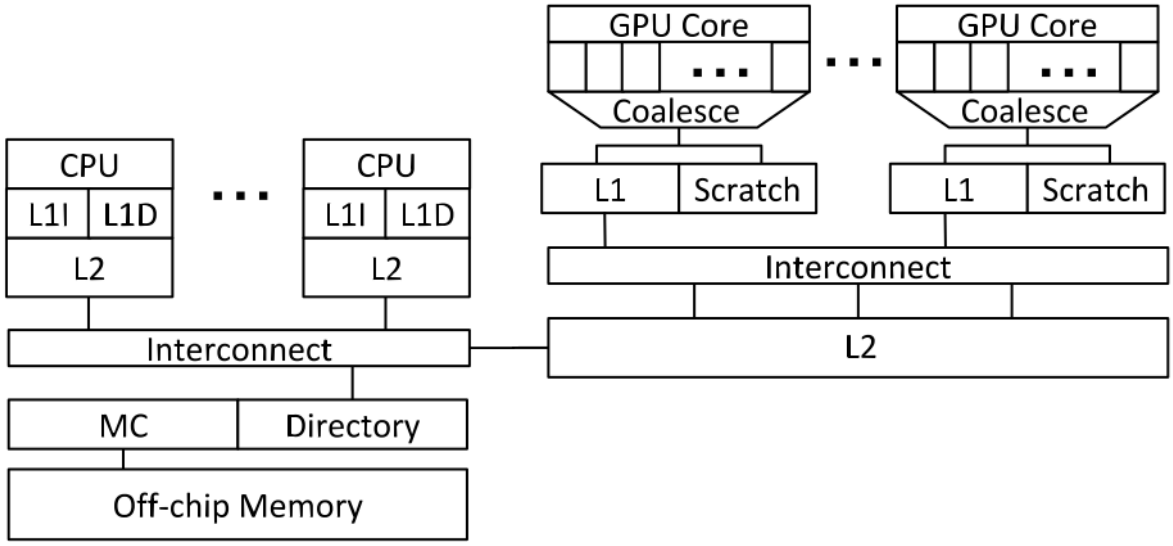


Figure 9: Interconnections for a fusion memory system example [16]

Gem5-gpu supports a shared virtual address space between the CPU and GPU (i.e. the GPU using the CPU page table for virtual to physical translations). Alternatively, through a configuration option, gem5-gpu models separate GPU and CPU physical address spaces.

Here is a trace of a memory operation through gem5:

1. GPGPU-Sim executes a Load/Store.
2. The warp-wide instruction is converted into lane operations and sent to a Load/Store Queue (LSQ) unit.
3. The LSQ gets the lane requests, coalesces them and then sends the request to the memory subsystem, Ruby in this case.
4. Ruby receives the request and simulates the cache hierarchy and memory (both timing and functional). The actual code that simulates the caches is automatically generated from the SLICC files.
5. Ruby returns the result after some amount of time to the LSQ, which in turn (on a load) returns the data to the CudaCore.
6. Finally, the CudaCore in gem5-gpu forwards the data back the actual core model in gpgpu-sim which (on a load) writes the data into a register.

5.2 Rodinia Benchmarks

Created to evaluate heterogeneous system with GPUs, it is also base of the latest works related to heterogeneous CPU-GPU processors on-chip. Gem5-gpu has them and also a version with a slightly modification at the memory copies in the CUDA Kernels. This modification is the removal of these memory copies from host to device and vice-versa, because at a fusion architecture where they are taking advantage of the unified virtual memory is unnecessary to do them. The nine selected benchmarks are: Back Propagation, Breath First Search, Hot Spot, K-means, Lava MD, Needleman-Wunsh, Particle Filter, Pathfinder and SRAD.

6 Analysis

In the previous chapter, we gave an overview of the tool gem5-gpu, where a unified memory system for CPUs and GPUs was implemented to have a shared virtual address space, similar to modern AMD APUs fusion. Also, as we mentioned, the tool has the possibility to do a “split” address space for CPU and GPU in order to have the necessity to do the memory copies between the two systems and have a similar behavior as the CUDA programming model doing copies of memory from host to device and vice versa. In order to compare both memory systems, the regular state-of the-art with copies and fused without copies of the Rodinia Benchmarks, we described similar CPU and GPU configurations and main memory, with the difference that when the system has a split address space we also split the addresses for both equally and the number of memory controllers: two for the baseline system and one for the unified memory system. The following table can help to have a better picture:

BASELINE		UNIFIED MEMORY	
CPU	**GPU**	**CPU**	**GPU**
4 Cores(x86 ISA) @ 2.5GHz	16 ShaderCores(PTX ISA) @700MHz	4 Cores(x86 ISA) @ 2.5GHz	16 ShaderCores(PTX ISA) @700MHz
L1 D: 64KB 2-way set associative	Warp Size: 32 B	L1 D: 64KB 2-way set associative	Warp Size: 32 B
L1 I: 32KB 4-way set-associative	L1 I/D: 64KB per SC	L1 I: 32KB 4-way set-associative	L1 I/D: 64KB per SC
L2 1MB 16-way set-associative (private)	L2: 1MB	L2 1MB 16-way set-associative (private)	L2: 1MB
2 GB with One Memory Controller (DDR3-1600 12.8GB/s)	2 GB with One Memory Controller (DDR3-1600 12.8GB/s)	4 GB with One Memory controller (DDR3-1600 12.8GB/s)	

Table 6-1: Baseline System and Unified Memory System

These two different memory systems configurations are meant to work both similarly in the processing cores to stress the two types of descriptions. We want to mainly observe the memory controller behavior running on each configuration the selected benchmarks from the Rodinia Suite. Is important to remember that the Rodinia benchmarks are implemented with CUDA and we have the two different types of configurations: first one with memory copies and two memory controllers for the baseline system and the second one without the memory copies and one memory controller for the unified memory system.

In order to validate our proposal in a first exploration approach for the hysteresis batch scheduling we focused our attention at the requests from a baseline system and the unified memory system memory controllers modeled at gem5-gpu Ruby. The baseline system has two different memory controllers because we have the two different memories “split” and the necessity of the CUDA memory copies is necessary. This separation at the physical level will help us to see separately the

request from CPU and GPU at determined phases of the execution with a “kernel status” level and relate who is doing the outstanding requests at each phase. At the unified memory system we just have one memory controller and it is not possible to check requests separated by each agent, but checking overlapping with the “kernel level” execution by this system vs. the baseline system we can infer the behavior and have an idea of the requestor and validate the idea of having hysteresis detection for batch scheduling detecting regions of outstanding requests by one or other agent.

In the hysteresis-batch scheduling, the bank level parallelism takes advantage of consecutive banks in the reordering of requests due to the consecutive row buffer hits that it will have when it is perfectly balanced and tuned. To replicate this behavior in order to obtain the upper bound numbers of the reduction at the stalls per requests, we look for the last bank issued, if it was consecutive to the new requests we are issuing, we eliminate the bank delay as it will be a row-hit access. With this approach we replicate the bank level parallelism of consecutive requests as in an ideal hysteresis batch scheduler (as we do not consider refresh or row-buffer timeouts).

This modification at the memory controller will be done at the unified memory system. The important numbers we need to check are the order of cycles reduced in total execution compared to the unified memory system.

6.1 Baseline Processor: Use of Memory Copies

The baseline system has two different memory controllers. One memory controller is going to be responsible for the memory requests from the CPU. The second memory controller is going to receive the requests from the GPU.

The memory address spaces are different from CPU to GPU and they need the use of memory copies to exchange data and instructions. The baseline system with split memories for CPU and GPU is going to help us to check the individual request from each processor (CPU or GPU) easier and to understand the level of stress of requests that each processor is giving to its own memory system.

Each memory controller reports a number of total memory requests received at the end of each execution by benchmark. We want to recall that we are using the Rodinia Benchmarks with the memory copies implemented.

We present the percentage of requests that each memory controller received in percentages to show which processor was giving the higher number of requests to its memory controller. (See Figure 10)

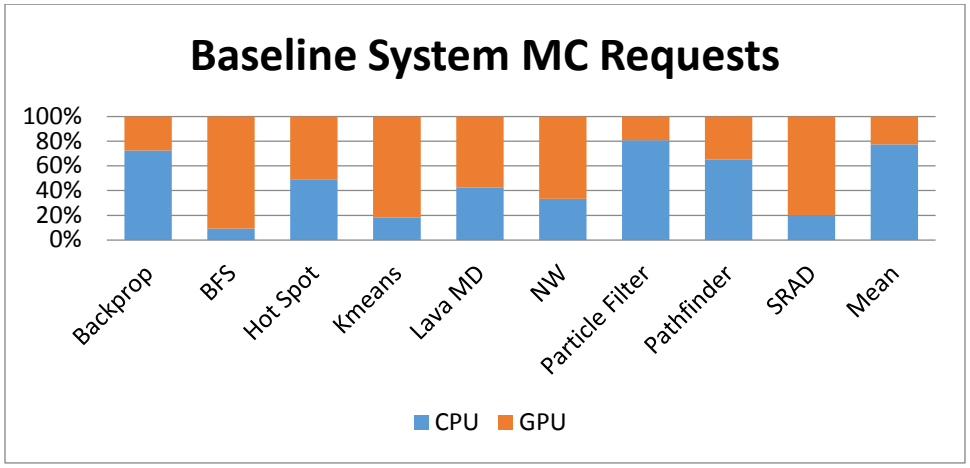


Figure 10: Percentage of the Number of Requests per Memory Controller at the Baseline System

There are benchmarks where the requests at the memory controllers are concentrated on GPU and others at the CPU. The mean gives us more concentration on the CPU memory controllers, but individually we can differentiate benchmarks where GPU memory controller has more requests like: BFS, Kmeans and SRAD. On the other hand, the CPU memory controller is getting more requests at: Backprop, Particle Filter and Pathfinder. Finally there are three benchmarks where the requests are similar between both memory controllers: Hot Spot, Lava MD and NW. These requests shown in Figure 10 can be also classified by type of request received at each memory controller: reads or writes. Figures 11 and 12 show the percentage of reads or writes of the total memory requests at a given memory controller: CPU or GPU.

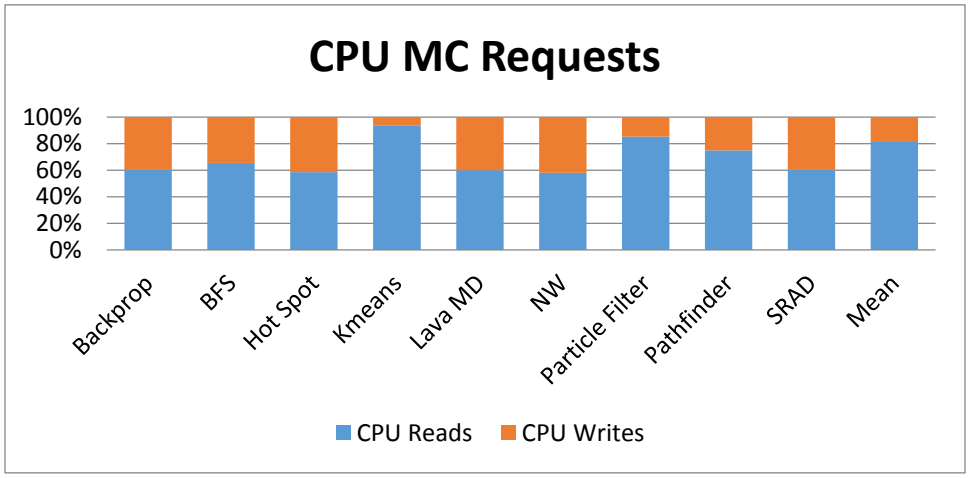


Figure 21: Read and Writes Requests at the CPU Memory Controller for the Baseline System

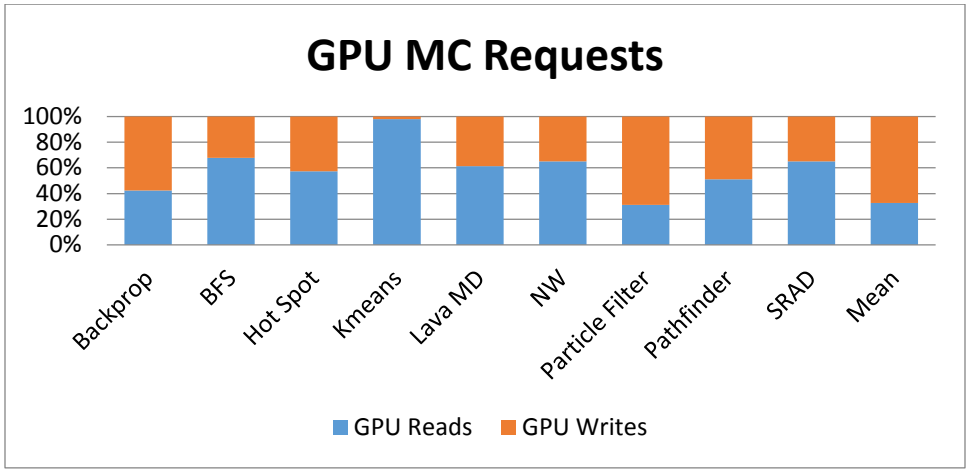


Figure 32: Read and Writes Requests at the GPU Memory Controller for the Baseline System

Memory requests have a number of stall cycles when the memory is busy. For the baseline system, the average number of stall cycles per request in each memory controller is shown at Figures 13 and 14.

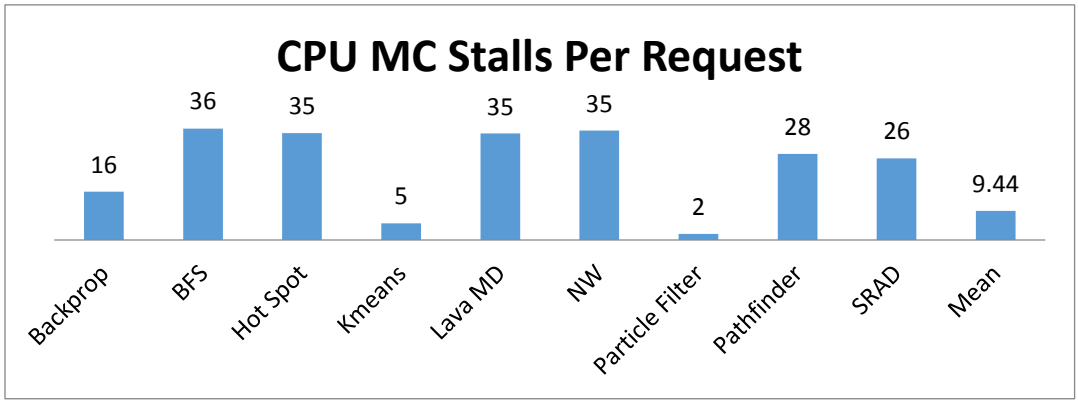


Figure 43: Stalls per request at the CPU Memory Controller of the Baseline System

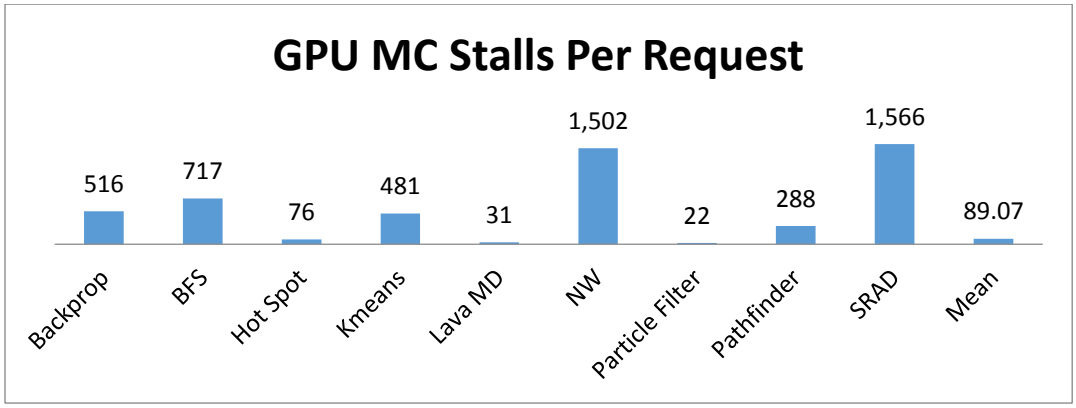


Figure 54: Stalls per request at the GPU Memory Controller of the Baseline System

6.2 Unified Memory Processor: No Memory Copies

The unified memory system has just one memory controller for the CPU and GPU, so it has to serve both processors requests. Figure 15 is classifies the number of requests at the memory controller by reads or writes. It is important to recall that we use the Rodinia Benchmarks without memory copies between CPU and GPU because at the Unified Memory System they share the same virtual address space.

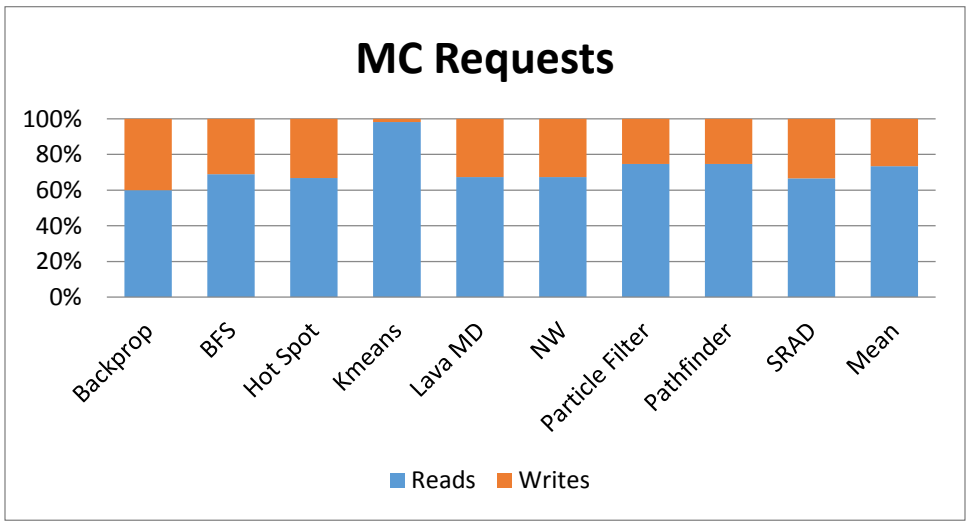


Figure 65: Read and Writes Requests at the Memory Controller of the Unified Memory System

The mean indicates more reads than writes. Figure 16 shows the number of stall cycles per requests:

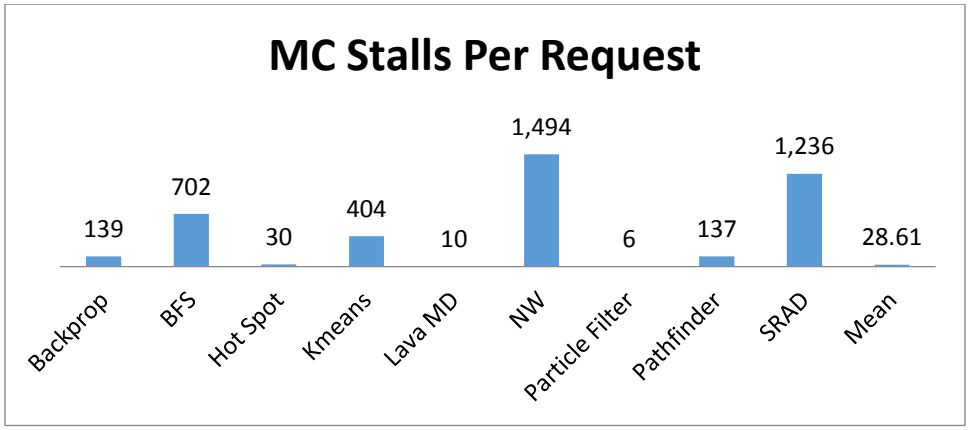
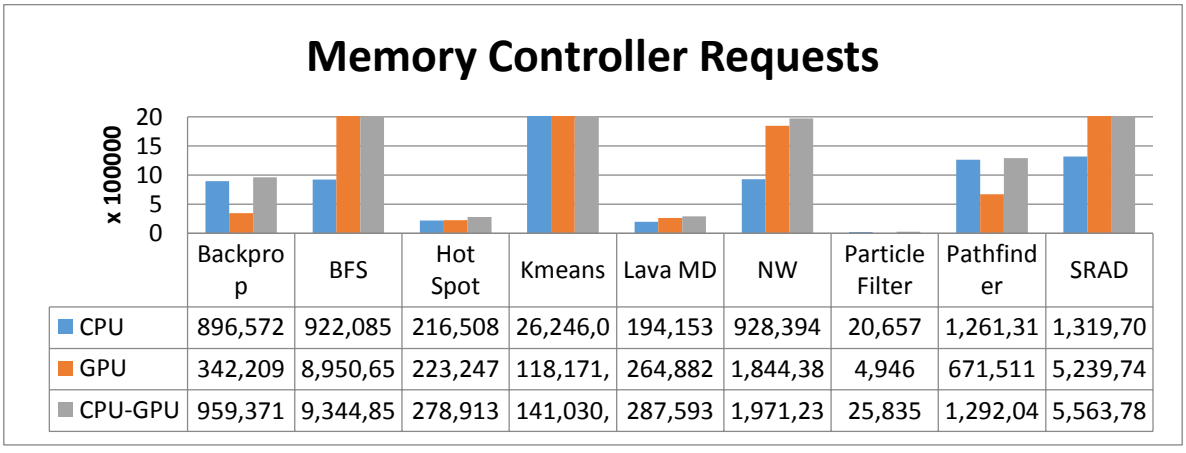


Figure 76: Stalls per request at the Memory Controller of the Unified Memory System

6.3 Analysis of the Hysteresis Detector

In the for a Hysteresis-Batch scheduler, one of the main characteristics is the ability to detect the different regions of activity with the memory controller requests. The requests will be received at arrival time and a two-bit saturation counter will be adding or subtracting depending on the ID of the requestor: CPU or GPU.

In order to validate the behavior of separated regions of execution for CPU and GPU we will compare the number of memory requests done by the Baseline System and the Unified Memory System. Figure 17 shows the general numbers of the requests received at the memory controllers:



	Backprop	BFS	Hot Spot	Kmeans	Lava MD	NW	Particle Filter	Pathfinder	SRAD
CPU	896,572	922,085	216,508	26,246,0	194,153	928,394	20,657	1,261,31	1,319,70
GPU	342,209	8,950,65	223,247	118,171,	264,882	1,844,38	4,946	671,511	5,239,74
CPU-GPU	959,371	9,344,85	278,913	141,030,	287,593	1,971,23	25,835	1,292,04	5,563,78

Figure 87: Total Number of MC Request CPU (Baseline System), GPU (Baseline System) and CPU+GPU (Unified Memory System)

From Figure 17 we can see a clearly relation between the memory controller that received more requests at the baseline system to the number of total request received at the unified memory system.

We are going to take the Hot Spot benchmark for a deeper analysis. The decision of taking this benchmark and no Lava MD is because at the Hot Spot numbers we have a closer similarity among the three memory controllers.

A closer look to the execution of the benchmark doing a kernel debugging execution by phases of the total execution we can see a mix behavior at the three different phases that the debugging shows at Figure 18.

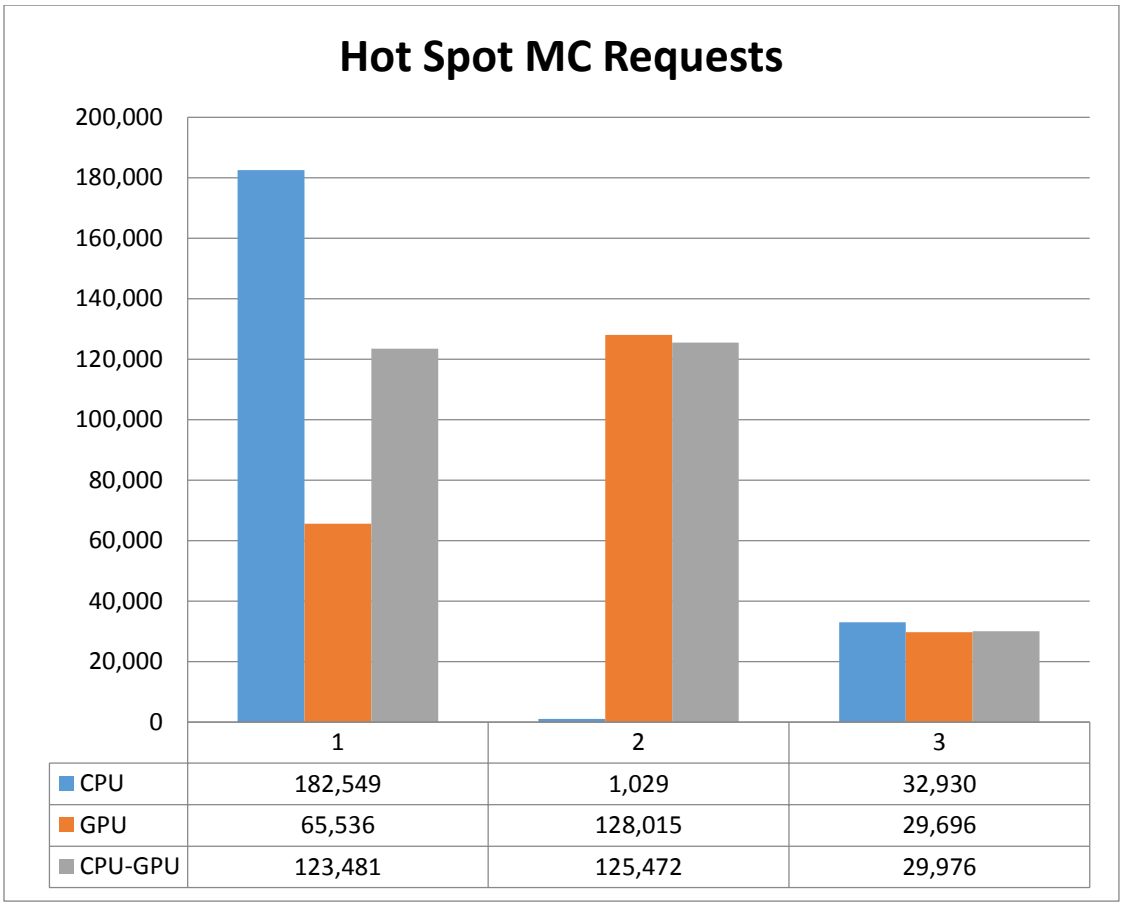
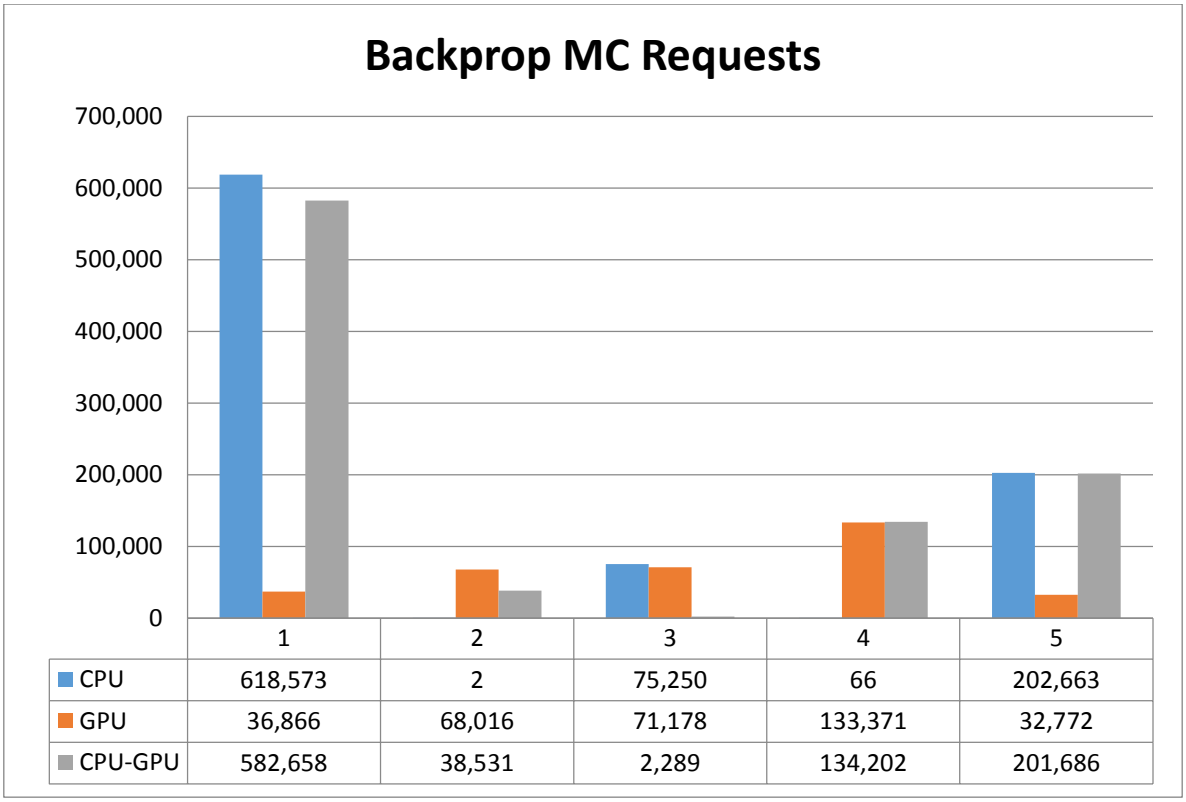


Figure 98: Hot Spot Memory Controller Requests by kernel debugging by phases. CPU (Baseline System), GPU (Baseline System) and CPU+GPU (Unified Memory System)

Starting with phase 2 of the execution kernel, clearly GPU requests dominate the baseline system and this is reflected at our unified memory system and our hysteresis detector would clearly reflect this for the scheduling purposes. Phase 1, Number of request are very well differentiated between CPU and GPU at the baseline system where the CPU requests almost triple the GPU ones. The

unified memory system numbers give the idea that the hysteresis detector will be detecting more memory intensity by the CPU and probably the GPU by some amount of times at the weak region. For the last phase we have the worst case due to the similarity of the number of request from CPU and GPU at the baseline system and the numbers shows also similar numbers. Here the hysteresis system would be in the weak range oscillating from CPU to GPU continually and neither CPU nor GPU threads will get the advantage of prioritizing their requests.

On the other hand Figure 19 shows the requests by kernel phases but from a very well balanced benchmark (i.e. Back Propagation). Phase number three from Back Propagation benchmark is the only phase where at the baseline system, memory request are very similar but the reduction of the number of request at the unified memory system is appreciated.



	1	2	3	4	5
CPU	618,573	2	75,250	66	202,663
GPU	36,866	68,016	71,178	133,371	32,772
CPU-GPU	582,658	38,531	2,289	134,202	201,686

Figure 19: Backprop Memory Controller Requests by kernel phases of execution CPU (Baseline System), GPU (Baseline System) and CPU+GPU (Unified Memory System)

Figures for LavaMD, Pathfinder and SRAD benchmarks with kernel debugging by phases can be seen in Annex C.

6.4 Hysteresis-Batch Scheduling at the Unified Memory System

Implementing the Hysteresis-Batch scheduling would give the opportunity to have a better reordering to the requests to main memory off-chip and with this a reduction in execution time cycles. The number of the cycles that a best case of Hysteresis-Batch scheduling can achieve is modeled with a reduction in latencies between the accesses to consecutive banks. This will give us the reduction execution time in number of cycles and check for the final impact on performance.

The execution time obtained with the unified memory system before and after the modification is shown at Figures 20 and 21:

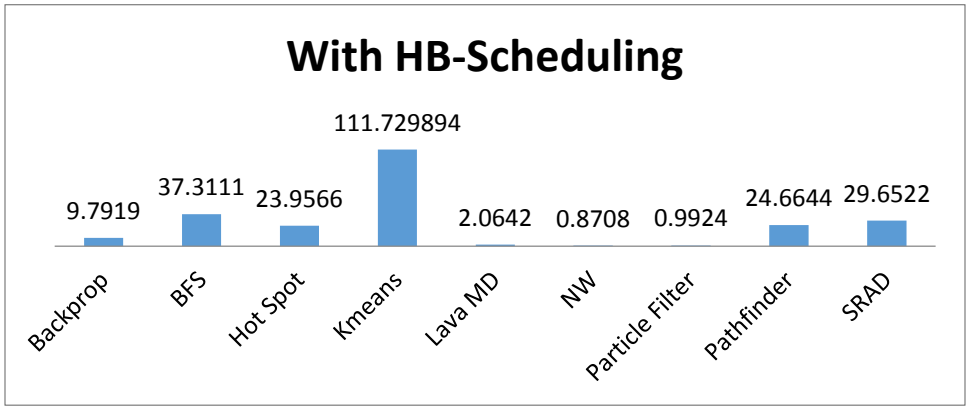


Figure 20: Execution time of the Unified Memory System with HB-Scheduling

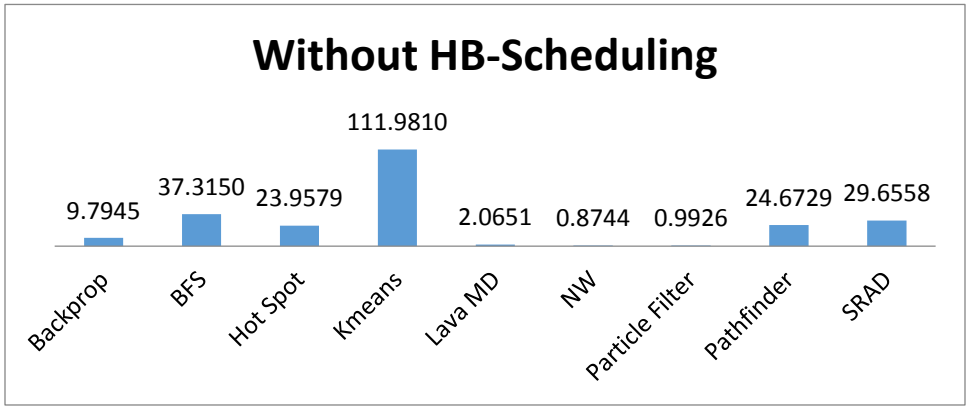


Figure 101: Execution time of the Unified Memory System with HB-Scheduling

The numbers of cycles that can be reduced on execution time are important in the order of 10^6. An important number taking into account that we just modify the scheduling reordering and millions of cycles can be saved.

7 Conclusions

A general algorithm for memory controller schedulers is far to exist. There are always more constraints than solutions when the complexity of On-chip systems is increasing. The analysis done at the beginning of this work compared two different memory systems: a baseline system with two memory controllers doing copies between them and a unified system with only one memory controller. An important reduction at the number of stall cycles was observed at the unified system. It is important to remember the existence of only one memory controller with similar characteristics to one of the memory controllers located at the baseline system. Using just this memory controller for the unified system resulted in less or slightly similar number of requests than the baseline system. With the motivation to increase the performance of the system we proposed a better treatment of these requests at the memory controller. The observed behaviors of requests at the memory controllers on the baseline system at different phases of the executed benchmarks encouraged us to propose a better treatment to these requests at the memory controller at the unified memory system. The Hysteresis-Batch scheduler proposed in this work for a Heterogeneous CPU-GPU processor took importance with the results of the analysis getting an important saving in execution cycles.

8 Future Work

New memory technologies with different and wide variety of characteristics are under research. Heterogeneous processors are called heterogeneous due to the diversity of processors. But heterogeneity can go to the memory system too in the near future. 3D-staking, for example, assembles layers of silicon connected with Through Silicon-Via (TSV) technology. This kind of connectivity directly on the chip, increase the bandwidth and decrease latency due bus connections. Those points are the Achilles heel of actual memory systems for high throughput processors. If the fabrication processes adapt fast, these memory technologies would be a reality sooner or later. The analysis of the integration of these memories to Heterogeneous processors must be explored to have a heterogeneous memory system where each memory technology will give its best characteristics for performance to the future computers.

9 Bibliography

[1] Moore, G. E. (2006). Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp. 114 ff.IEEE Solid-State Circuits Newsletter, 3(20), 33-35.

[2] Heinecke, A., Klemm, M., & Bungartz, H. J. (2012). From gpgpu to many-core: Nvidia fermi and intel many integrated core architecture. Computing in Science & Engineering, 14(2), 78-83.

[3] Wulf, W. A., & McKee, S. A. (1995). Hitting the memory wall: implications of the obvious. ACM SIGARCH computer architecture news, 23(1), 20-24.

[4] Lindholm, E., Kilgard, M. J., & Moreton, H. (2001, August). A user-programmable vertex engine. In Proceedings of the 28th annual conference on Computer graphics and interactive techniques (pp. 149-158). ACM.

[5] Damaraju, S., George, V., Jahagirdar, S., Khondker, T., Milstrey, R., Sarkar, S., ... & Subbiah, A. (2012, February). A 22nm ia multi-cpu and gpu system-on-chip. In Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International (pp. 56-57). IEEE.

[6] Daga, M., Aji, A. M., & Feng, W. C. (2011, July). On the efficacy of a fused cpu+ gpu processor (or apu) for parallel computing. In Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on (pp. 141-149). IEEE.

[7] Jacob, B., Ng, S., & Wang, D. (2010). Memory systems: cache, DRAM, disk. Morgan Kaufmann.

[8] Rixner, S., Dally, W. J., Kapasi, U. J., Mattson, P., & Owens, J. D. (2000). Memory access scheduling (Vol. 28, No. 2, pp. 128-138). ACM.

[9] Mutlu, O., & Moscibroda, T. (2007, December). Stall-time fair memory access scheduling for chip multiprocessors. In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (pp. 146-160). IEEE Computer Society.

[10] Mutlu, O., & Moscibroda, T. (2008, June). Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In ACM SIGARCH Computer Architecture News (Vol. 36, No. 3, pp. 63-74). IEEE Computer Society.

[11] Collange, S., Defour, D., & Parello, D. Barra, a Parallel Functional GPGPU Simulator.

[12] Zakharenko, V. (2012). *FusionSim: characterizing the performance benefits of fused CPU/GPU systems* (Doctoral dissertation, University of Toronto).

[13] Ubal, R., Jang, B., Mistry, P., Schaa, D., & Kaeli, D. (2012, September). Multi2Sim: a simulation framework for CPU-GPU computing. In Proceedings of the 21st international conference on Parallel architectures and compilation techniques (pp. 335-344). ACM.

[14] Power, J., Hestness, J., Orr, M., Hill, M., & Wood, D. (2014). gem5-gpu: A heterogeneous cpu-gpu simulator.

[15] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S. H., & Skadron, K. (2009, October). Rodinia: A benchmark suite for heterogeneous computing. In Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on (pp. 44-54). IEEE.

[16] Hestness, J., Keckler, S. W., & Wood, D. (2014, October). A comparative analysis of microarchitecture effects on CPU and GPU memory system behavior. In Workload Characterization (IISWC), 2014 IEEE International Symposium on (pp. 150-160). IEEE.

[17] Power, J., Hill, M. D., & Wood, D. (2014, February). Supporting x86-64 address translation for 100s of GPU lanes. In High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on (pp. 568-578). IEEE.

[18] Subramanian, G., Thakker, U., Haria, S., Shukla, R., & Lin, H. (2015). Exploring CPU-GPU Coherence.

[19] Patterson, D. A., & Hennessy, J. L. (2013). Computer organization and design: the hardware/software interface. Newnes.

[20] Hennessy, J. L., & Patterson, D. A. (2011). Computer architecture: a quantitative approach. Elsevier.

[21] Li, D., & Wu, J. (2012). Energy-aware scheduling on multiprocessor platforms. Springer Science & Business Media.

[22] Uchiyama, K., Arakawa, F., Kasahara, H., Nojiri, T., Noda, H., Tawara, Y., ... & Shikano, H. (2012). Heterogeneous Multicore Processor Technologies for Embedded Systems. Springer.

[23] Kim, H., Vuduc, R., Baghsorkhi, S., & Choi, J. (2012). Performance analysis and tuning for general purpose graphics processing units (GPGPU). Morgan & Claypool Publishers.

Annex-A

A.1 CUDA and Heterogeneous CPU – GPU Computing

Compute Unified Device Architecture (CUDA) is an extension for languages like C and C++ to program GPU devices and do general purpose computations. Programmers write a normal serial program that is executed at the host, the CPU.

This program calls parallel “kernels”, this is a function designed to be executed by many threads. Until now this seems pretty normal programming without any expertise needed, but the way the programmer organizes these threads in the kernel(s) is where the real deal comes into the play.

When invoking a kernel, we organize threads into a hierarchy of thread blocks and a grid of thread blocks. A thread block is a set of concurrent threads that can cooperate among themselves though barrier synchronization and through shared access to a memory space private to the block. A grid is a set of thread blocks that may each be executed independently and thus may execute in parallel.

The programmer specifies the number of threads per block and the number of blocks comprising the grid. Each thread is given a unique “thread ID” number <<threadIdx>> within its thread block, numbered 0, 1, 2, ... , <<blockDim–1>>, and each thread block is given a unique “block ID” number <<blockIdx>> within its grid. CUDA supports thread blocks containing up to 512 threads. For convenience, thread blocks and grids may have 1, 2, or 3 dimensions, accessed via .x, .y, and .z index fields:

kernel<<<dimGrid, dimBlock>>>(... parameter list ...);

The programmer can use a convenient degree of parallelism for each kernel, rather than having to design all phases of the computation to use the same number of threads. Figure XXXX shows an example of a CUDA-like code sequence. It first instantiates “kernel F” on a 2D grid of 3 × 2 blocks where each 2D thread block consists of 5 × 3 threads. It then instantiates “kernel” on a 1D grid of four 1D thread blocks with six threads each. Because kernel G depends on the results of kernel F, they are separated by an inter-kernel synchronization barrier.

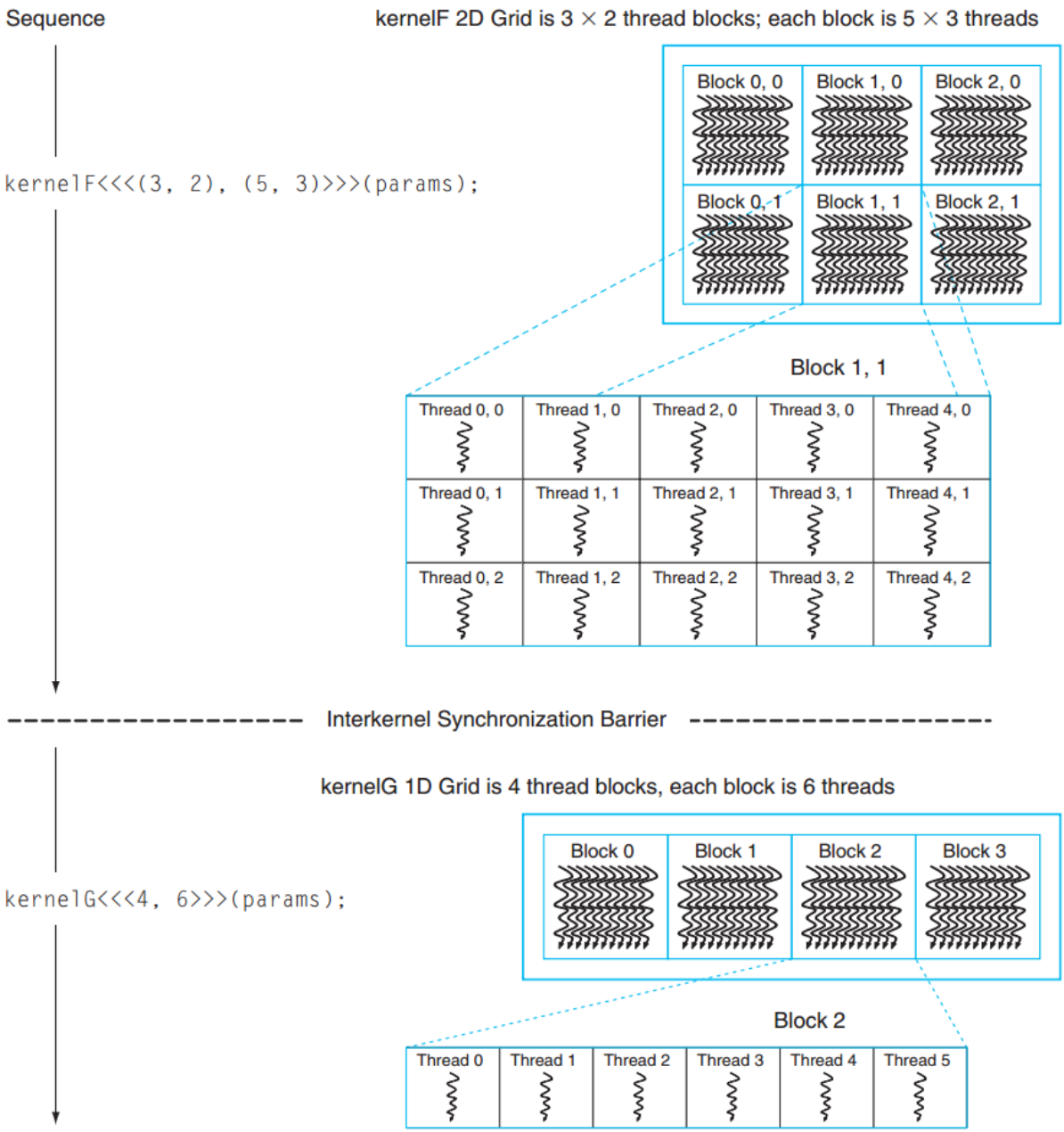


Figure 22: Sequence of kernel F instantiated, an inter kernel synchronization barrier, followed by kernel G [19]

Let's not forget that to support a heterogeneous system architecture combining a CPU and a GPU, each with its own memory system; CUDA programs must copy data and results between host memory and device memory. The overhead of CPU – GPU interaction and data transfers is minimized by using DMA block transfer engines and fast interconnects. Compute-intensive problems large enough to need a GPU performance boost amortize the overhead better than small problems.

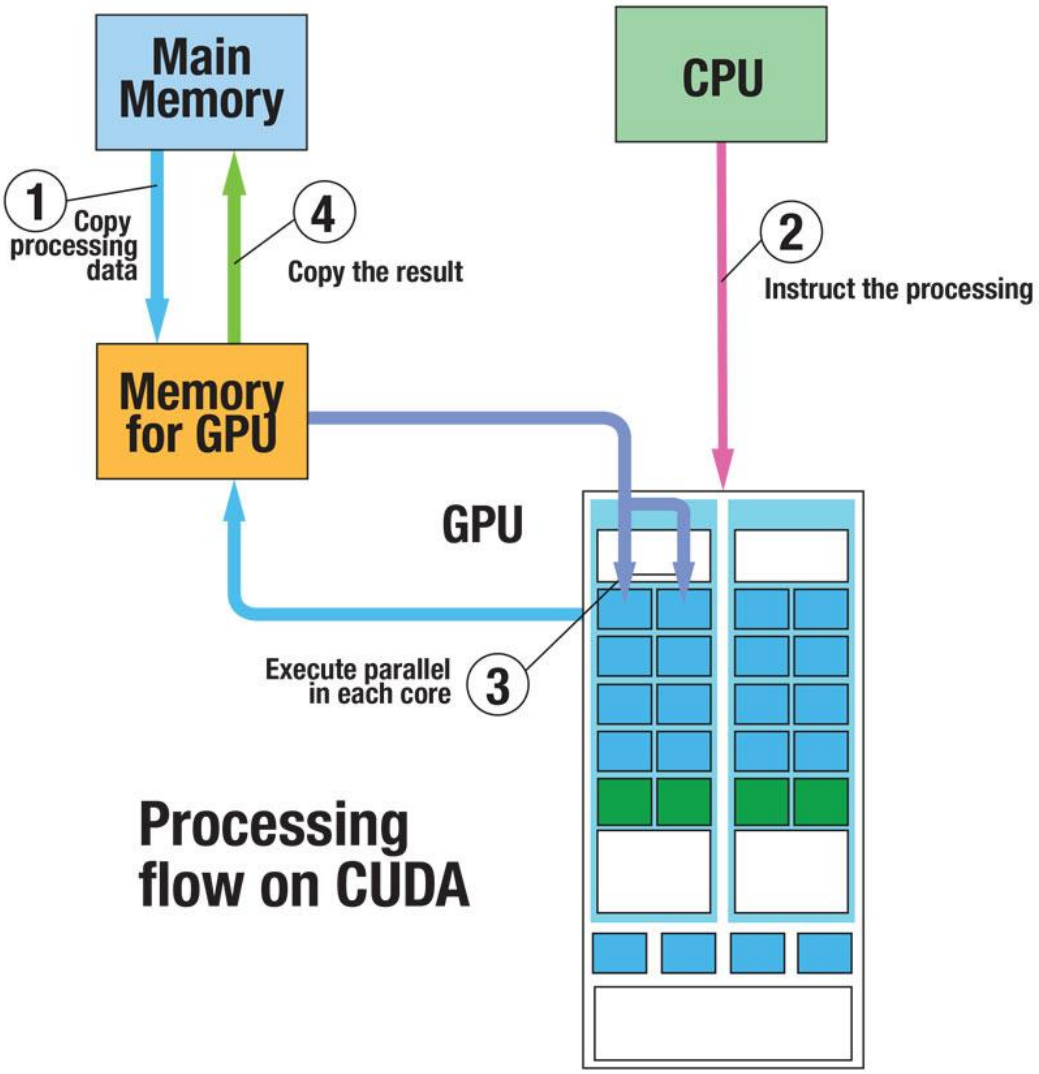
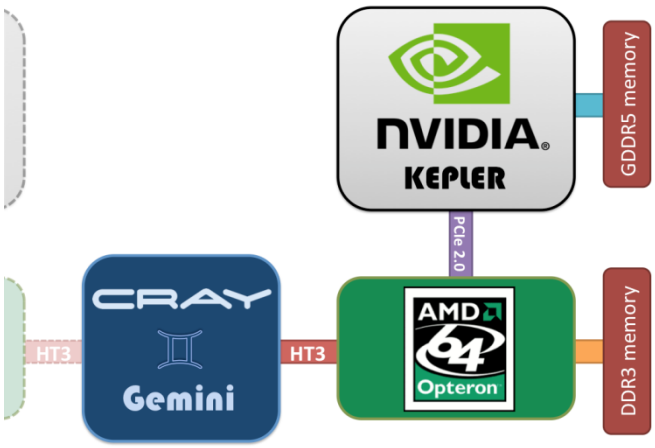


Figure 113: Processing Flow on CUDA (Source: RTC Magazine)

A.2 Titan Supercomputer Heterogeneous Node

The example of Titan as a Heterogeneous CPU – GPU System will bring us the idea of how the General Purpose (GP) GPU is use in a system, mainly the memory configuration on each side of the CPU and GPU.

Titan has 18, 688 nodes. Each node has one CPU and one GPGPU with following characteristics:



Titan Compute Nodes (Cray XK7)			
Accelerator (GPU)	NVIDIA Tesla K20x (2688Cores)	732 MHz	6 GB (GDDR5)
Node (CPU)	AMD Opteron 6274 (16 cores)	2.2 GHz	32 GB (DDR3)

Figure 12 Titan Heterogeneous Computing Node (Source: OLCF)

Figure 24 shows the interconnections between the different chips. The CPU, called the node, is connected to an interconnection ASIC proprietary of Cray Inc. and in the other side is a connection to the accelerator, the GPGPU. In this work, the interconnection of nodes in a system is not our main interest so we will discard the explanation for the “Gemini” ASIC and the connection to the other nodes. What we want to show and what is relevant for this case of study is the interaction of the node and the accelerator.

If we take a closer look to Figure 25, there is a direct connection between the CPU and the GPU with a PCIe 2.0 in purple. But each of them, CPU and GPU, has its own memory device; different in capacity and technology.

This configuration demands explicit copies of data and instructions from the host, node or CPU to the device, accelerator or GPU. Copies of these, data and instructions, are done through the PCIe that is connection them. The compute acceleration that the GPU is giving works well only for large offload workloads due to the slow data transfer between the CPU and GPU.

On top of that, another disadvantage is the expert level of programming necessary to take real advantage of the GPU compute. Many programming languages are available such as CUDA and OpenCL. CUDA is from NVIDIA, the enterprise that develops the GPUs Titan has and we will have a brief overview of the processing flow. OpenCL is framework for heterogeneous platforms in general, not just with GPUs, also DSPs and FPGAs where a system has a number of compute devices (other CPUs or accelerators) attached to a host processor (CPU).

A.3 AMD and Heterogeneous Uniform Memory Access (hUMA)

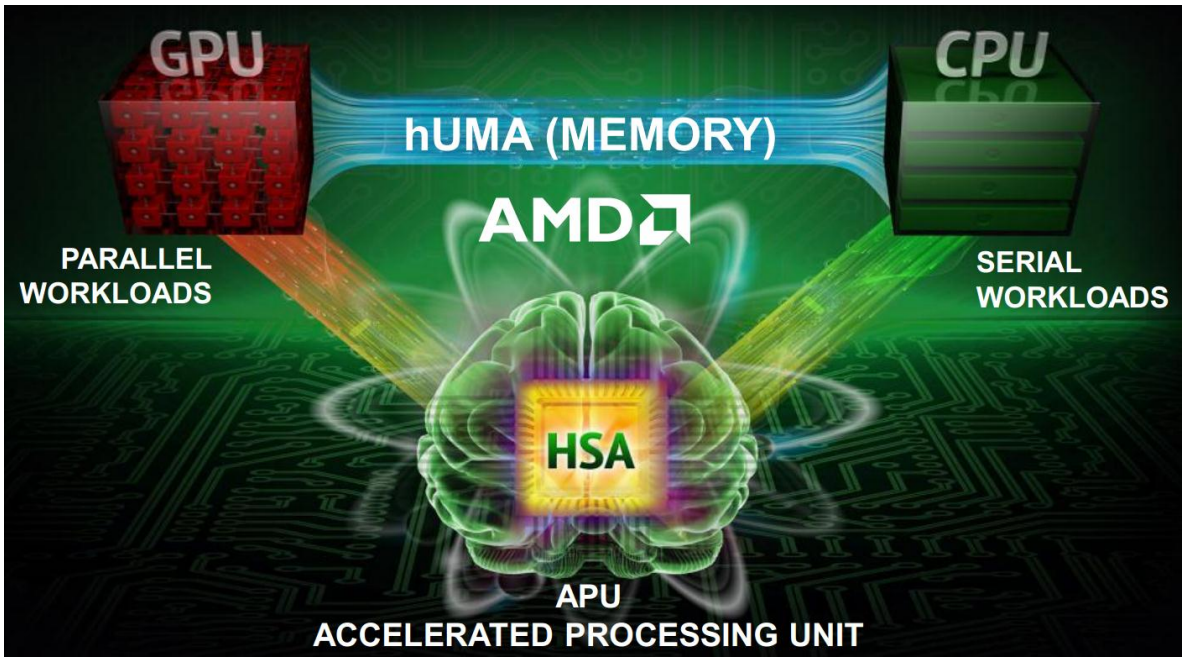


Figure 25: hUMA from AMD (Source: AMD)

Recalling the evolution from CPUs on multicore, just before the introduction of GPU on chip, there was a Unified Memory Access (UMA) so all the cores in this system share a single memory address space. Later with the integration of the GPU at these APUs, a Non-Uniform Memory Access (NUMA) was created. It required data to be managed across different address spaces: GPU Memory and CPU Memory, adding complexity due frequent copies, synchronization and address translations.

With hUMA features from HAS Foundation, CPU and GPU processes can dynamically allocate memory from the entire memory space. Updates made by one of the processing element, CPU or GPU, will be seen by all other processing elements due to a bi-directional coherent. Also GPU can take page faults and is no longer restricted to page locked memory.

For data sharing, the CPU can simply pass a pointer to entire data structure since the GPU can now follow embedded links, then GPU completes the computation and CPU can read the result directly with no copying needed. This highly speeds the GPU access to the system memory and frees the computation on GPUs from the copying overheads.

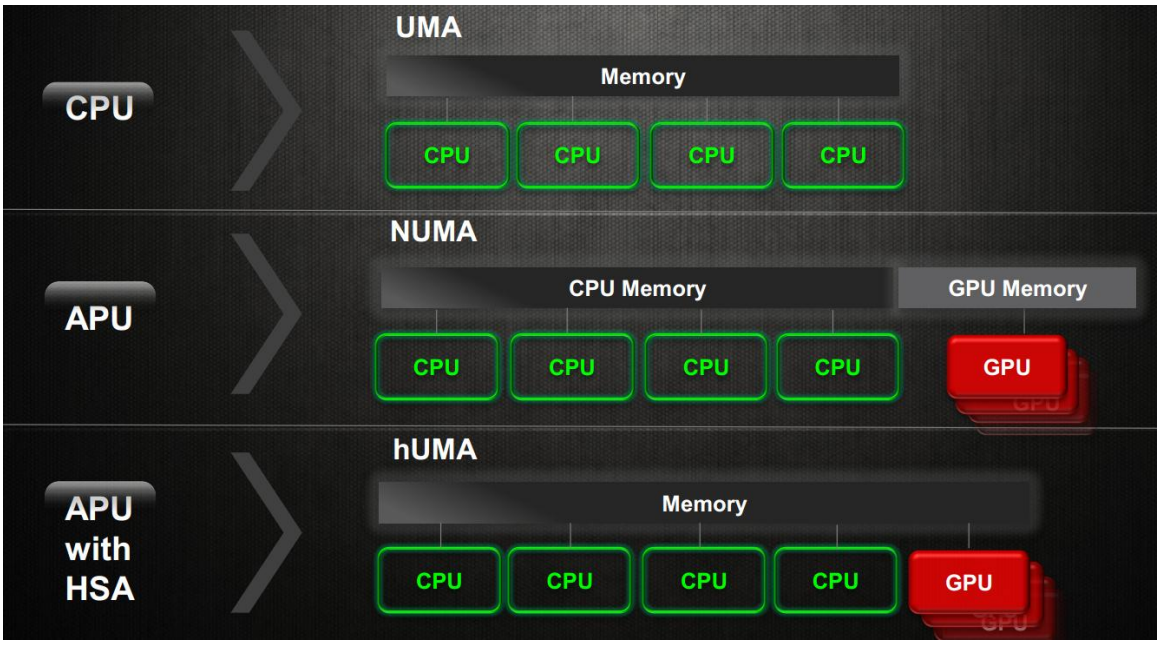


Figure 26: Comparison between CPU and APU's (Source: AMD)

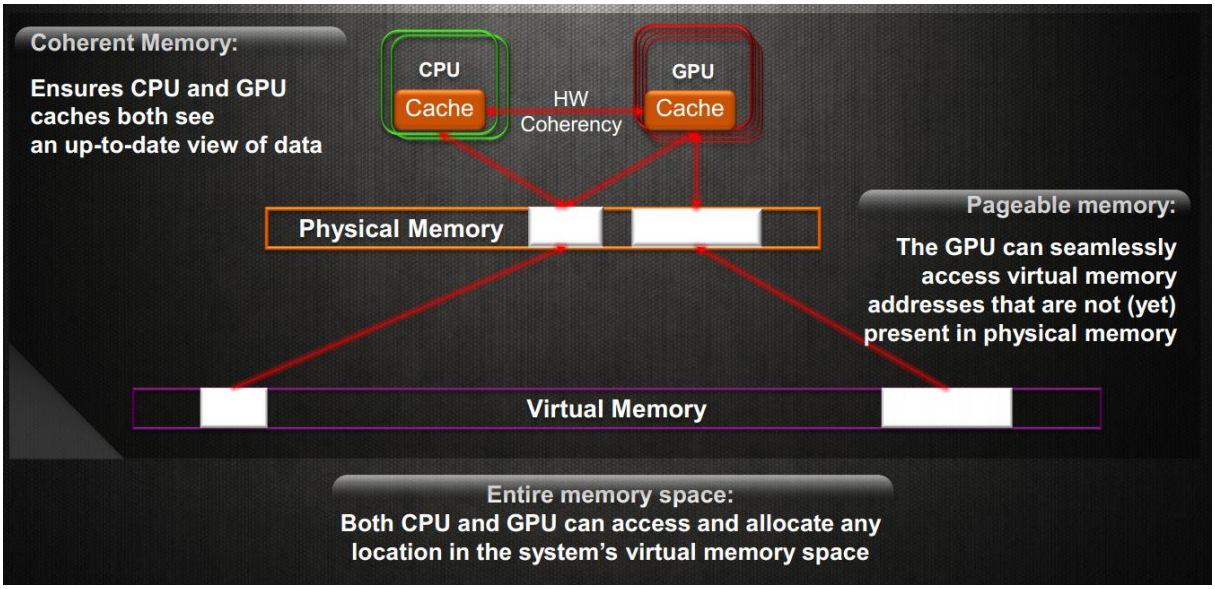


Figure 137: hUMA Key features (Source: AMD)

Annex-B

B.1 RUBY & SLICC

It is a flexible infrastructure capable of accurately simulating a wide variety of memory systems. Supports a domain specific language called SLICC (Specification Language for Implementing Cache Coherence) where one can define many different types of cache coherence protocols.

Essentially the SLICC defines the cache, memory, and DMA controllers as individual per memory block state machines that together form the overall protocol. By defining the controller logic in a higher level language, SLICC allows different protocols to incorporate the same underlining state transition mechanisms with minimal programmer effort.

It models inclusive/exclusive cache hierarchies with various replacement policies, coherence protocol implementations, interconnection networks, DMA and Memory controllers, various sequencers that initiate memory requests and handle responses. The models are modular, flexible and highly configurable. Three key aspects of these models are:

1. Separation of concerns = e.g. the coherence protocol specifications are separate from the replacement polices and cache index mapping; the network topology is specified separately from the implementation.

2. Rich Configurability = Almost any aspect affecting the memory hierarchy functionality and timing can be controlled.

3. Rapid prototyping = A high-level specification language, SLICC, is used to specify functionality of various controllers.

Cache Topology

The topology of the cache hierarchy in gem5-gpu is implemented with Ruby. Gem5-gpu should be able to use any topology that Ruby supports (mesh, pt2pt, crossbar, etc.). However, these topologies were created with homogeneous CPU cores in mind. Using these topologies may result in strange behavior. For instance, you may have GPU and CPU cores scattered throughout the mesh randomly. Pt2Pt and crossbar should work fine since they are completely flat topologies.

For the VI_Hammer coherence protocol, cluster topology is hardcoded. The cluster topology is a hierarchy of crossbars meant to model clusters of different kinds of cores. All GPU cores (and their

L1s) are connected to 1 crossbar, which is also connected to the GPU L2. All CPU cores (and their private L1s and L2s) are connected to another crossbar. These two crossbars are connected to another crossbar, which is also connected to the Directory and Memory Controller.

Ruby Memory Controller:

This module models a single channel, connected to any number of DIMMs with any number of ranks of DRAMs each. Each memory request is placed in a queue associated with a specific memory bank. This queue is of finite size; if the queue is full the request will back up in an (infinite) common queue and will effectively throttle the whole system. This sort of behavior is intended to be closer to real system behavior than if we had an infinite queue on each bank. The head item on a bank queue is issued when all of the following are true:

1. The bank is available
2. The address path to the DIMM is available
3. The data path to or from the DIMM is available

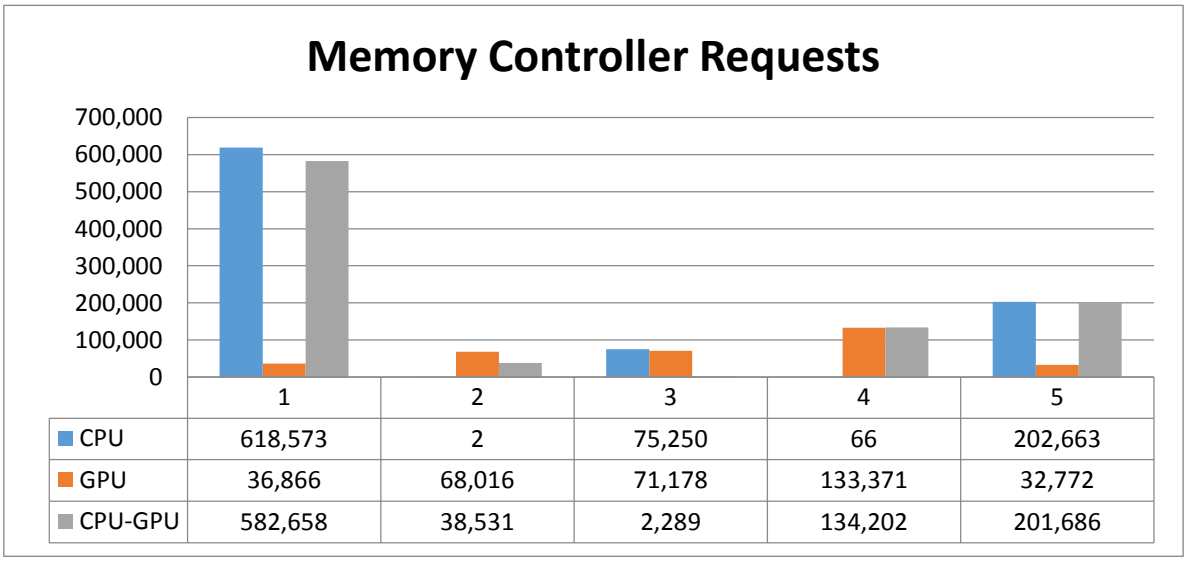
Note that we are not concerned about fixed offsets in time. The bank will not be used at the same moment as the address path, but since there is no queue in the DIMM or the DRAM it will be used at a constant number of cycles later, so it is treated as if it is used at the same time. They are assuming closed bank policy; that is, we automatically close each bank after a single read or write.

The only non-trivial scheduling problem is the data wires. A write will use the wires earlier in the operation than a read will; typically one cycle earlier as seen at the DRAM, but earlier by a worst-case round-trip wire delay when seen at the memory controller. So, while reads from one rank can be scheduled back-to-back every two cycles, and writes (to any rank) scheduled every two cycles, when a read is followed by a write we need to insert a bubble. Furthermore, consecutive reads from two different ranks may need to insert a bubble due to skew between when one DRAM stops driving the wires and when the other one starts (These bubbles are parameters). This means that when some number of reads and writes are at the heads of their queues, reads could starve writes, and/or reads to the same rank could starve out other requests, since the others would never see the data bus ready. For this reason, we have implemented an anti-starvation feature. A group of requests is marked "old", and a counter is incremented each cycle as long as any request from that batch has not issued. If the counter reaches twice the bank busy time, we hold off any newer requests until all of the "old" requests have issued.

Annex C:

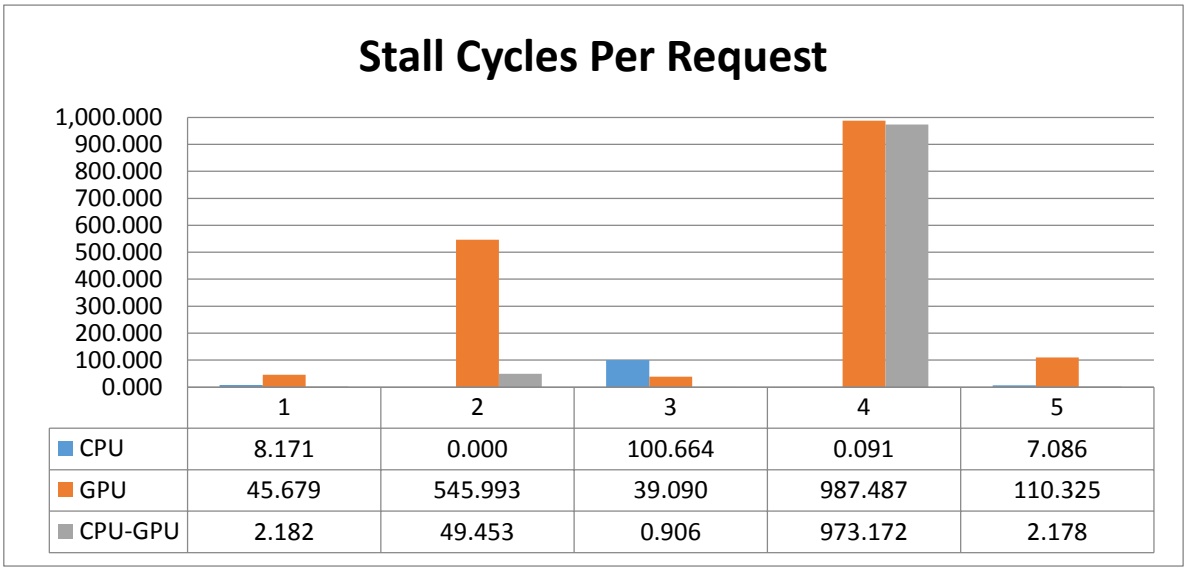
C.1 Benchmarks Results at Kernel Phase Execution

Backprop



	1	2	3	4	5
CPU	618,573	2	75,250	66	202,663
GPU	36,866	68,016	71,178	133,371	32,772
CPU-GPU	582,658	38,531	2,289	134,202	201,686

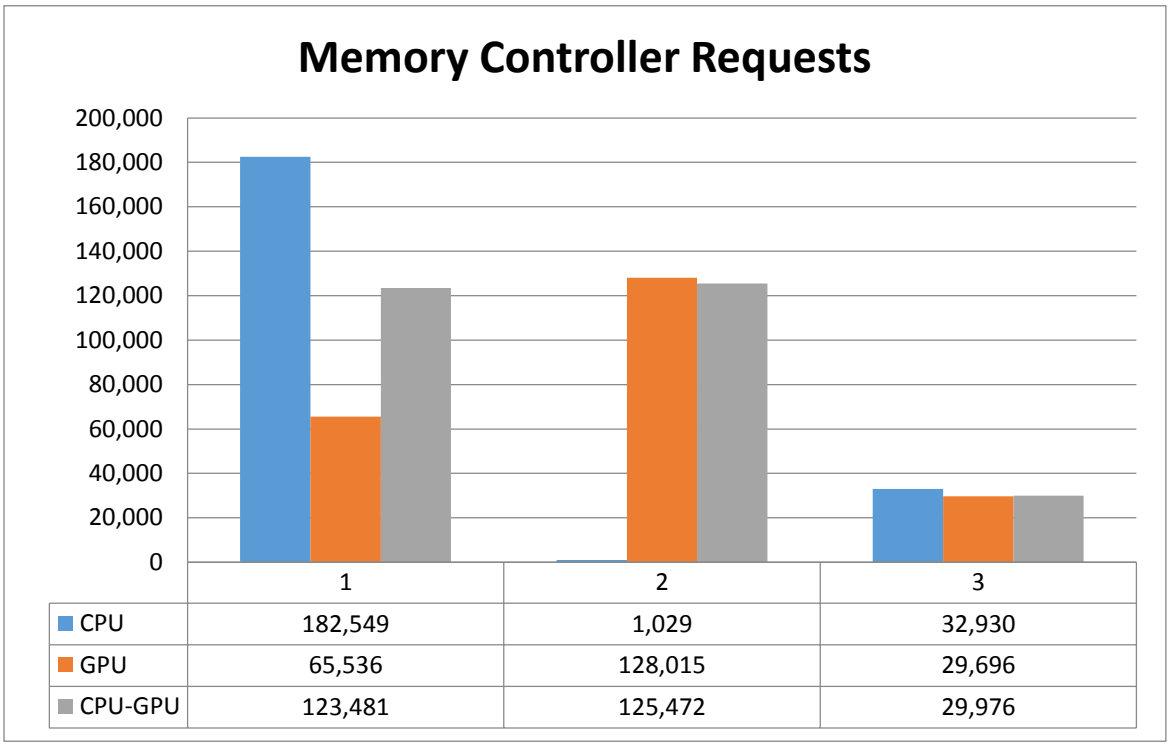
Figure 148: Backprop-Memory Controller Requests by kernel execution phase



	1	2	3	4	5
CPU	8.171	0.000	100.664	0.091	7.086
GPU	45.679	545.993	39.090	987.487	110.325
CPU-GPU	2.182	49.453	0.906	973.172	2.178

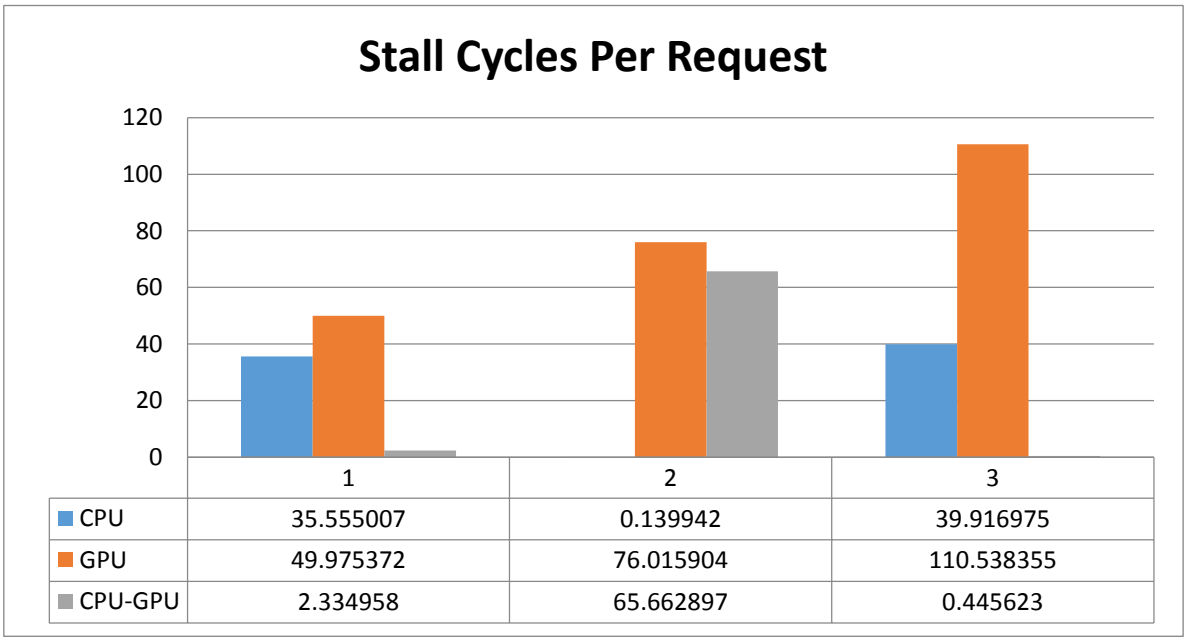
Figure 29: Backprop-Stall Cycles Per Request by kernel execution phase

Hot Spot



	1	2	3
CPU	182,549	1,029	32,930
GPU	65,536	128,015	29,696
CPU-GPU	123,481	125,472	29,976

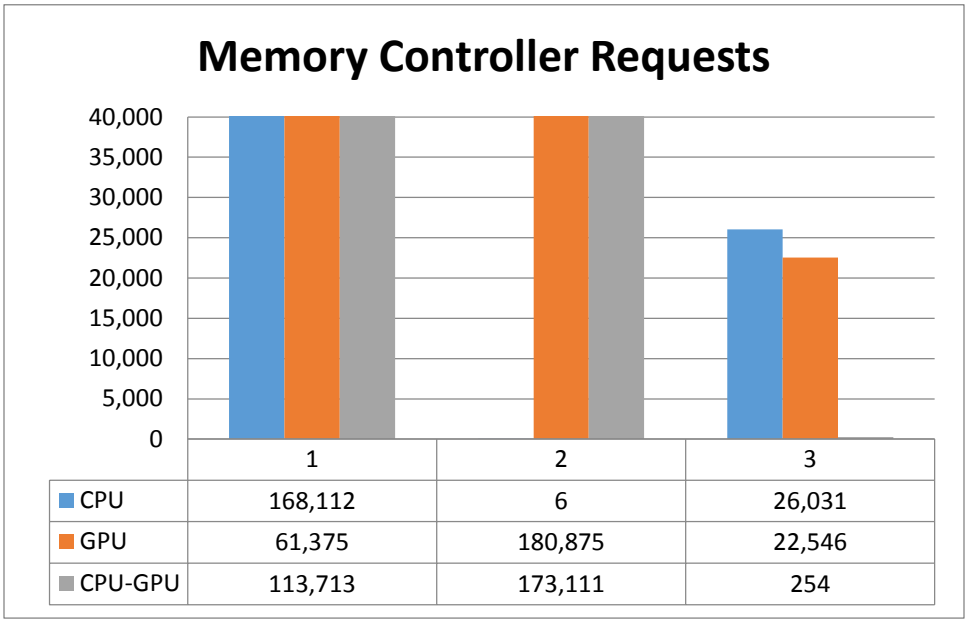
Figure 150: Hot Spot-Memory Controller Requests by kernel execution phase



	1	2	3
CPU	35.555007	0.139942	39.916975
GPU	49.975372	76.015904	110.538355
CPU-GPU	2.334958	65.662897	0.445623

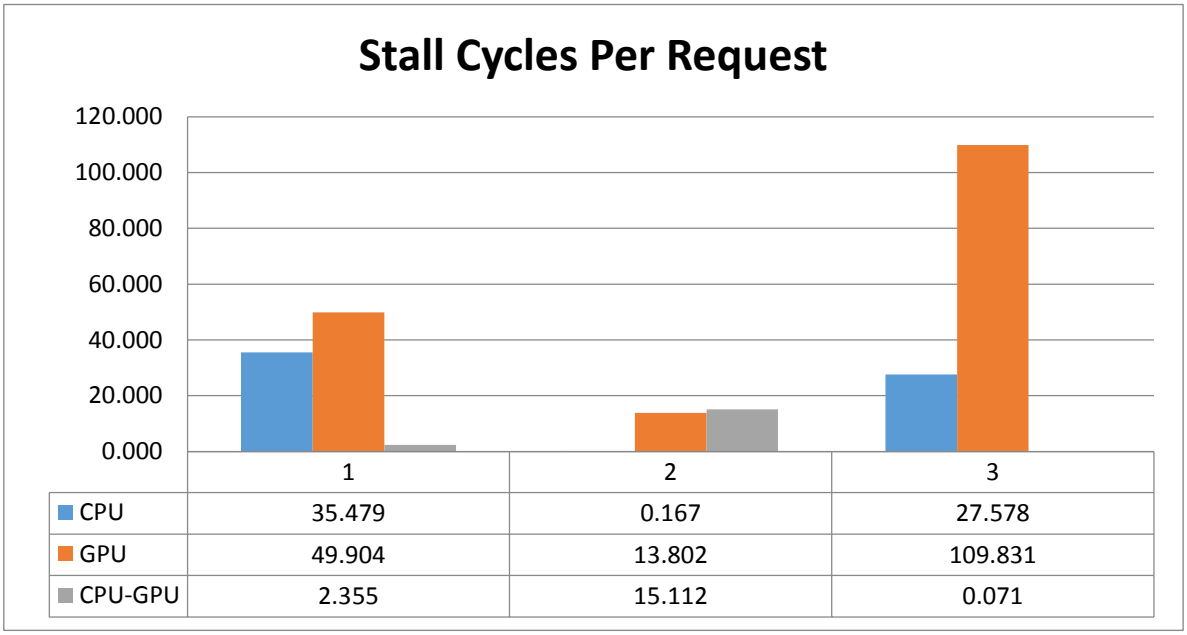
Figure 161: Hot Spot-Stall Cycles Per Request by kernel execution phase

Lava MD



	1	2	3
CPU	168,112	6	26,031
GPU	61,375	180,875	22,546
CPU-GPU	113,713	173,111	254

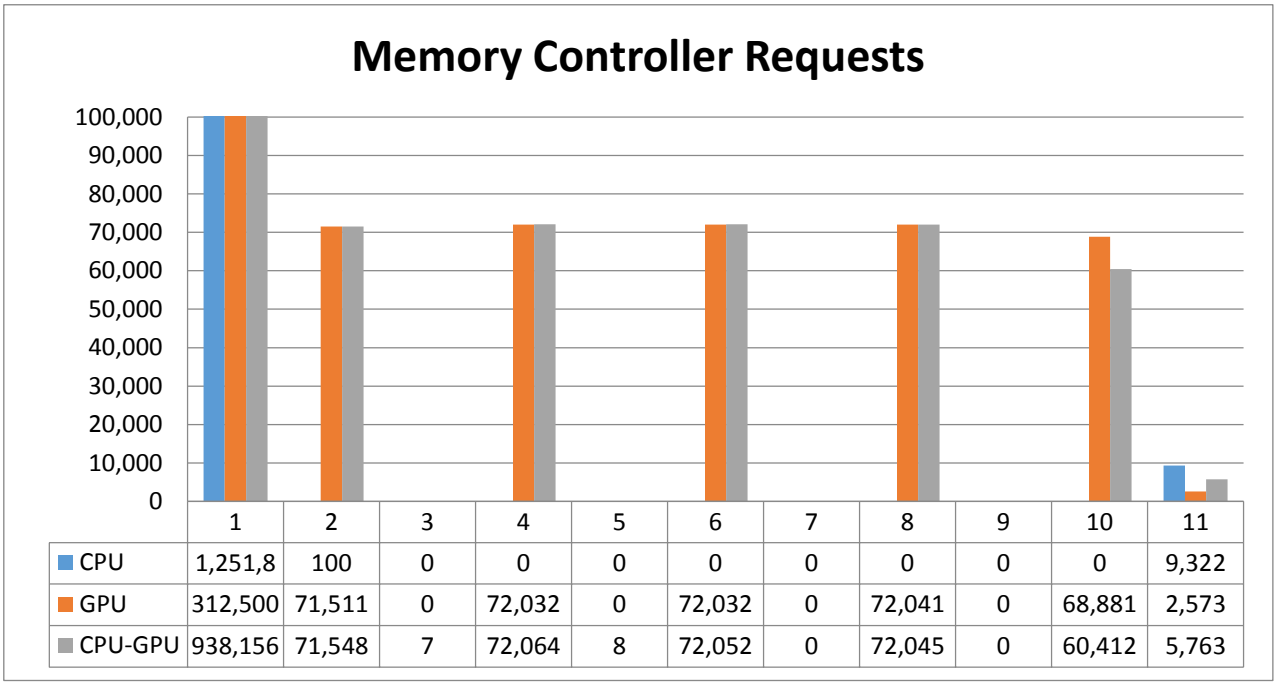
Figure 172: LavaMD-Memory Controller Requests by kernel execution phase



	1	2	3
CPU	35.479	0.167	27.578
GPU	49.904	13.802	109.831
CPU-GPU	2.355	15.112	0.071

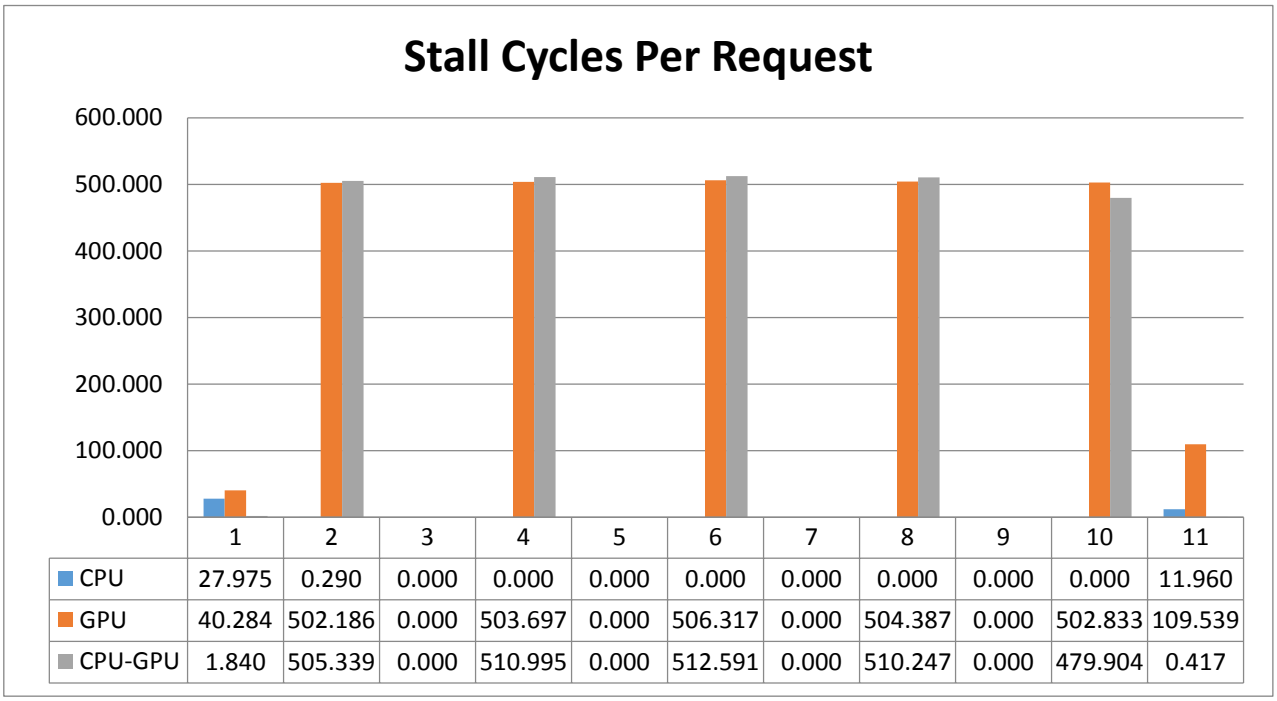
Figure 183: LavaMD-Stall Cycles Per Request by kernel execution phase

Pathfinder:



	1	2	3	4	5	6	7	8	9	10	11
CPU	1,251,8	100	0	0	0	0	0	0	0	0	9,322
GPU	312,500	71,511	0	72,032	0	72,032	0	72,041	0	68,881	2,573
CPU-GPU	938,156	71,548	7	72,064	8	72,052	0	72,045	0	60,412	5,763

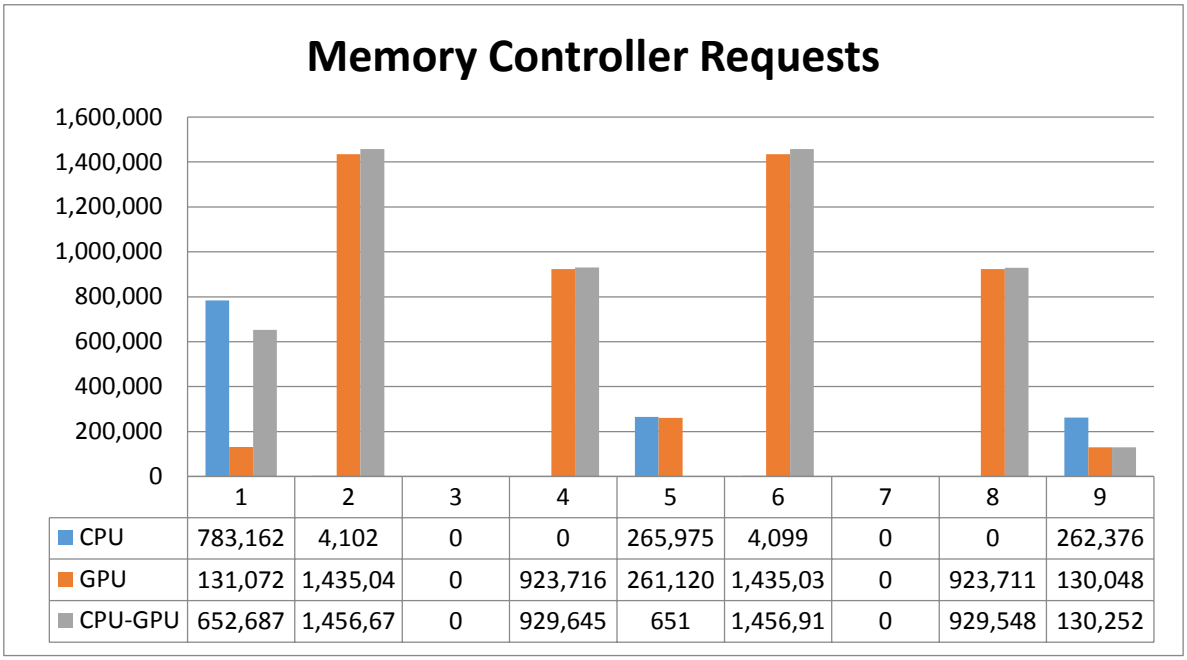
Figure 194: Pathfinder-Memory Controller Requests by kernel execution phase



	1	2	3	4	5	6	7	8	9	10	11
CPU	27.975	0.290	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	11.960
GPU	40.284	502.186	0.000	503.697	0.000	506.317	0.000	504.387	0.000	502.833	109.539
CPU-GPU	1.840	505.339	0.000	510.995	0.000	512.591	0.000	510.247	0.000	479.904	0.417

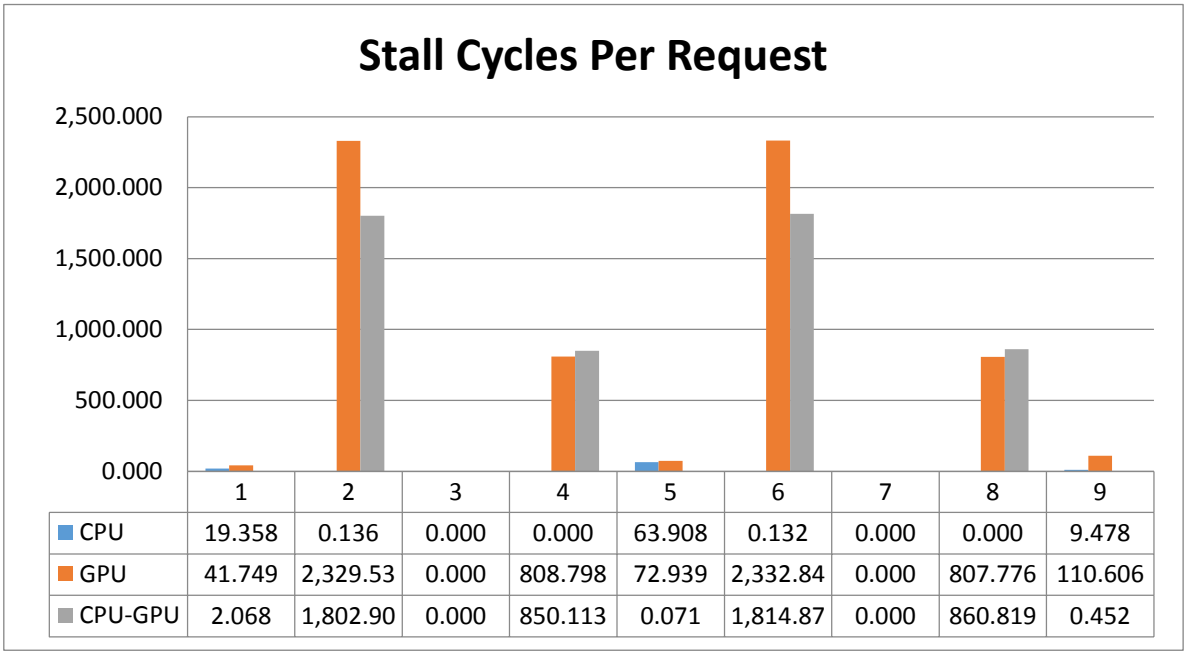
Figure 205: Pathfinder-Stall Cycles Per Request by kernel execution phase

SRAD



	1	2	3	4	5	6	7	8	9
CPU	783,162	4,102	0	0	265,975	4,099	0	0	262,376
GPU	131,072	1,435,04	0	923,716	261,120	1,435,03	0	923,711	130,048
CPU-GPU	652,687	1,456,67	0	929,645	651	1,456,91	0	929,548	130,252

Figure 216: SRAD-Memory Controller Requests by kernel execution phase



	1	2	3	4	5	6	7	8	9
CPU	19.358	0.136	0.000	0.000	63.908	0.132	0.000	0.000	9.478
GPU	41.749	2,329.53	0.000	808.798	72.939	2,332.84	0.000	807.776	110.606
CPU-GPU	2.068	1,802.90	0.000	850.113	0.071	1,814.87	0.000	860.819	0.452

Figure 227: SRAD-Stall Cycles Per Request by kernel execution phase