# Improving Memory Hierarchy Performance in Heterogeneous System Architecture (HSA)

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
**Master of Engineering**
IN
**Faculty of Engineering**

BY

**Patel Arth Kausheybhai**

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

June, 2015

# Declaration of Originality

I, **Patel Arth Kausheybhai**, with SR No. **04-04-00-10-41-13-1-10228** hereby declare that the material presented in the thesis titled

**Improving Memory Hierarchy Performance in Heterogeneous System Architecture (HSA)**

represents original work carried out by me in the **Deparment of Computer Science and Automation** at **Indian Institute of Science** during the years **2013-2015**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discusions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Prof. R. Govindarajan Advisor Signature

**© Arth Patel**
**June, 2015**
**All rights reserved**

DEDICATED TO

*My Family and Friends*

*for their continuous support and encouragement.*

*Signature of the Author*: .............................................

Patel Arth Kausheybhai
Dept. of Computer Science and Automation
Indian Institute of Science, Bangalore

*Signature of the Thesis Supervisor*: .............................................

Prof. R. Govindarajan
Professor
Dept. of Computer Science and Automation
Indian Institute of Science, Bangalore

# Acknowledgements

I am extremely thankful to Prof. R. Govindarajan for sharing his expertise, and sincere and valuable guidance. He has been enlightening, motivating and encouraging force throughout this project. His constant feedback and advice has been instrumental.

I would also like to thank and appreciate my fellow lab mates for the insightful discussions and valuable support.

I thank all Department faculty and staff members for their help and assistance.

I would like to thank all my friends for their support and positive influence throughout my journey at IISc.

Finally, the acknowledgement would be incomplete without thanking my parents for their inspiration, love, support and encouragement.

# Abstract

The increasing use of General-Purpose computing on Graphics Processing Units (GPGPU) has attracted the attention of researchers to several research issues around current architecture with discrete Graphics Processing Units (GPU). In particular the problem of data transfer overhead and memory consistency between CPU and GPU. Newer Heterogeneous System Architecture (HSA) have been proposed to overcome the issues of the traditional architecture for GPGPU computing. However, the switch to Heterogeneous Systems have elevated challenges on caching mechanism, memory controller design, power management, and memory bandwidth management. While solutions proposed work well for architectures with discrete GPUs, they are still inapt for the challenges of HSA systems.

The goal of this project is to explore HSA memory subsystem components in detail to understand challenges involved, and to come up with mechanisms to overcome them. Critical memory subsystem components like Last Level Cache (LLC), Memory Scheduling, and Dynamic Random-Access Memory (DRAM) controller need critical attention to achieve comparable sustained performance to traditional Chip Level Multiprocessor (CMP) architectures.

In this work, we evaluate the memory subsystem for HSA. We propose constrained shared LLC and evaluate the performance impact in such systems. We finally propose a 2-level memory access scheduling algorithm, which reduces the effective CPU memory access latency by up to 86% in heavily loaded HSA.

# Contents

**Acknowledgements** i

**Abstract** ii

**Contents** iii

**List of Figures** v

**List of Tables** vi

**1 Introduction** 1

**2 Motivation** 4
- 2.1 Traditional GPGPU Architecture . . . . . 4
  - 2.1.1 Pointer Dereference . . . . . 4
  - 2.1.2 Data Copy Overhead . . . . . 5
  - 2.1.3 Memory Capacity Limitation . . . . . 5
- 2.2 Memory Architecture in HSA . . . . . 5
  - 2.2.1 Shared vs. Private LLC and Different Cache-Line Size . . . . . 6
  - 2.2.2 Cache-line Replacement and Partitioning Policy . . . . . 7
  - 2.2.3 Memory Access Scheduling . . . . . 8

**3 Problem Statement** 9

**4 Efficient Mmemory Hierarchy Design for HSA** 10
- 4.1 Shared vs. Split LLC Organization . . . . . 10
- 4.2 Constrained Partitioning in Shared Cache . . . . . 10
- 4.3 Effective Memory Access Scheduling for HSA . . . . . 11

**5 Experimental setup** **13**
5.1 Simulator Framework . . . . . 13
5.1.1 gem5-gpu . . . . . 14
5.1.2 Dinero IV . . . . . 15
5.1.3 DRAMSim2 . . . . . 15
5.2 Benchmarks . . . . . 15
5.2.1 Issues . . . . . 16

**6 Results** **18**
6.1 Simulation configuration rationale . . . . . 19
6.2 Simulation results . . . . . 19
6.2.1 Shared vs. Split LLC Organization . . . . . 19
6.2.2 Sensitivity of LLC Occupancy . . . . . 21
6.2.3 Memory access scheduling for HSA . . . . . 22

**7 Conclusion** **30**

**8 Future work** **31**

**Bibliography** **32**

# List of Figures


1.1 CPU-GPU system with Discrete Devices . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Heterogeneous System Architecture . . . . . . . . . . . . . . . . . . . . . . . . . 2

6.1 Average CPU IPC for bp3 with 4 SMs and 16 SMs for unconstrained shared vs split LLC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
6.2 Legends for Figure 6.1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
6.3 LLC occupancy sensitivity of CPU programs in bp3 . . . . . . . . . . . . . . . . . 25
6.4 Legends for Figure 6.3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
6.5 LLC Hitrate for Backprop for CPU and GPU for different LLC occupancy . . . 26
6.6 LLC Hitrate for Gaussian for CPU and GPU for different LLC occupancy . . . . 26
6.7 Memory access latency for Backprop with Open and Close policies . . . . . . . . 27
6.8 Memory access latency for Gaussian with Open and Close policies . . . . . . . . 27
6.9 Main memory access latency curve of Gaussian for OCD-FRFCFS vs FR-FCFS 28
6.10 Main memory access latency histogram of CPU programs with Backprop for OCD-FRFCFS vs FR-FCFS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
6.11 Main memory access latency histogram of CPU programs with $k$-means for OCD-FRFCFS vs FR-FCFS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

# List of Tables

5.1 Workload configurations . . . . . . . . . . 16
5.2 Working benchmarks with gem5-gpu . . . . . . . . . . 17

6.1 Common configuration . . . . . . . . . . 18

# Chapter 1

# Introduction

To match up with Moore's Law, processor technology has evolved in many directions. Increasing core frequency, superscalar processing, and increasing number of cores on a single chip, are some of the very successful attempts. As these methods have already hit their physical boundaries [1][2], the focus has shifted to computation cores, that are heterogeneous in nature. Accelerators like GPUs were originally designed for performing highly concurrent vector calculations to support graphic processing and multimedia application like gaming, scientific applications like Computer Aided Design (CAD). However, massive multithreading capabilities and higher energy efficiency have led the development to the era of GPGPU computing. Programming models like NVIDIA CUDA[3], and OpenCL[4] are widely adopted for offloading such computations to GPUs.

Traditionally, discrete GPU devices are connected through Peripheral Component Interconnect-Express (PCIe) bus (See Figure 1.1). In this setup, both CPU and GPU devices have their private, and disjoint memory hierarchies. Hence, any data required by GPU computation has to be first copied from CPU main memory to GPU memory through PCIe bus before GPU can be invoked to process the data. Similarly, for GPGPU computation, data required by any kernel is copied into GPU memory first, and the kernel is launched. Once the kernel exits, the processed data is copied back to CPU main memory.

There are limitations and overhead involved due to disjoint sets of memory hierarchies in this architecture. Some of the issues include Pointer dereference, Data copy overhead and Memory capacity limitation; which are discussed in more detail in Chapter 2.

In Heterogeneous System Architecture (HSA), heterogeneous computation cores such as CPU, GPU and Digital Signal Processor (DSP) are integrated on same die. HSA system architecture specification[5] provided by HSA Foundation also specifies that all compute cores in HSA must have access to a unified virtual memory address space. The aim of HSA is to

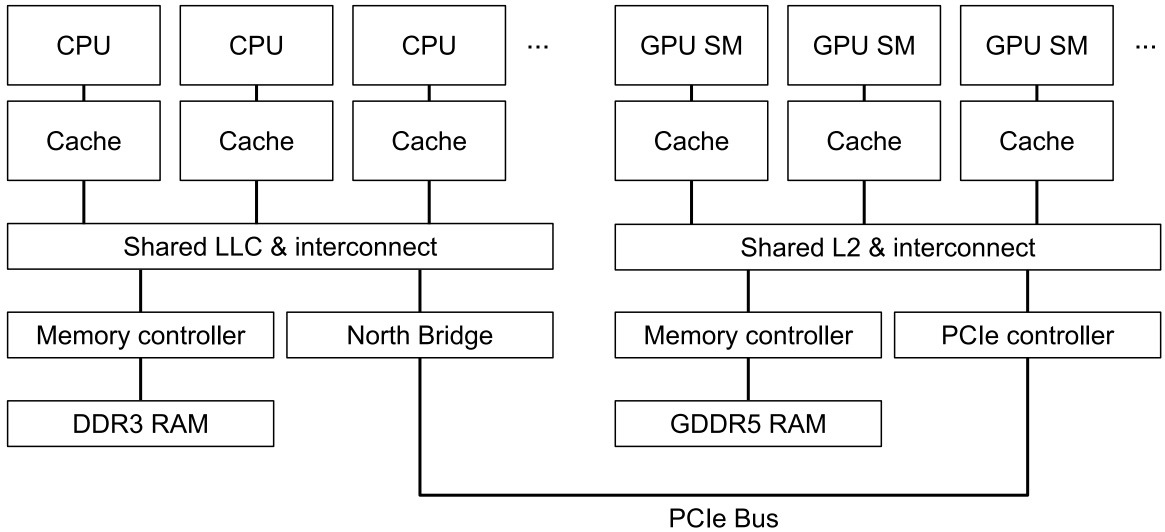



Figure 1.1: CPU-GPU system with Discrete Devices

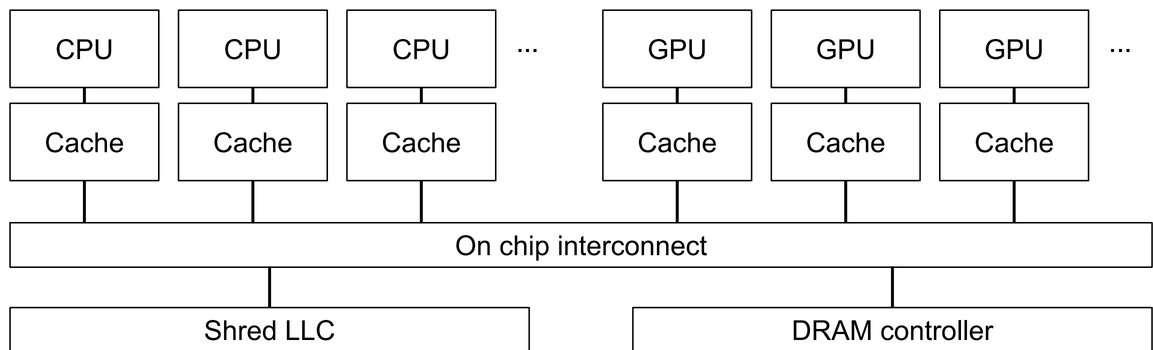



Figure 1.2: Heterogeneous System Architecture

reduce the communication latency between such computation cores, and to relieve programmer from the task of moving data between disjoint memory of such devices.

Most widely adopted HSA designs include CPU cores and GPU SMs on same chip. Intel Sandy Bridge[6], AMD Fusion[7], and nVidia Denver[8] are some examples of such implementation. Figure 1.2 shows an example of one such implementation with shared LLC between CPU cores and GPU SMs. Here, the memory consistency is maintained through on-chip interconnect.

Issues pertaining to HSA memory system are shown in Section 2.2. We then study the implications of constrained sharing of LLC and memory access scheduling. The results of which are explained in Chapter 6.

# Chapter 2

# Motivation

In this chapter, issues pertaining to traditional GPGPU architectures are discussed in detail, followed by the discussion of how they are addressed by HSA. Moreover, issues relevant to HSA memory hierarchy are discussed in detail, giving motivation of the project to solve some of those issues.

## 2.1 Traditional GPGPU Architecture

Traditional GPGPU architectures have several issues and overheads, as mentioned in Chapter 1.

### 2.1.1 Pointer Dereference

When a CPU application uses pointers to access data, dereferencing such pointers is a straightforward task, because the pointers point to the same virtual memory address space. However, when the same computations are performed on GPU, due to the disjoint set of memory, it becomes impossible to directly dereference such memory accesses. One of the ways in which applications handle this issue is by copying all required data structures in an array, and use indexes for referencing memory through the course of execution on GPUs. This quickly becomes infeasible when the data objects are dynamically added and/or removed during execution. Few such data structures are sorted linked list, sparse graph and sparse matrices.

Recent CUDA 6 supports unified memory access using new cudaMallocManaged API call[9]. This is handled by copying required data between CPU and GPU memory on pointer dereference by the kernel on GPU. Even though it relieves the programmer from the burden of marshalling and unmarshalling data, overhead of transferring data between disjoint memory sets still persist.

### 2.1.2 Data Copy Overhead

For each kernel execution on GPU, data has to be copied between CPU and GPU memory. Hence each GPU computation involves data copy overhead. Even though the time required for copying huge data structures is amortized by the actual compute time, the power consumed by the copy operation stays an issue. For computations like scalar multiplication of large vectors or matrix to vector multiplication, GPU becomes a prime candidate for choice of computation core, due to it's enormous parallel processing power. In such computations, the number of accesses to transferred data may not be enough to amortize the latency and power consumption of the copy operation.

### 2.1.3 Memory Capacity Limitation

Typical GPUs currently being used for GPGPU computations have less than 6GB of on-board memory, while the CPU memory are typically 5x-20x higher in High Performance Computing (HPC) environments. Any application with higher memory footprint than available, has to be treated differently by streaming the data from CPU memory, just before the data is required by the computation. The task becomes more complicated when the data access patterns are irregular in nature, for example, graph traversal algorithms.

## 2.2 Memory Architecture in HSA

HSA foundation has defined system architecture specification[5], that tries to address the issues discussed in Section 2.1. As a result of the requirement of unified address-space, both CPU and GPU can share pointers without any additional transfer overhead. The data copy overhead between CPU memory and GPU memory is eliminated as the memory is unified into a single virtual address space. This feature, zero-copy[10], is essentially removal of data copy overhead. HSA architecture, as a result of a unified address-space, supports large amount of addressable main memory to both CPU cores and GPU SMs and allows pointer dereferences from all compute cores.

With unified address-space, there are new issues that arise as the memory architecture has to handle data accesses from all heterogeneous computing cores. This results in issues at different components of memory hierarchy, as discussed below.

### 2.2.1 Shared vs. Private LLC and Different Cache-Line Size

In traditional architecture, GPU memory accesses and CPU memory accesses go to disjoint sets of memory, resulting in effectively lower bandwidth requirement on both memory hierarchies. In HSA, the main memory bandwidth becomes a shared resource between CPU cores and GPU SMs as all LLC misses map to the same memory bus. Hence, efficient utilization of LLC becomes more critical. If CPU and GPU working sets exceed their LLC capacity, the performance starts degrading drastically.

Another issue at LLC is deciding cache line size. In traditional architecture with discrete GPUs, the cache line size for GPU is conventionally 128 or 256-bytes wide, while for CPU cache, it is conventionally 64-bytes wide. Efficient cache designs have been proposed that can handle different cache line sizes for different regions of a single cache[11] for CMPs. In HSA, both CPU and GPU may share LLC. In such case, LLC has to handle requests of different line-width from different computation cores. Moreover, using a single cache-line size for LLC, may result in poorer performance in different scenario.

Two basic LLC implementations are possible in this scenario.

- Shared LLC across CPU cores and GPU SMs
- Private CPU-LLC, shared across all CPU cores; private GPU-LLC, shared across all GPU SMs

While, all earlier proposed designs have used LLC that are private between CPU cores as well as private between GPU SMs, we proposed shared LLC between all CPU cores and GPU SMs.

When LLC is shared across CPU cores and GPU SMs, and one of the CPU or GPU is mostly idle, the large part of the idle LLC can be utilized by the other device to gain speedup. However, GPUs in general access memory at much higher rate as compared to CPU accesses. Hence, if the LLC is shared, frequent GPU accesses will evict many of the performance-critical cache lines brought in by CPU cores, resulting in poorer performance for CPU cores.

In the case of private CPU-LLC + GPU-LLC situation, there is no interference between cache lines brought by CPU and GPU. Moreover, as explained above, different cache line size can be employed for respective CPU-LLC and GPU-LLC. However, when only one of the CPU or GPU intensive application is being run (typically the use case of low-power mobile devices), one of the LLC stays under occupied. While the idle cache capacity could help in improving the performance of other cores, it is not directly possible to do so in this implementation. This increases the off-chip memory access traffic, which results in higher energy consumption.

Having a good LLC design and policy that can perform better, as well as perform well in terms of power. We explore the shared LLC design for HSA architecture in this work.

### 2.2.2 Cache-line Replacement and Partitioning Policy

When a LLC miss occurs, a new cache line has to be brought in by the memory controller. Except for the cold cache misses, this involves invalidating one of the existing cache line, and replacing it by the incoming data. The problem of sharing LLC across competing applications on different CPU cores have been studied extensively in literature. These schemes can be classified in two sub-categories.

- Coarse-grain partitioning
- Fine-grain partitioning

While coarse-grain partitioning methods are easy to implement at hardware level[12], they do not scale well with increasing number of cores. This has led to the development of techniques that can perform fine-grain partitioning with minimal hardware changes[13].

More relevant work in similar direction include techniques to detect thrashing applications, and prevent them from evicting useful cache lines from LLC[14]. Sophisticated Memory Level Parallelism (MLP) aware policies can take into account the bank-level parallelism that can be exploited while replacing cache lines[15]. Some more relevant work has been done for improving replacement policies in CMPs[16][17].

In HSA architecture with shared LLC between CPU cores, and GPU SMs, the choice of evicting cache line becomes complex, and it becomes essential that the replacement policy can select cache line with minimal overhead, while maintaining desired cache partitioning. The replacement policy should take into account that the GPUs are designed to be latency tolerant as compared to CPUs, which are typically sensitive to memory access latency. Hence a cache miss for CPU access potentially has more negative impact than that of the GPU. In earlier research for HSA[18], it is observed that even after bypassing LLC for a part of all GPU accesses, the performance degradation in GPU is not as significant as the savings of LLC footprint. More work has been done in similar direction for Heterogeneous computing units[19].

Most of the previous algorithms use different metric, designed for homogeneous cores to decide partition size for each core. Due to the heterogeneous nature of HSA, those techniques can not be directly applied to HSA. Methods to make such metric independent of the dynamic parameters for CPU vs GPU accesses are being explored by the researchers.

### 2.2.3 Memory Access Scheduling

All the LLC misses have to be served by bringing the required data from main memory. For each LLC miss, the memory controller gets a request to bring in the cache line. It's not necessary for the memory controller to address all requests in-order. Hence, in CMPs, memory access scheduling algorithms have been employed[20][21][22] to improve performance by exploiting bank-level parallelism, and increase DRAM buffer hits.

In HSA, treating both CPU and GPU access as equal may lead to poorer performance for CPU, while not yielding significant advantage to GPU performance. GPU SMs are designed to hide memory access latency with their massively parallel thread pool, and their ability to do scheduling and context switches at hardware level. On the other hand, the performance of CPU applications are typically very prone to memory access latency, because of the software mode context switches, which involve higher overhead than the access latency, and suffers memory access stalls. Depending on available Thread Level Parallelism (TLP), GPUs can tolerate limited amount of latency. Hence in HSA, giving CPU access higher priority may not always improve overall performance. While GPU memory access requests can be delayed, there still needs to be a fairness metric to guarantee eventual access.

Recently, research has been done for cooperative heterogeneous computation[23], where the load on CPU cores and GPU SMs are of same nature. The case of non cooperative load is yet to be explored, thus giving a possible direction to the project.

# Chapter 3

# Problem Statement

In Section 2.2, we saw the issues currently pertaining to HSA memory subsystems, which are either not explored in HSA context, or have a great scope of improvement. HSA systems are not only designed for High Performance Computing (HPC) applications, but are also being deployed in mobile devices as System on Chip (SoC). Hence it becomes equally important to tune HSA memory architecture for both performance, as well as energy efficiency.

This motivates our work on improving memory subsystem performance of HSA. We explore the following problems.

- Determining the cases, where Shared vs Private LLC improve performance for CPU cores and GPU SMs.
- Coming up with a metric and mechanism to determine and enforce fair partitioning of LLC shares among CPU cores and GPU SMs.
- Improving memory access scheduling issues at memory controller to improve upon latency-caused delays in CPUs, while not affecting the GPU performance significantly.

# Chapter 4

# Efficient Mmemory Hierarchy Design for HSA

In this chapter, we explain our contributions in improving the efficiency of memory hierarchy in HSA.

## 4.1 Shared vs. Split LLC Organization

Shared LLC proposal (Section 2.2.1) is attractive from the perspective of utilizing the entire cache by the only application that is running on one of the CPU or GPU device. However, the effect of unconstrained sharing of LLC is not suited for the performance as most GPGPU applications tend to dominate LLC. This has not been studied in the literature available. In this study we focus on the LLC sharing and determine whether there is a need to perform any logical partitioning to ensure certain occupancy for the CPU applications over the GPGPU application.

## 4.2 Constrained Partitioning in Shared Cache

Our results demonstrate that in unconstrained sharing of LLC, the magnitude of LLC interference from GPGPU application results in significant performance degradation for the CPU applications. This in turn requires a logical partitioning scheme for shared LLC across CPU and GPU applications. We propose and evaluate different LLC occupancy for CPU applications and show that even a small, but non-zero, occupancy for CPU applications is essential to sustain the performance of CPU applications when running with GPGPU applications. Our results

show that the shared memory channel between CPU and GPU becomes a bigger constraint for the performance of CPU application than the LLC occupancy; as the sharing of memory channel bandwidth tends to increase the latency of requests.

## 4.3 Effective Memory Access Scheduling for HSA

From our results, we find that the number of memory accesses performed by the GPGPU application is at least an order of magnitude higher than the collective CPU accesses. Due to the high number of memory access requests, the GPGPU application tends to hog the memory, which results in poorer effective bandwidth and access latency for the CPU applications.

Memory access scheduling algorithms like First Ready - First Come First Serve (FR-FCFS) tries to schedule the requests with Row Buffer hit before the request with Row Buffer miss. Due to the large number of GPGPU memory access requests and a very small visibility window, the algorithm seldom gives priority to the CPU requests, resulting in very high access latency for the latency sensitive CPU applications.

This motivated the need to design a HSA aware memory access scheduling algorithm. Some research has been done to address the issues. A specific example of a HSA aware scheme is Staged Memory Scheduling (SMS)[24]. However, the SMS scheme is costly in terms of hardware implementation due to it's complex 3-level scheduling algorithm.

Since the access pattern and requirements are very different between GPU, CPU and other HSA devices, we classify all the requests to be either latency sensitive or latency insensitive. The capabilities and priorities of all devices are easy to establish, hence all the devices can be classified in these two categories during designing stage.

Based on our binary classification of latency sensitivity, We propose a 2-level algorithm, Orchestrated Clustered Dispatching - then - FR-FCFS (OCD-FRFCFS). The first step of the algorithm is to cluster the requests based on their latency sensitivity, which is explained in Algorithm 1. These clusters are then dispatched to FR-FCFS scheduler, which reorders the requests within each cluster to improve the overall Row Buffer Hitrate (RBH). This allows faster dispatch of latency sensitive requests, as well as preserves the Row Buffer Hitrate by performing FR-FCFS scheduling on each cluster.

**Data**: A request $R$ to be scheduled; Memory controller queue $Q$; An adaptable parameter $N$, where $0 < N \leq Q.capacity$; Aging timeout tolerance for latency-insensitive requests $T$
**Result**: Memory controller queue $Q$, containing request $R$
**if** *$R$ is a latency sensitive request* $\wedge$ *$Q.Top()$ is not older than $T$* **then**
  **if** $\exists$ *a latency sensitive request $R'$ in memory controller queue $Q$* **then**
    $pos = Q.LastIndexOf(R')$;
    $Q.InsertAt(pos + 1, R)$;
  **else**
    $Q.InsertAt(N, R)$;
  **end**
**else**
  $Q.InsertAtEnd(R)$;
**end**

**Algorithm 1:** OCD-FRFCFS algorithm

# Chapter 5

# Experimental setup

This chapter explains the simulation framework and the benchmarks used in our experiments.

## 5.1 Simulator Framework

There are two simulation methodologies commonly employed while analyzing memory subsystems.

- Trace based simulation
- Cycle accurate simulation

In trace based simulations, first the memory access traces are generated using instrumentation tools like Pinatrace[25] for PinTool, or Lackey[26] for Valgrind. Once the traces are generated, a cache hierarchy simulator, like Dinero IV[27], can be employed to generate statistics on different points in memory hierarchy. The disadvantage with trace based simulation is that it often fails to capture the dimension of time. Memory accesses performed at two different frequencies can have significant difference in performance, power, latency. However they are seen by trace simulator as the same.

In cycle accurate simulation, the programs are run on an architectural simulator like gem5[28], or GPGPUsim[29]. The simulators simulate entire architectural state of processor including registers, pipeline stages, cache hierarchy, memory controller, etc on each clock cycle. The results of such simulation gives more accurate statistical data to analyze the behaviour of programs, and simulated architecture. However these simulations have huge slowdown as compared to trace based simulations.

In our earlier study, we used cycle accurate simulator Multi2sim[30] version 4.2. Multi2sim has the ability to simulate both CPU and GPU architectures, while allowing user configurable split memory hierarchy. While gem5, and GPGPUsim are more mature simulators, their support is limited to CPU only, and GPU only simulations respectively. However, the Multi2sim has many shortcomings in terms of portability and configurability. In favor of time, and flexibility, we dropped the Multi2sim.

We finally evaluate our results on gem5-gpu[31], which is a fusion of gem5 and GPGPUSim simulators. Being highly unstable and under heavy development, gem5-gpu was not chosen in the beginning. Since the implementation of GPU atomic memory accesses in December-2014[32], the gem5-gpu has become practical for the project.

Full system simulation, being too time consuming, forced us to use trace based simulation as a supplement. To make the trace based simulation meaningful, we generate the traces with physical address, timestamps and source application ID. This allowed us to capture enough information about each memory access. For the trace based simulations, the memory subsystem components were simulated using a combination of modified Dinero IV and modified DRAMSim2[33]. Brief introduction, and the changes made to all simulators are shortly explained in the following chapters.

### 5.1.1 gem5-gpu

In gem5-gpu, the CPU cores are simulated using gem5 simulator code, while the GPU SMs are simulated using GPGPUsim code. Both of these work in lockstep synchronization at a very fine granularity of 1 pico second. The cache side interface from the GPGPUsim SMs are exposed to gem5 as processing cores. This allows simulation of GPU's global memory hierarchy by Ruby[34] component of gem5.

The original source code of gem5-gpu doesn't support 3-level cache hierarchy or configurable split/shared caches. We modified the simulator to allow split/shared 3-level cache configuration. Moreover, the original code generates statistics at the end of simulation. We modified the code to generate statistics after every $1ms$ of simulated time. The timeline based statistics enables us to explore and track the details throughout the lifecycle of CPU workloads and GPU kernel launches.

Since our interest is in studying shared/split LLC and memory controller, beyond the private L2, we use the memory accesses experienced at private L2 cache. This drastically reduces the size of traces, while allowing rapid simulation of remaining cache hierarchy and memory components. Moreover we used a flat memory model with fixed main memory access latency

to generate the traces. This eliminates the effects of memory access latency onto the execution of the workloads.

### 5.1.2 Dinero IV

Dinero IV is a time-independent trace simulator for uniprocessor cache hierarchy.

For simulating multi-source cache hierarchy, we used a multi-core version of Dinero IV, called DineroIVmc[35] which is created by CSRL lab at University of North Texas. We modified the DineroIVmc to allow tracking of the timing and access source information for all accesses. The original software only generates statistics for entire trace simulation. We also added the code to log the LLC misses, along with the timing and source information to be able to use DRAMSim2 in conjunction with DineroIVmc.

### 5.1.3 DRAMSim2

DRAMSim2 is a cycle accurate model of DRAM memory controller, the DRAM modules which comprise system storage, and the buses by which they communicate[36].

Instead of using the DRAMSim2 front-end, we only use the shared library for our simulation. This allows us to keep track of each individual requests, source and latency. We also modified the internal code of the shared library, to facilitate the information storage and retrieval. The proposed algorithm was implemented in DRAMSim2, along with the existing FRFCFS algorithm. Finally, to be able to simulate different version of DRAM models, newer high frequency DDR3 and DDR4 configurations were implemented.

## 5.2 Benchmarks

For our evaluation, we use the CPU benchmark suite PARSEC[37] version 3.0, which is designed to represent emerging workload, and represents a wide range of shared-memory programs for chip multi processing workloads. For the GPU workloads, we use benchmark suite RODINIA[38] no-copy, which is designed for heterogeneous computing infrastructure using CUDA. In contrast to the unmodified RODINIA benchmark suite, the no-copy version doesn't perform any host-to-device and device-to-host data transfers. It uses unified virtual addresses to access the data across CPU and GPU.

Reference paper[18] uses SPEC2006 benchmarks[39]. The reason for using PARSEC benchmark suite over SPEC2006 benchmark suite is that the PARSEC benchmarks are designed to

stress on chip multiprocessor architecture, resulting in better testing of on-chip interconnects, and memory hierarchy. Many SPEC2006 benchmarks are compute intensive, and stresses on efficiency of compute units in a processor, which is not the goal of our study.

Reference paper[18] uses AMD APP SDK[40] sample applications as GPU benchmarks. However, the sample applications are included for illustration purposes, and may not reflect majority of real workloads. Moreover, the AMD APP SDK sample applications performs device-to-host and host-to-device data copies, which affects the simulation statistics. We use thee RODINIA no-copy benchmarks to investigate real workload condition in HSA.

Each workload in all the simulations consists of 4 PARSEC benchmarks running on 1 CPU core each, and one RODINIA no-copy benchmark running on the GPU. We use single thread version of each PARSEC benchmarks and bind it to a CPU core at the beginning of simulation. The workload used in this report are enlisted in Table 5.1.

### 5.2.1 Issues

Due to somewhat unstable nature of gem5-gpu simulator, very few benchmarks were able to run to completion. Although the website[41] shows that 14 of the RODINIA no-copy benchmarks should work, only 8 of the PARSEC benchmarks, and 9 of the RODINIA no-copy benchmarks were able to run to completion. The working benchmarks are listed in Table 5.2.

During the simulation, only 9 workloads, consisting of 6 PARSEC benchmarks and 3 RODINIA benchmarks, simulated to completion with least crashes. All other workload configurations didn't yield enough data to produce any meaningful results. Part of the increased instability of gem5-gpu can be attributed to the modification done to allow 3-level cache hierarchy and split/shared configuration options.

| Name | PARSEC | RODINIA |
|---|---|---|
| bp1 | Blackscholes, Canneal, Dedup, Fluidanimate | Backprop |
| bp2 | Blackscholes, Dedup, Fluidanimate, Streamcluster | Backprop |
| bp3 | Blackscholes, Canneal, Dedup, StreamCluster | Backprop |
| ga1 | Blackscholes, Canneal, Dedup, Fluidanimate | Gaussian |
| ga2 | Blackscholes, Dedup, Fluidanimate, Streamcluster | Gaussian |
| ga3 | Blackscholes, Canneal, Dedup, StreamCluster | Gaussian |
| km1 | Blackscholes, Canneal, Dedup, Fluidanimate | $k$-means |
| km2 | Blackscholes, Dedup, Fluidanimate, Streamcluster | $k$-means |
| km3 | Blackscholes, Canneal, Dedup, StreamCluster | $k$-means |

Table 5.1: Workload configurations

| Benchmark Suite | Benchmarks |
|---|---|
| PARSEC 3.0 | Blackscholes, Canneal, Dedup, StreamCluster, FluidAnimate, FreqMine, Swaptions, BodyTrack |
| RODINIA no-copy | Backprop, BFS, Cell, Gaussian, *k*-means, LUD, MummerGPU, ParticleFilter, StreamCluster |

Table 5.2: Working benchmarks with gem5-gpu

# Chapter 6

# Results

Our simulations model CPUs after x86 architecture, and the GPU SMs after NVIDIA Fermi[42] architecture. The architecture configuration used for simulations is shown in Table 6.1. The simulation took average of 12 days to complete.

| CPU | |
|---|---|
| Core | 4 cores, 2.5 GHz, Out of Order, 192-entry Reorder buffer |
| L1 Cache | 4-way 32KB private D-cache, 2-way 16KB private I-cache, 128 byte line width, LRU replacement |
| L2 Cache | 8-way 128KB private unified, 128 byte line width, LRU replacement |
| **GPU** | |
| Core | 4 SMs, 700 MHz, 32 warp size, 32K shader registers/warp, 48 warps/SM, 2 scheduler/SM, Greedy then oldest (GTO) scheduling |
| Shared Memory | 48 KB/SM, 128 byte line width |
| L1 Cache | 4-way 32KB private unified, 128 byte line width, LRU replacement |
| L2 Cache | 8-way 256KB private unified, 128 byte line width, LRU replacement |
| **Memory** | |
| LLC (Shared) | 16-way 6MB unified, 128 byte line width, LRU replacement |
| LLC (Split) | 16-way 4MB unified for CPU, 16-way 2 MB unified for GPU, 128 byte line width, LRU |
| Memory Controllers | 1 controller, 64-bit interface, Open Page policy, FR-FCFS scheduling |
| DRAM | DDR3-1600, 2 Ranks, 8 Banks/Rank, 8 Devices/Rank, 2Kb row-buffer |

Table 6.1: Common configuration

In this chapter, we will briefly see rationale behind the simulated architecture configurations.

Then the simulation results, and conclusions drawn from them are explained in detail.

## 6.1 Simulation configuration rationale

The architecture we simulate is roughly modeled after AMD's A8 series of mobile Accelerated Processing Unit (APU). The A8-4500M processor[43] contains 4 CPUs operating at 1.9-2.8 GHz frequency, and 256 shader cores operating at 496-685 MHz. In high end A8[44] and A10[45] series devices, there are up to 512 shader cores operating at 720 MHz frequency.

The configuration rationale is also in line with low power mobile devices. Nvidia Tegra series of mobile processors, namely Tegra K1 and Tegra X1 includes GPU cores based on Kepler and Maxwell architecture. The Tegra K1[46] has 4 ARM cores operating at 2.3 GHz, and 192 shader cores operating at 852 MHz, with total 5 Watts of power consumption. The newer Tegra X1[47] has 4 ARM A57 and 4 low power ARM A53 cores (frequencies undisclosed), and 256 shader cores operating at 1 GHz, with total 10 Watts of power consumption.

Initial simulations were run with 16 SMs, which corresponds to 512 shader cores. Due to the suspicion that higher number of GPU SMs would magnify the impact of GPU SMs on CPU cores, we simulated the architectures with 4 SMs and 8 SMs.

We finally use the configuration with 4 SMs for the rest of simulations, which corresponds to 128 shader cores, as our baseline architecture. While most HSA implementation have much powerful GPU components, we use less powerful devices and still show that the performance implications are significant.

## 6.2 Simulation results

In this section we show our result in coherence with the discussion in Chapter 4. We identify performance bottlenecks, rationalize the claims made about the proposed solution to subvert the performance implications in HSA.

### 6.2.1 Shared vs. Split LLC Organization

In this configuration, we simulate the unconstrained shared LLC vs the split LLC configurations as shown in Table 6.1. We analyze the cache hierarchies for 3 different GPU configurations. We classify them as moderate, mildly aggressive, highly aggressive configurations corresponding to 4 SMs, 8 SMs and 16 SMs.

# 6. RESULTS

We show the average Instructions Per Cycle (IPC) curves of the CPU benchmarks for bp3 workloads in Figure 6.1 for moderate and highly aggressive configurations. The kernel entry points are marked with vertical lines wherever required.

Since Backprop consists of 2 GPU kernels, the first kernel launch line indicates launch of kernel-1. The second line indicates the launch of kernel-2 in unconstrained LLC configuration. The third line indicates the launch of kernel-2 in split LLC configuration.

For unconstrained sharing with 4 SM configuration, the IPC of CPU application Stream-Cluster is reduced by 25%, from 0.84 to 0.63 during kernel-1 execution. For most other CPU benchmarks, difference is less than 7%. However, with memory aggressive kernel-2, the IPC for CPU application Canneal is reduced by 83%, from 0.47 to 0.08. For all other CPU benchmarks the difference is between 75% and 40%. For split configuration with moderate configuration, the IPC of CPU applications are mostly sustained during kernel-1. With memory aggressive kernel-2, however, we still see 55% IPC degradation for CPU application Canneal.

For highly aggressive GPU configuration with unconstrained shared LLC, the IPC degradation is between 84% and 73% for kernel-1 and between 98% and 80% for kernel-2. For highly aggressive GPU configuration with split LLC, the IPC degradation of all CPU applications is below 7% for kernel-1. However, the difference is 68% for Canneal and 17% for Blackscholes for kernel-2.

Similar results were observed for all other workload. Due to space constraint, we have not shown the graphs for the remaining workloads.

From the results, it's clear that even with moderate GPU configuration, all the CPU programs have huge performance degradation in unconstrained shared LLC environment. Even with split LLC configuration, it is evident that some of the CPU applications see performance degradation. Especially the streaming application like Canneal, which has larger working set with streaming memory access pattern, has a significant performance degradation with highly aggressive GPU configuration.

The ratio of LLC accesses for CPU and GPU is roughly 1:20, 1:55, 1:25 for Backprop, Gaussian and $k$-means respectively in moderate configuration. This in turn implies that unconstrained shared LLC is mostly occupied by GPGPU application. From this, we conclude that a logical partitioning scheme has to be employed to sustain the performance of CPU applications running along with GPGPU application in HSA.

Very high difference in LLC access count results in equivalently high difference in main memory accesses. In split LLC configuration, the performance degradation can be attributed to the reduced effective bandwidth due to high number GPU memory requests.

### 6.2.2 Sensitivity of LLC Occupancy

As seen in Section 6.2.1, unless a smart LLC partitioning algorithm is employed for shared LLC, the CPU applications experience significant performance degradation. This amplifies the necessity of HSA aware cache partitioning algorithm. Therefore we further explored the LLC occupancy sensitivity of such applications.

In practice, a logical cache partitioning algorithm has to be employed, along with coarse/fine grained partitioning implementation. However, due to the unstable simulation framework, and time constraint, we could not implement the partitioning algorithm. To be able to speculate the performance implications of different cache occupancy impact on performance, we analysed the sensitivity of LLC cache occupancy for CPU cores and GPU SMs by assigning them fixed portion of LLC. We simulated 6 MB total LLC, split in 20%-80%, 40%-60%, 60%-40%, 80%-20%, 100%-0% ratios for CPU cores and GPU cores respectively. Since bypassing cache in 3-level hierarchy was not possible with the simulation environment, we assign 6 MB LLC to the CPU cores and 1 MB LLC to the GPU SMs, to emulate 100%-0% configuration.

The IPC for CPU applications throughout the simulations are shown in Figure 6.3. We also show the average LLC hit rate curves for Backprop and Gaussian in Figures 6.5, 6.6 due to their very different nature of hit rate during kernel execution.

In Figure 6.3, we show the variation in IPC of CPU benchmarks in bp3 with different LLC partition allocated to the CPU cores. Due to the space constraint, we just plot the 100%-0%, 60%-40% and 20%-80% ratios in graph. The important thing to notice here is that only for StreamCluster there is a significant difference of 24% in IPC. Since 20% CPU LLC is not enough to hold the working set for StreamCluster, we see significant performance degradation when kernel-2 is executed. Canneal sees similar performance degradation during kernel-2 execution. Blackscholes experience 12% of IPC degradation during kernel-2 execution. We do not show Dedup in graph, since the performance degradation is 1%. The lower IPC before kernel-1 launch can be attributed to the additional CPU LLC occupied by the CPU core that launches GPGPU program.

Although the CPU LLC share increases from 20% to 100%, there is a insignificant performance improvement, unless the LLC share is unable to hold the working set. The conclusion we derive from this result is that, as long as there is enough LLC to hold the working set of the CPU application, the gain from additional LLC share allocation is very small.

In Figure 6.5, we see that the with increase of GPU LLC occupancy from 40% to 80%, the hit rate for Backprop kernel-1 increases from 2.3% to 3.4%. For Backprop kernel-2, the increases from 6% to 9%.

From Figure 6.6, we see that the Gaussian kernel has a very high hit rate, of 71% and 75%, for 40% and 80% LLC occupancy respectively. However, the increase of LLC occupancy, only achieves 6% increase in LLC hit rate.

A noteworthy point is that the LLC hit rate drastically changes the ratio of memory accesses performed by the CPU and GPU cores at LLC and main memory. Due to the LLC filtering, the ratio of memory accesses performed by CPU cores and GPU SMs are 1:70, 1:60, 1:40 for Backprop, Gaussian and $k$-means respectively with 60%-40% ratio.

We conclude that although there is a need for a smart and HSA aware LLC partitioning algorithm, it is not the root cause of the performance degradation for many of the CPU workloads. Performance of CPU workloads with larger working set is mainly constrained by the available bandwidth and average memory access latency. Hence, we propose a new memory access scheduling algorithm, OCD-FRFCFS in Section 4.3 to sustain the performance of CPU application in HSA.

### 6.2.3 Memory access scheduling for HSA

The memory controllers typically follow open-page or close-page policy with FR-FCFS memory access scheduling. In open page policy, the row buffer is kept open till the new request refers to a different row. In close-page policy, the row buffer is closed as soon as all the pending requests are serviced. Some of the device manufacturers use proprietary methods, e.g. Intel's Adaptive Open-Page[48] is a combination of both the policies.

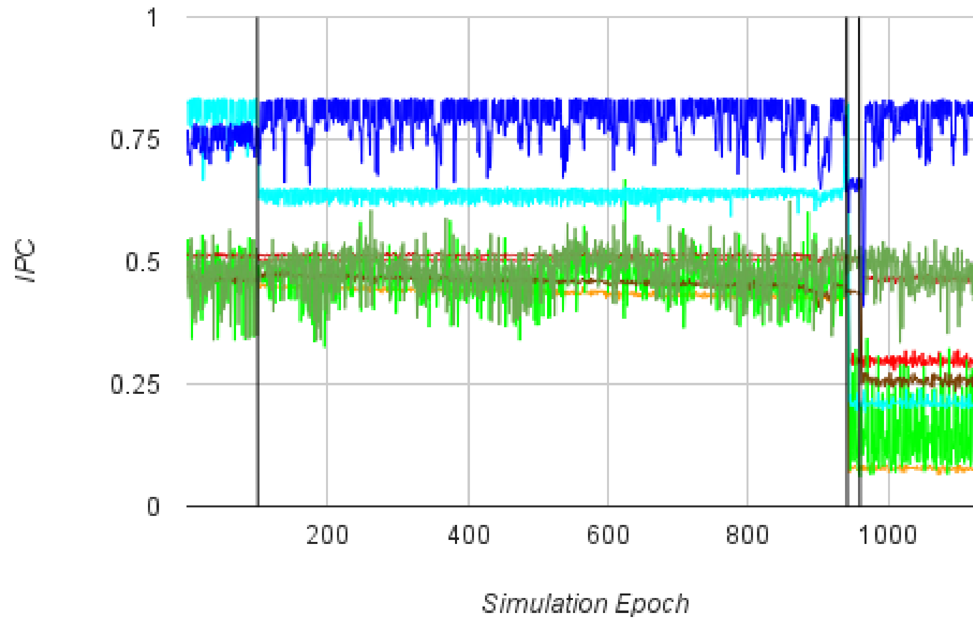
To evaluate the effect of open and close policies, we performed trace based simulation for all workloads. The latency curves for both policies are shown in Figure 6.7, 6.8. Since $k$-means kernel executes hundreds of times, the duration of each execution is very small. From the results, the average latency for CPU and GPU programs increase to 1.17x, 2.5x, 2x for Backprop, Gaussian and $k$-means for close-page policy with respect to open-page policy. This shows that Gaussian and $k$-means have fairly regular memory access patterns.

From simulations with FR-FCFS scheduling algorithm, we found the latency for the CPU requests becomes high when the concurrent GPU workload has very regular memory access pattern. For Gaussian, it is approximately 32 times, from 110 CPU cycles to 3500 CPU cycles. For $k$-means, the difference is approximately 7 ties, from 100 CPU cycles to 691 CPU cycles. This is because the FR-FCFS scheduler continuously tries to schedule the requests from GPU with high RBH, effectively starving the accesses with row buffer miss.

We show the latency curve of our proposed OCD-FRFCFS algorithm with FR-FCFS scheme for Gaussian workload in Figure 6.9. Here it is observed that the latency for CPU applications is

always constrained below 700 CPU cycles, which is 5x smaller than 3500 observed in open-page FR-FCFS. This shows that the OCD-FRFCFS algorithm performs at least 80% better than FR-FCFS algorithm. For peak latency, however, it performs 86% better, as latency reduces from 5000 CPU cycles to 700 CPU cycles. We also see that the average memory access latencies for GPU is comparable with FR-FCFS.

To be able to compare OCD-FRFCFS with FR-FCFS when GPU workload has irregular memory access patterns, we also show the latency histograms for both algorithms in Figures 6.10, 6.11. From the latency trend, it can be noticed that the OCD-FRFCFS not only outperforms the FR-FCFS with regular GPU workload, but also matches it when the GPU workload is irregular.



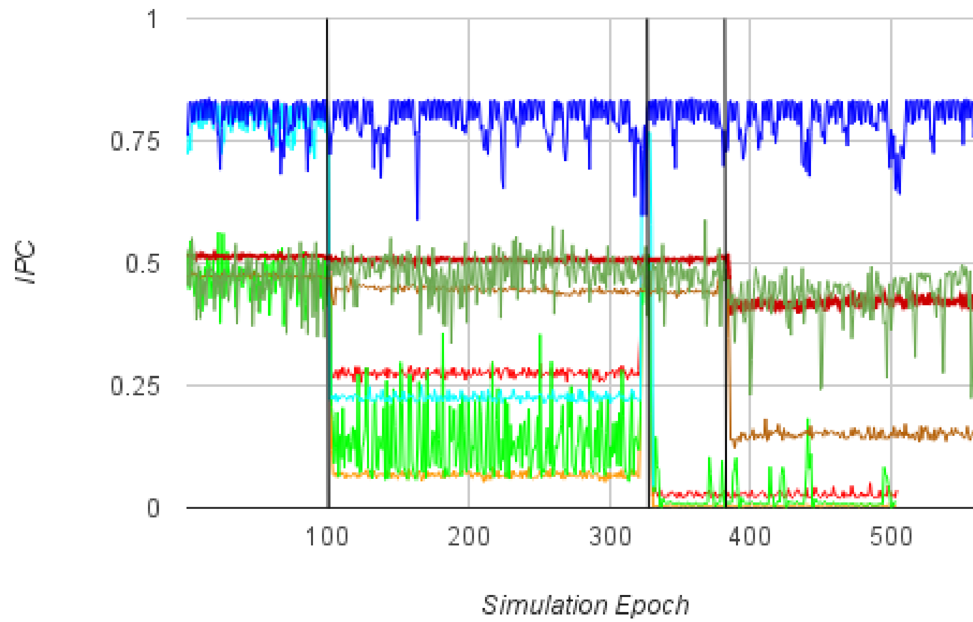




Figure 6.1: Average CPU IPC for bp3 with 4 SMs and 16 SMs for unconstrained shared vs split LLC

Shared-blackscholes
Shared-canneal
Shared-dedup
Shared-streamcluster
Split-blacksholes
Split-canneal
Split-dedup
Split-streamcluster
Kernel Launches

Figure 6.2: Legends for Figure 6.1

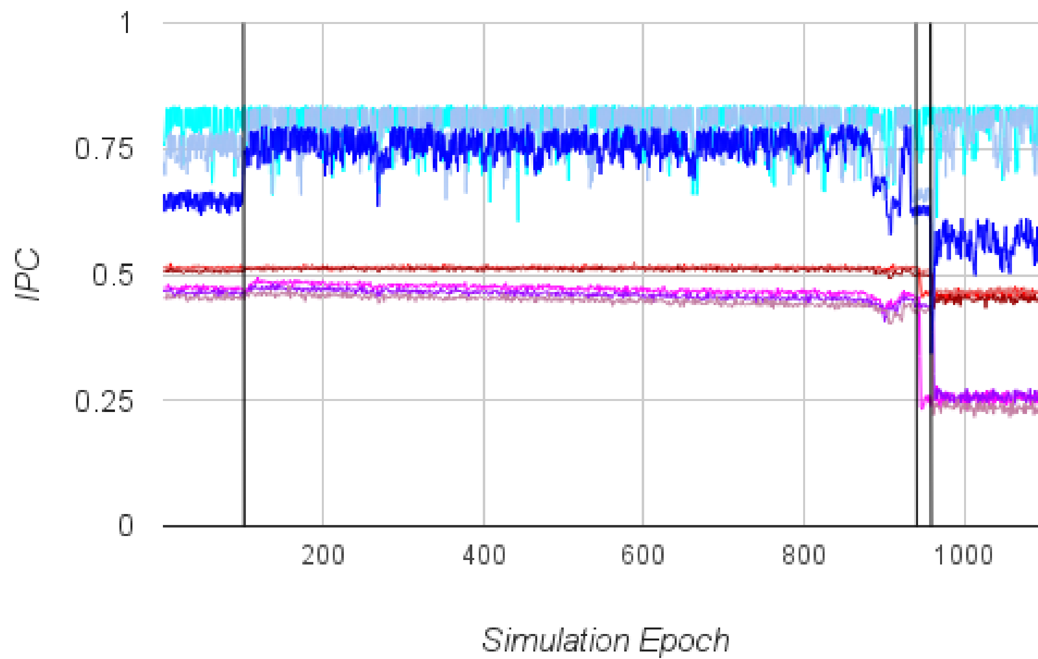



Figure 6.3: LLC occupancy sensitivity of CPU programs in bp3

100-0 blackscholes
100-0 canneal
100-0 streamcluster
60-40 blackscholes
60-40 canneal
60-40 streamcluster
20-80 blackscholes
20-80 canneal
20-80 streamcluster
Kernel Launches

Figure 6.4: Legends for Figure 6.3

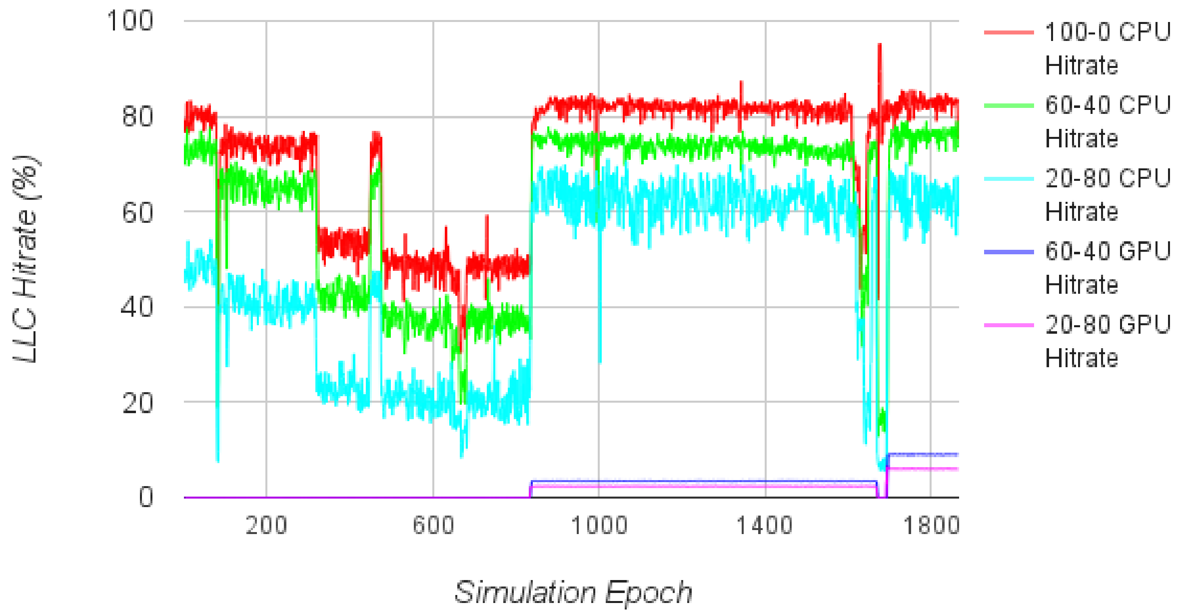



Figure 6.5: LLC Hitrate for Backprop for CPU and GPU for different LLC occupancy

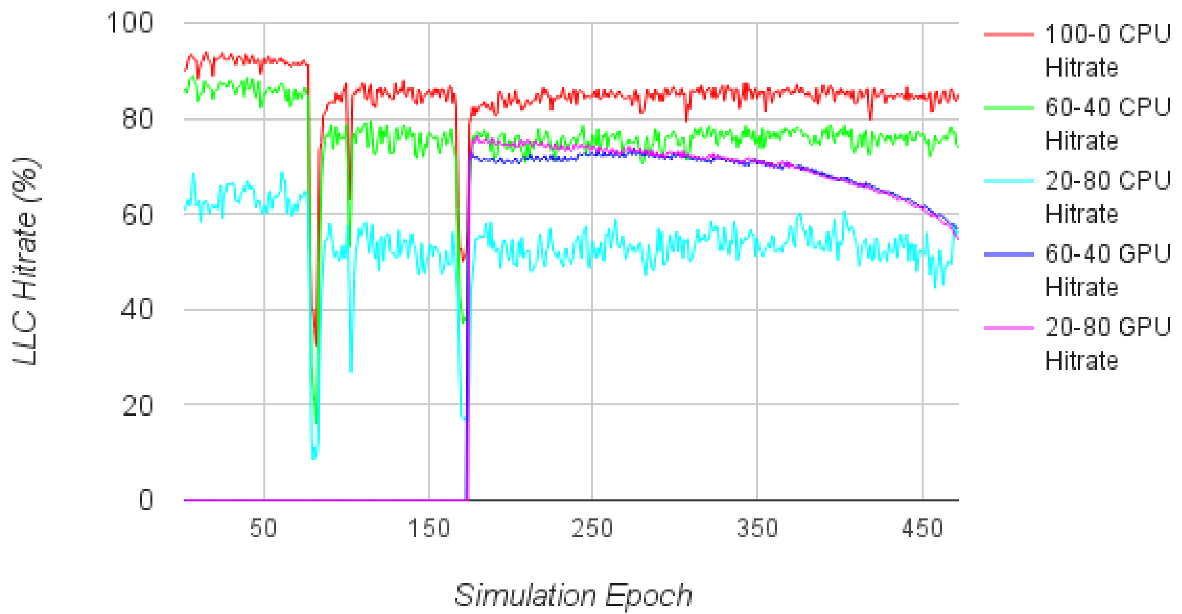



Figure 6.6: LLC Hitrate for Gaussian for CPU and GPU for different LLC occupancy

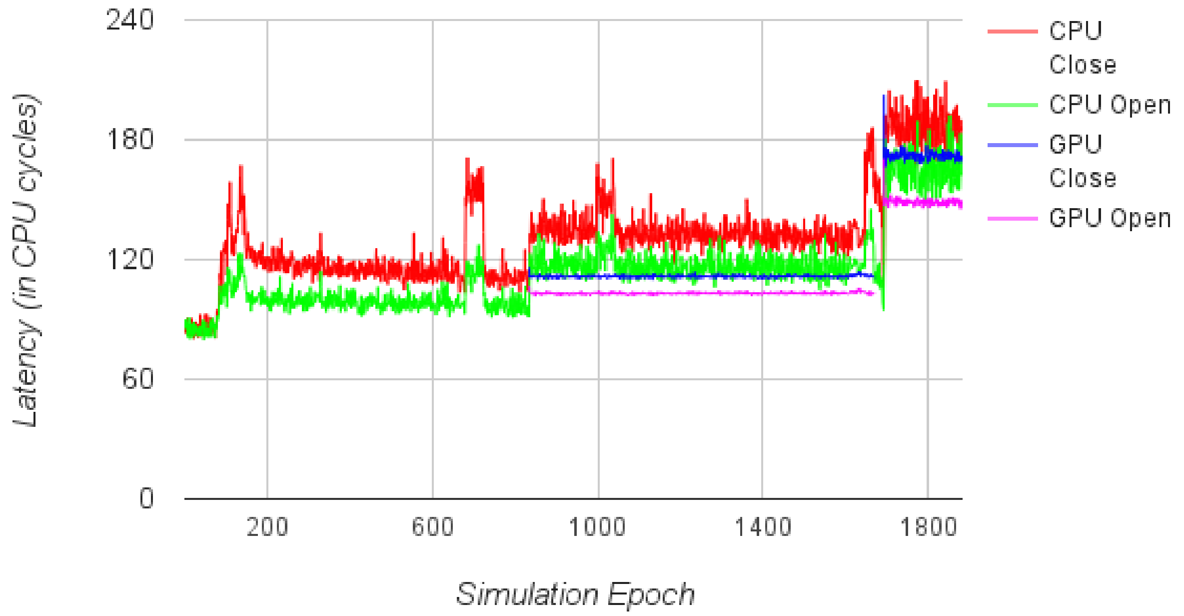



Figure 6.7: Memory access latency for Backprop with Open and Close policies

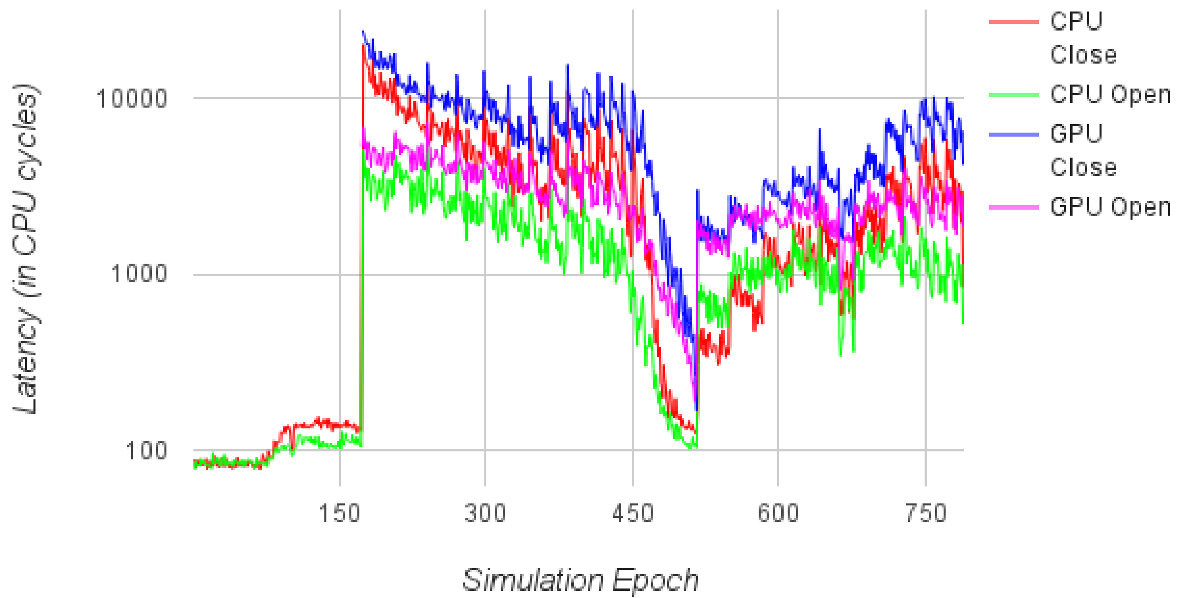



Figure 6.8: Memory access latency for Gaussian with Open and Close policies

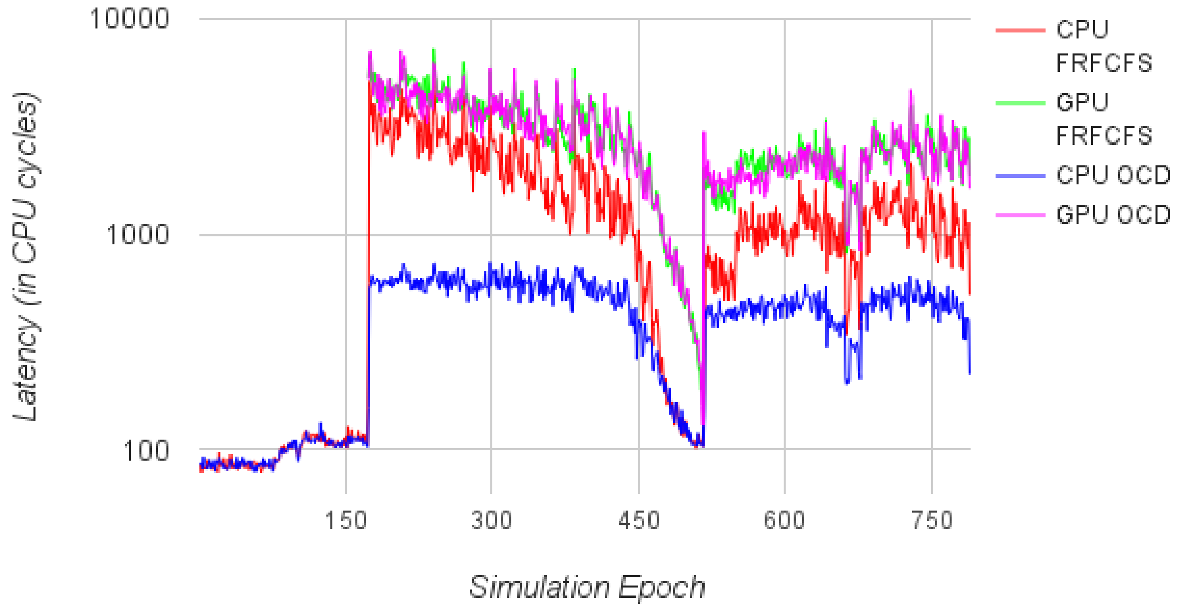



Figure 6.9: Main memory access latency curve of Gaussian for OCD-FRFCFS vs FR-FCFS

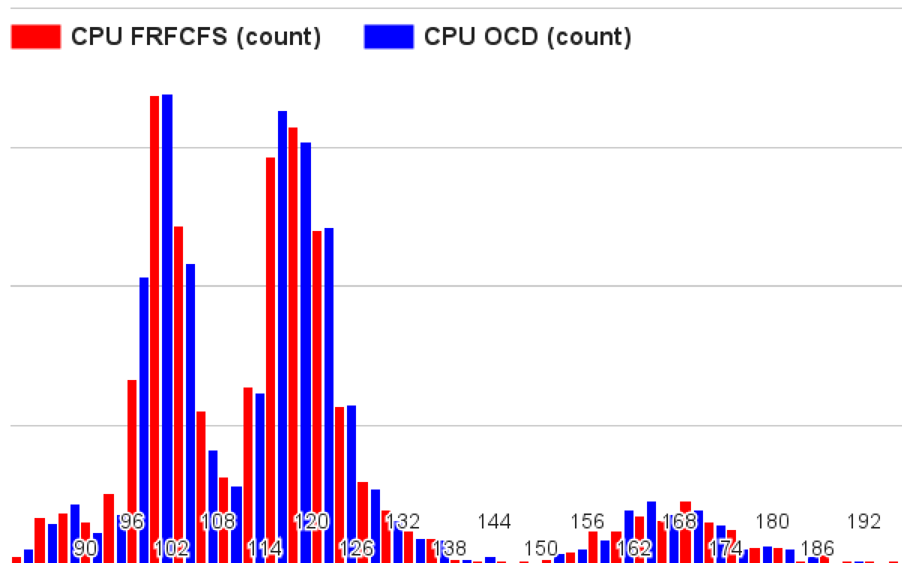



Figure 6.10: Main memory access latency histogram of CPU programs with Backprop for OCD-FRFCFS vs FR-FCFS

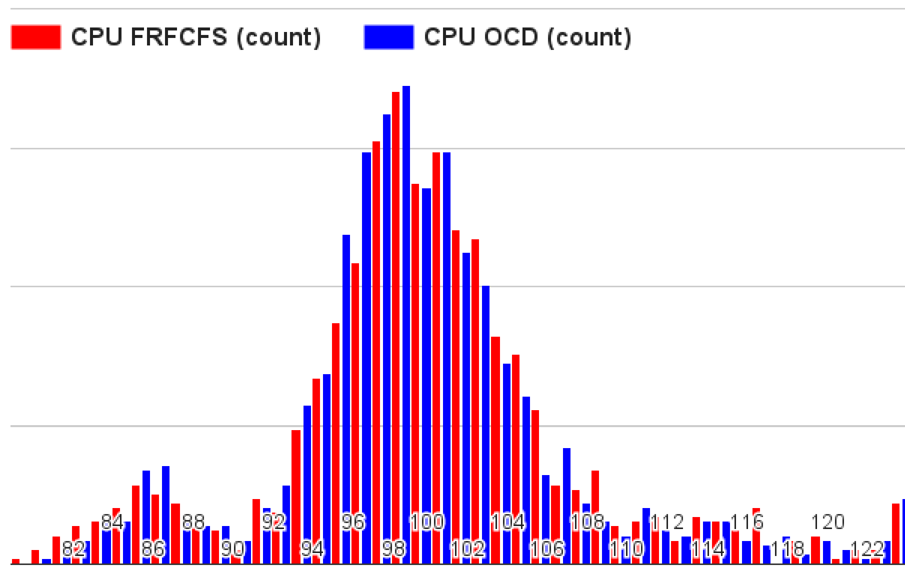



Figure 6.11: Main memory access latency histogram of CPU programs with $k$-means for OCD-FRFCFS vs FR-FCFS

# Chapter 7

# Conclusion

While much research efforts have been put in the discrete devices, the integration of processing cores such as CPUs and GPUs have created new system design challenges. Most of the recent research on HSA, has been done to improve the performance and efficiency of GPU components. The CPU components of HSA requires critical attention to prevent severe performance degradation. In this work we discover how increased LLC pressure and off-chip memory access elevates the penalties.

We identify the problems pertaining to shared cache, and conclude that the unconstrained sharing of LLC is futile. Any HSA aware LLC partitioning algorithm would significantly improve the performance for CPU, while imposing minimal effect on GPU performance.

The traditional memory controller design and scheduling algorithms perform poorly due to the different requirements and nature of computation cores in HSA. This influences the memory access latency and effective bandwidth for different devices.

The binary classification of memory requests based on latency sensitivity of the sources is a novel approach which allows smarter memory access scheduling with lower hardware costs. Our memory access scheduling algorithm OCD-FRFCFS outperforms FR-FCFS in HSA, and has lesser hardware complexity than similar approach like SMS. OCD reduces the average memory access latency by up to 86% for CPU programs running concurrently with regular GPU kernels. We also show that when running with irregular GPU kernels, OCD does not have any negative effects on average latency.

Our evaluations both support our claims and show the positive impact of our algorithm in HSA at nominal hardware cost.

# Chapter 8

# Future work

The future steps for the project includes the following:

- With future releases of gem5-gpu, simulate more workloads to verify our claims and support our algorithm OCD-FRFCFS.
- Evaluate the current LLC partitioning algorithm for their effectiveness in performing LLC partitioning in HSA.
- Compare OCD-FRFCFS with SMS.
- Instead of trace based simulation, perform detailed simulation of OCD-FRFCFS.
- Simulate heterogeneous multi-channel memory with different properties like bandwidth, latency, operating frequencies per channel.

# Bibliography


[1] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005. 1

[2] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011. 1

[3] About cuda. https://developer.nvidia.com/about-cuda. (Visited on 10/19/2014). 1

[4] Opencl - the open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/. (Visited on 10/19/2014). 1

[5] Hsa platform system architecture specification. http://www.hsafoundation.com/?ddownload=4944. (Visited on 10/19/2014). 1, 5

[6] Products (formerly sandy bridge). http://ark.intel.com/products/codename/29900/Sandy-Bridge#@All. (Visited on 10/24/2014). 3

[7] Benedict R Gaster and Lee Howes. The future of the apu–braided parallelism. *Fusion Developer Summit*, 2011. 3

[8] Fastest processors, smartphones, and tablets— nvidia tegra — nvidia. http://www.nvidia.com/object/tegra.html. (Visited on 10/24/2014). 3

[9] Unified memory in cuda 6 — parallel forall. http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/. (Visited on 10/21/2014). 4

[10] Pierre Boudier and Graham Sellers. Memory system on fusion apus. *AMD Fusion developer summit*, 2011. 5

[11] Snehasish Kumar, Hongzhou Zhao, Arrvindh Shriraman, Eric Matthews, Sandhya Dwarkadas, and Lesley Shannon. Amoeba-cache: Adaptive blocks for eliminating waste in

the memory hierarchy. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 376–388. IEEE Computer Society, 2012. 6

[12] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432. IEEE Computer Society, 2006. 7

[13] R Manikantan, Kaushik Rajan, and R Govindarajan. Probabilistic shared cache management (prism). In *ACM SIGARCH Computer Architecture News*, volume 40, pages 428–439. IEEE Computer Society, 2012. 7

[14] Yuejian Xie and Gabriel H Loh. Scalable shared-cache management by containing thrashing workloads. In *High Performance Embedded Architectures and Compilers*, pages 262–276. Springer, 2010. 7

[15] Moinuddin K Qureshi, Daniel N Lynch, Onur Mutlu, and Yale N Patt. A case for mlp-aware cache replacement. *ACM SIGARCH Computer Architecture News*, 34(2):167–178, 2006. 7

[16] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 381–391. ACM, 2007. 7

[17] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM, 2010. 7

[18] Vineeth Mekkat, Anup Holey, Pen-Chung Yew, and Antonia Zhai. Managing shared last-level cache in a heterogeneous multicore processor. In *PACT*, pages 225–234. IEEE, 2013. 7, 15, 16

[19] Jaekyu Lee and Hyesoon Kim. Tap: A tlp-aware cache management policy for a cpu-gpu heterogeneous architecture. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012. 7

[20] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010. 8

[21] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 63–74. IEEE Computer Society, 2008. 8

[22] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. *Memory access scheduling*, volume 28. ACM, 2000. 8

[23] Hao Wang, Ripudaman Singh, Michael J Schulte, and Nam Sung Kim. Memory scheduling towards high-throughput cooperative heterogeneous computing. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 331–342. ACM, 2014. 8

[24] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H Loh, and Onur Mutlu. Staged memory scheduling: achieving high performance and scalability in heterogeneous systems. *ACM SIGARCH Computer Architecture News*, 40(3):416–427, 2012. 11

[25] Pin: Pin 2.13 user guide. https://software.intel.com/sites/landingpage/pintool/docs/62141/Pin/html/index.html#MAddressTrace. (Visited on 10/18/2014). 13

[26] Valgrind. http://valgrind.org/docs/manual/lk-manual.html. (Visited on 10/18/2014). 13

[27] Dinero iv trace-driven uniprocessor cache simulator. http://pages.cs.wisc.edu/~markhill/DineroIV/. (Visited on 10/18/2014). 13

[28] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011. 13

[29] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009. 13

[30] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2012. 14

[31] Jason Power, Joel Hestness, Marc Orr, Mark Hill, and David Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *Computer Architecture Letters*, 13(1), Jan 2014. 14

[32] Main_page []. https://gem5-gpu.cs.wisc.edu/wiki/#news. (Visited on 06/06/2015). 14

[33] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16–19, 2011. 14

[34] Ruby - gem5. http://www.m5sim.org/Ruby. (Visited on 06/06/2015). 14

[35] Computer systems research laboratory - multicore dineroiv. http://csrl.unt.edu/dineroIVmc/. (Visited on 05/27/2015). 15

[36] Dramsim2. http://www.eng.umd.edu/~blj/dramsim/. (Visited on 05/27/2015). 15

[37] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011. 15

[38] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009. 15

[39] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006. 15

[40] App sdk - a complete development platform - amd. http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/. (Visited on 10/24/2014). 16

[41] gem5-gpu status - google sheets. https://docs.google.com/spreadsheet/ccc?key=0AvwlHlT78qDYdG5pRENBUWNfQUw0dXctY1ZEZjFxMXc#gid=15. (Visited on 05/28/2015). 16

[42] Peter N Glaskowsky. Nvidias fermi: the first complete gpu computing architecture. *White paper*, 2009. 18

[43] Amd a8-series a8-4500m - am4500dec44hj. http://www.cpu-world.com/CPUs/Bulldozer/AMD-A8-Series%20A8-4500M.html. (Visited on 05/30/2015). 19

[44] Amd a8-series a8-5500 - ad5500oka44hj / ad5500okhjbox. http://www.cpu-world.com/CPUs/Bulldozer/AMD-A8-Series%20A8-5500.html. (Visited on 05/30/2015). 19

[45] Amd a10-series a10-7850k - ad785kxbi44ja / ad785kxbjabox. http://www.cpu-world.com/CPUs/Bulldozer/AMD-A10-Series%20A10-7850K.html. (Visited on 05/30/2015). 19

[46] Tegra k1 next-gen mobile processor — nvidia tegra — nvidia. http://www.nvidia.com/object/tegra-k1-processor.html. (Visited on 05/30/2015). 19

[47] Tegra x1 super chip — nvidia tegra — nvidia. http://www.nvidia.com/object/tegra-x1-processor.html. (Visited on 05/30/2015). 19

[48] J.M. Dodd. Adaptive page management, July 11 2006. US Patent 7,076,617. 22