2019

Simulation, Analysis, and Optimization of Heterogeneous CPU-GPU Systems

Christopher Giles
University of Central Florida

SIMULATION, ANALYSIS, AND OPTIMIZATION OF HETEROGENEOUS CPU-GPU SYSTEMS

by

CHRISTOPHER EDWARD GILES
B.S. University of Central Florida, 2007
M.S. University of Central Florida, 2011

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2019

Major Professor: Mark Heinrich

ABSTRACT

With the computing industry's recent adoption of the Heterogeneous System Architecture (HSA) standard, we have seen a rapid change in heterogeneous CPU-GPU processor designs. State-of-the-art heterogeneous CPU-GPU processors tightly integrate multicore CPUs and multi-compute unit GPUs together on a single die. This brings the MIMD processing capabilities of the CPU and the SIMD processing capabilities of the GPU together into a single cohesive package with new HSA features comprising better programmability, coherency between the CPU and GPU, shared Last Level Cache (LLC), and shared virtual memory address spaces. These advancements can potentially bring marked gains in heterogeneous processor performance and have piqued the interest of researchers who wish to unlock these potential performance gains. Therefore, in this dissertation I explore the heterogeneous CPU-GPU processor and application design space with the goal of answering interesting research questions, such as, (1) what are the architectural design trade-offs in heterogeneous CPU-GPU processors and (2) how do we best maximize heterogeneous CPU-GPU application performance on a given system. To enable my exploration of the heterogeneous CPU-GPU design space, I introduce a novel discrete event-driven simulation library called KnightSim and a novel computer architectural simulator called M2S-CGM. M2S-CGM includes all of the simulation elements necessary to simulate coherent execution between a CPU and GPU with shared LLC and shared virtual memory address spaces. I then utilize M2S-CGM for the conduct of three architectural studies. First, I study the architectural effects of shared LLC and CPU-GPU coherence on the overall performance of non-collaborative GPU-only applications. Second, I profile and analyze a set of collaborative CPU-GPU applications to determine how to best optimize them for maximum collaborative performance. Third, I study the impact of varying four key architectural parameters on collaborative CPU-GPU performance by varying GPU compute unit coalesce size, GPU to memory controller bandwidth, GPU frequency, and system wide switching fabric latency.

This dissertation is dedicated to my wife and children.

Without your endless love and support,

my research and this dissertation would not have been possible.

ACKNOWLEDGMENTS

First and foremost, I would like to thank Dr. Mark Heinrich who guided me through my academic activities and served as my principal advisor and mentor for many years. Dr. Heinrich's tireless advisement over the years has enabled me to improve upon my research greatly and made the authoring of this dissertation possible. I consider myself truly blessed to have had the opportunity to work with Dr. Heinrich. Second, I would like to thank the members of my dissertation committee Dr. Mark Heinrich, Dr. Rickard Ewetz, Dr. Mingjie Lin, Dr. Sumanta Pattanaik, and Dr. Elena Flitsiyan for their invaluable insight and direction towards my research and the finalization of this dissertation. Third, I would like to thank my fellow laboratory cohorts Christina Peterson, Ramin Izadpanah, Anahita Davoudi, Osama Tameemi, and Pierre LaBorde for their continuous help and encouragement, through thick and thin, and for making the journey fun. Fourth, I would like to thank my Mom and Dad whose enduring patience and resolute support over my life is the greatest gift anyone has ever given me. Thanks Mom and Dad! Finally, I would like to thank the United States Department of the Navy and the Science, Mathematics, and Research for Transformation (SMART) Scholarship program committee. As a SMART Scholarship recipient, I was provided the opportunity to pursue the degree of Doctor of Philosophy while maintaining a positive work and family life balance. Receiving the SMART Scholarship was a defining moment for me and propelled me towards graduation. Thank you for this opportunity and for your confidence in selecting me as a SMART recipient.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CHAPTER 1: INTRODUCTION

With the computing industry's recent adoption of the Heterogeneous System Architecture (HSA) standard we have seen a rapid change in heterogeneous CPU-GPU processor design and implementation [1]. State-of-the-art heterogeneous processors tightly integrate multicore CPUs and multi-compute unit GPUs together on a single die. This results in a single cohesive package that brings together the MIMD processing capabilities of the CPU and the SIMD processing capabilities of the GPU with new HSA features including coherency between the CPU and GPU, shared Last Level Cache (LLC) [2, 3, 4], and shared virtual memory address spaces [5, 6]. In addition, high-level programming languages, like OpenCL 2.0, have been updated to make use of these new HSA features [7]. These advancements can potentially bring marked gains in heterogeneous processor performance and have piqued the interest of researchers who wish to unlock these potential performance gains. First with the advent of *zero copy* memory management between the CPU and GPU [3, 8, 9], where memory objects are passed between the CPU and GPU by pointer reference only, and now with the recent advent of collaborative CPU-GPU processing, where the CPU and GPU jointly process work in a coherent and shared virtual memory environment.

Recent research has shown that the collaborative CPU-GPU execution approach can result in measurable speedups over the preceding non-collaborative GPU-only approach [4, 10, 11, 12]. In the non-collaborative GPU-only approach workloads are only executed on the GPU and the entirety of the CPU is relegated to only performing the controlling functions of the GPU. However, the recent research presented here has been conducted on fixed physical systems comprising a single point of reference regarding architectural and application configuration with limited or no trade space analyses on impacts to overall system performance. Therefore, in this dissertation I explore the heterogeneous CPU-GPU processor and application design space with the goal of answering interesting research questions, such as, (1) what are the architectural design trade-offs in heterogeneous

CPU-GPU processors and (2) how do we best maximize heterogeneous CPU-GPU application performance on a given system.

To enable my exploration of the heterogeneous CPU-GPU processor and application design space, I introduce a novel discrete event-driven simulation library called KnightSim, a novel computer architectural simulator called M2S-CGM, and implement a set of heterogeneous CPU-GPU benchmarks. KnightSim is a fast discrete event-driven simulation methodology that is intended for use in the development of future computer architectural simulations. KnightSim extends an older proven event driven simulation methodology known as "The Threads Package". The Threads Package has previously been used in at least two publicly known computer architectural simulators [3, 13]. KnightSim implements events as independently executable x86 "KnightSim Contexts". By design, KnightSim Contexts encapsulate all of the functionality and interfaces associated with a single target simulated system element in an individually executable package. This implementation methodology enjoys several benefits from this approach. First, occupancy and contention, which have been proven to be a critical determinant of system performance [14], are automatically modeled by KnightSim Contexts. Other simulation methodologies, like those of Gem5 and Multi2Sim, do not do this and require additional events, state flags, and levels of abstraction to achieve a realistic occupancy and contention model. Second, executing a KnightSim Context only requires a long jump, see Sec. 2.2.1. This mechanism is faster as compared to scheduling and running an event's call-back function because a KnightSim Context's stack is not created and torn down each time the context is executed. Finally, KnightSim Context execution can be performed in parallel because each KnightSim Context is independently executable in a multithreaded environment. These properties make KnightSim a promising tool for use in the development of computer architectural simulations.

M2S-CGM provides end-to-end simulation of the system elements required to simulate modern and future non-coherent and coherent heterogeneous CPU-GPU processor architectural models.

M2S-CGM extends the multicore out-of-order x86 CPU model and multi-Compute Unit (CU) Southern Islands GPU model found in Multi2Sim [15] and adds a novel highly detailed CPU-GPU memory system model. M2S-CGM's memory system model provides coherence protocols, execution-driven discrete models of system wide occupancy and contention, CPU and GPU cache structures, directories, virtual memory mechanisms, switching fabrics, a system agent, a memory controller, and SDRAM. I provide a validation of M2S-CGM and establish that M2S-CGM provides good correlation to modern computing systems and that the information ascertained from its use is reliable and can be used for trade-off decisions in proposed architectural implementations. This provides researchers the ability to conduct a range of experiments with varying degrees of configurability in the memory system for both the CPU and GPU. In addition, I implement a set of benchmarks comprising both non-collaborative GPU-only and collaborative CPU-GPU implementations of Backpropagation, Block Matrix Multiply, Edge Detection, Nearest Neighbor, and Write. Backpropagation and Nearest Neighbor are ported from the Rodinia benchmark suite [16] and Block Matrix Multiply, Edge Detection, and Write are hand implemented. These benchmarks are fully compatible with M2S-CGM, are representative of a range of scientific applications, are well-suited for execution on the GPU, and have distinct collaborative CPU-GPU memory access patterns.

I then utilize M2S-CGM for the conduct of three architectural studies. First, I study the architectural affects of shared LLC and CPU-GPU coherence on the overall performance of non-collaborative GPU-only applications. In this study I found that enabling coherence between the CPU and GPU and sharing the LLC can lead to measurable performance gains over non-coherent CPU-GPU executions. Additionally, the results of my first study make it apparent that new performance gains are possible if applications make better usage of the CPU during GPU kernel execution time. This motivated my decision to implement my own collaborative CPU-GPU benchmarks to support the second and third study. In my second study I profile and analyze my set of

collaborative CPU-GPU benchmarks and determine how to best optimize them for maximum collaborative performance. I establish collaborative performance profiles for each of my benchmarks and use them to conduct a detailed performance analysis and report the results in this dissertation. My collaborative CPU-GPU performance profiles show overall collaborative speedups while varying CPU cores/threads from one to eight and CPU workload percentage from 20% to 80%. The results identify each benchmark's optimization points and show that my set of collaborative CPU-GPU applications can achieve speedups as high as 2.23x over that of the non-collaborative GPU-only versions. In addition, the results provide a few rules of thumb regarding how best to reason about what an unprofiled collaborative CPU-GPU application's optimized settings could be. Finally, I utilize my set of benchmark's again and study the impact of varying four key architectural parameters on collaborative CPU-GPU performance. In the study I vary GPU compute unit coalesce size, GPU to memory controller bandwidth, GPU frequency, and system wide switching fabric latency. My results show that future looking architectural changes can lead to a theoretical average speedup of 3.33x over the average speedups of our benchmark's best case collaborative CPU-GPU executions and reach a theoretical average speedup of 6.3x over that of our benchmark's non-collaborative GPU-only executions.

1.1 Research Contributions

This dissertation makes the following research contributions:

- I present a detailed discussion, with pseudocode, of the implementation of KnightSim and Parallel KnightSim and present the results of a performance analysis of KnightSim, Parallel KnightSim and three different event-driven simulation methodologies that are widely in use today. KnightSim and Parallel KnightSim simplify computer architectural simulator programmability and introduce a new cycle level parallel processing capability that can speedup

future architectural simulation.

- I present M2S-CGM and provide five benchmarks comprising both non-collaborative GPU-only and collaborative CPU-GPU implementations. M2S-CGM and its benchmarks enable execution-driven simulation-based research within the collaborative CPU-GPU design space where previously not possible.

- I present the results of my first architectural study regarding the effects of shared LLC and CPU-GPU coherence on the overall performance of non-collaborative GPU-only applications. This work provides new directions to researchers by establishing that the CPU and GPU should be made coherent and share LLC and virtual address spaces. This eliminates the need for expensive underlying mechanisms like memory copies between the CPU and GPU and paves the way for higher levels of CPU-GPU parallelism.

- I present the results of my second architectural study regarding the analysis and optimization of collaborative CPU-GPU applications and determine how to best optimize them for maximum collaborative performance. This work establishes a method for determining how to best optimize collaborative CPU-GPU applications and motivates future research regarding the development of profiling tools for heterogeneous CPU-GPU applications.

- I present the results of my third architectural study regarding the impacts of varying four key architectural parameters on collaborative CPU-GPU performance. My results provide new directions in heterogeneous CPU-GPU processor design and establish that computer architectural researchers should focus on increasing GPU compute unit coalesce size, GPU frequency, and lowering switching fabric latency in future heterogeneous CPU-GPU processors.

- To the best of my knowledge this dissertation presents the first in-depth simulation backed study of the collaborative CPU-GPU trade space that includes optimizing operating points

and analyzing the performance impact of key architectural parameters.

- Ready-made and fully working implementations of KnightSim, Parallel KnightSim, M2S-CGM, and the benchmarks presented in this dissertation are made available as free software and can be found on GitHub.

1.2 Dissertation Organization

This dissertation is organized as follows. Chapter 2 provides a background regarding discrete event-driven simulation methodologies then introduces and discusses the implementation and modeling methodologies of KnightSim and Parallel KnightSim. A detailed performance analysis is performed comparing KnightSim and Parallel KnightSim to that of three other mainstream discrete event-driven simulation methodologies. The work presented in chapter 2 extends my work previously presented in [17, 18]. Chapter 3 provides a background in heterogeneous CPU-GPU processors and applications and then introduces and discusses the implementation of M2S-CGM. Significant discussion is provided regarding M2S-CGM's x86 CPU, AMD Southern Islands GPU, and memory system simulation models. M2S-CGM is validated by comparison of M2S-CGM to a physical test system. The work presented in chapter 3 extends my work previously presented in [3]. Chapter 4 provides a background regarding the organizational use of the GPU in heterogeneous CPU-GPU applications and presents the results of three architectural studies. The architectural studies presented are based upon the culmination of all the work previously presented in this dissertation. The work presented in chapter 4 extends my work previously presented in [3, 19]. Chapter 5 covers the related work to my own. And finally, Chapter 6 draws conclusions to this dissertation and discusses new directions for future research.

CHAPTER 2: DISCRETE EVENT-DRIVEN SIMULATION METHODOLOGIES

This chapter discusses a novel context-based event-driven simulation methodology called KnightSim, as presented in [17, 18], and includes a significant expansion in discussion. KnightSim extends an older event-driven simulation library called "The Threads Package" by (1) incorporating corrections to functional issues that were introduced by the recent additions of stack protection, pointer mangling, and source fortification in the Linux software stack, (2) incorporating optimizations to the event engine, and (3) introducing a novel parallel implementation. The chapter starts by providing a sufficient background regarding event-driven simulation methodologies so that readers can understand the subject matter of the chapter. Then, a thorough discussion of the implementation methodologies of both KnightSim and Parallel KnightSim is given with expanded discussion regarding their usage in computer architectural functional and power modeling. The chapter concludes with a performance analysis and draws comparisons of both KnightSim and Parallel KnightSim to that of three other mainstream event-driven simulation methodologies in use today.

2.1 Background and Motivation

One of the most fundamental building blocks in any computer architectural simulation system is its event engine. A computer architectural simulation system's event engine provides the mechanism with which the simulator will carry out its simulation tasks. Additionally, the event engine introduces a temporal property in execution by allowing developers to specify a time when a given simulation task should take place. In the context of computer architectural simulation systems, the time when a simulation task begins is typically at the start of a desired simulated clock cycle.

At the time of writing this dissertation, most event engines used in mainstream computer architectural simulation systems utilize an implementation technique where events are registered at initialization and are provided a single callback function to execute. In essence, the event engine's scheduler will call the user provided function that is linked by the event when the number of developer specified cycles transpires. The event engine's scheduler maintains execution order by performing a heapify on a global event queue, or another equivalent approach, as new events are scheduled. When all events scheduled for a given cycle transpire, the cycle count is allowed to increment forward in time. In this approach, creating realistic computer architectural models of simulation elements requires the amalgamation of multiple events along with other primitives, like state flags, for execution control. Developers must carefully endeavor to model the latency, occupancy, and contention incurred by the modeled element.

A lesser known, but proven approach to event-driven simulation is an approach based on execution contexts. In this approach contexts are similar to the traditional events described above, but are more like micro kernels as they are individually executable on a given CPU core. In practical usage, each context represents a simulation element and encompasses all of the simulation element's functionality and data. Whole systems are then modeled as collections of contexts functionally working together. Context scheduling is explicitly handled between the contexts themselves with the use of an advance and await mechanism, which is further explained in this chapter. Ultimately, contexts are grouped logically, like the real hardware being modeled, and await advancement from a neighboring context. During execution, contexts can pause and assess a latency. When the context is paused no work will be performed by the context. This mechanism also results in automatically modeling occupancy and contention amongst contexts. The intrinsic properties of contexts providing a means to automatically model occupancy and contention makes the context-based approach an ideal approach for use in computer architectural simulation systems and serves as the motivation behind creating KnightSim and Parallel KnightSim.

2.2 KnightSim Implementation Methodology

The makeup of a KnightSim Context is shown in Alg. 2.1. The context is defined by functions that encapsulate all of the user's desired functionality and interfaces associated with the simulated element. During simulation execution, contexts await until they are notified that they should execute by an advance from one or more previously running contexts, but will not execute until they are ready. Contexts currently in the run state may pause any number of simulation cycles or await a future event to assess a latency. Simulation cycle time increases once all contexts have either entered a pause or await state.

Algorithm 2.1 A KnightSim Context

```
 1: procedure USER_FUNCTION(context* ctx)
 2:     long long i ← 1;
 3:     \\Other local variables here
 4:     loop
 5:         await(my_eventcount, i++, ctx);
 6:         \\Do work after being advanced
 7:         pause(1, ctx);                    ▷ Charges a latency for work performed
 8:         \\Finish doing work
 9:         advance(neighboring_eventcount, ctx);
10:         \\Clean up and return to await state
11:     end loop
12:     return
13: end procedure
```

The context's assessed latency during the paused or await state provides the mechanism to automatically model the occupancy of that context as no other work can be performed by the context during that time. Contention is automatically modeled as contexts must compete for modeled system resources. Individual contexts stall by pausing or awaiting as they wait for access to a particular resource. These additional stalls result in longer access latency for current and subsequent invocations as contexts wait for modeled hardware resources to become available. In the following subsections I provide discussion and present details regarding the implementation methodology of

KnightSim.

Algorithm 2.2 KnightSim Globals

```
1:  globals
2:      if wordsize == 64 then
3:          typedef long int _jmp_buf[8];
4:      else if x86_64 then
5:          typedef long long int _jmp_buf[8];
6:      else
7:          typedef int _jmp_buf[6];
8:      end if
9:
10:     typedef struct context{
11:     _jmp_buf buf;                          ▷ Buffer for CPU registers
12:     unsigned long long count;
13:     void (*start)(struct context*);        ▷ Context's function at execution
14:     char* stack;                           ▷ Context's unique stack
15:     int stacksize;
16:     struct context* next_ctx;              ▷ Context's batch list pointer
17:     } context;
18:
19:     typedef struct eventcount{
20:     struct context* next_ctx;
21:     unsigned long long count;
22:     } eventcount;
23: end globals
```

2.2.1 Events as KnightSim Contexts

KnightSim implements events as KnightSim Contexts, which are independently executed by the CPU at runtime. A context is represented by a struct that defines the context itself, and one or more eventcounts [20]. Pseudocode describing the implementation of contexts and eventcounts is shown in Alg. 2.2.

The context structure comprises a jump buffer, count, function pointer, stack pointer, stack size, and context pointer. The jump buffer is a primitive data type that is utilized by my hand implemented

`setjmp`() and `longjmp`() assembly functions, see the Appendix. My implementations of `setjmp`() and `longjmp`() correct functional issues introduced by the recent additions of stack protection, pointer mangling, and source fortification in the Linux software stack. Usage of the standard Libc `setjmp`() and `longjmp`() functions will render this methodology non-functional in modern Linux distributions. Determination of the correct data type and size of the context's jump buffer is shown at the top of Alg. 2.2. The context's count is used to synchronize the context with an eventcount's state. The context's function pointer is assigned the address of the user's provided entry function. The stack pointer points to an allocated region of memory of user provided stack size. Each context's stack is unique, resides in user memory space, and contains that context's execution data. Contexts execute in a shared memory space and can operate on global C/C++ objects as well. The context pointer is used to form a singly linked list that comprises a batch of contexts that are ready to run at a given cycle. When a context enters the pause or await state the next context in the list is executed until the list is empty.

Eventcounts are objects that provide a mechanism with which to determine if a context should be placed in the run or await state. Eventcounts comprise a count that is used as an incrementer and a pointer to a context that is awaiting an advance of the eventcount. The eventcount's count records the number of times the eventcount has been advanced. Contexts await the advance of eventcounts and when the counts of both an eventcount and context are equal the awaiting context runs. Typically, each context will have at least one unique eventcount assigned to it, but this is not required.

Context batches are stored via a hash table and are formed as each context enters the pause state. Contexts are added to the table by hashing the context's designated future execution cycle with the global hash table's number of rows minus one. The global hash table's number of rows is set as a power of two and must be large enough to ensure that pausing contexts form batches of only one future execution cycle. I find that a hash table size of 512 is more than sufficient to meet

this requirement. This is an optimized approach that maintains a high level of performance and does not require a modulus operation. Selecting the next context batch to run requires hashing the current global cycle count with the global hash table's number of rows minus one. A count of the number of unique context batches referenced by the hash table is kept. Simulation ends when the global hash table count is set to zero or the simulation's execution reaches a desired end point.

Algorithm 2.3 Eventcount Initialization

```
1: procedure EVENTCOUNT_INIT(void)
2:     eventcount* ec ← NULL;
3:     ec ← (eventcount*)malloc(sizeof(eventcount));
4:     ec->count ← 0;
5:     ec->next_ctx ← NULL;
6:     return ec;
7: end procedure
```

Algorithm 2.4 Context Initialization

```
1: procedure CTX_INIT((*func)(context*), int size)
2:     context* ctx ← NULL;
3:     ctx ← (context*)malloc(sizeof(context));
4:     ctx->count ← sim_cycle;
5:     ctx->stack ← (char*)malloc(size);
6:     ctx->stacksize ← size;                    ▷ Stack overflow check
7:     ctx->start ← func;                        ▷ User defined function
8:     ctx->buf[ip] ← context_start();
9:     ctx->buf[sp] ← stack_top_ptr;
10:    ctx->next_ctx ← NULL;
11:    ctx_hash_insert(ctx, ctx->count&ROWS);
12:    return
13: end procedure
```

2.2.2 Initialization

Prior to simulation execution each user created eventcount and context is initialized. Eventcount initialization is straightforward and comprises the allocation of the eventcount with the use of

malloc(), the initialization of the eventcount's count to zero, and the initialization of the eventcount's context pointer to NULL, as shown in Algo. 2.3. Context initialization is shown in Alg. 2.4 and comprises the allocation of the context itself with the use of malloc(), initialization of the context's count, allocation of the context's stack with use of malloc(), assignment of the stack size, assignment of the user's provided entry function, manipulation of the instruction and stack pointers in the context's jump buffer, initialization of the context's context pointer to NULL, and finally insertion of the context itself into the applicable context batch. The entry function embodies the functionality of the element this context will simulate, as shown in Alg. 2.1. During initialization the context's count is assigned the global cycle count. Contexts may be created and destroyed at any time, before and during simulation execution.

The context's jump buffer is uninitialized after being created. Thus, I assign a starting instruction pointer and stack pointer by hand to give the context our desired starting position and unique stack memory. This manual configuration of the context's jump buffer is what makes each context independently executable. I ignore other CPU registers at initialization because they will be obtained the first time setjmp() is called. Additionally, I push a pointer to the context onto the context's stack for retrieval later. This allows us to resolve information about the context after the context's initial jump.

The pseudocode shows an instruction pointer assignment as the head of a context_start() function. On initial execution, each context will first jump to the head of this function and then retrieve the pointer to itself. The context's start() function is then called and passed a pointer to the context itself for future access. Program execution is now placed at the head of the user's provided entry function with resolution of the assigned context, see Alg. 2.1. Additionally, the pseudocode shows a stack pointer assignment as the top of the allocated stack that is calculated as shown in

Equ. 2.1 for both 32bit and 64bit Linux environments.

$$stack\_top\_ptr = stack\_ptr + stack\_size - sizeof(\textbf{int}) \tag{2.1}$$

The assignment of the instruction and stack pointers to the context's jump buffer is architecture dependent and must be accounted for at time of compilation. The instruction pointer and stack pointers are assigned to jump buffer positions *five* and *four* in the 32bit Linux x86 environment and are assigned to jump buffer positions *seven* and *six* in the 64bit Linux x86 environment.

2.2.3 Scheduling

Pseudocode showing the mechanisms responsible for providing KnightSim Context scheduling is shown in Alg. 2.5, Alg. 2.6, Alg. 2.7, and Alg. 2.8.

Algorithm 2.5 Context Scheduling: Advance

```
1: procedure ADVANCE(eventcount* ec, context* ctx)
2:     ec->count++;
3:     if ec->next_ctx and ec->next_ctx->count == ec->count then
4:         ec->next_ctx->next_ctx ← ctx->next_ctx;
5:         ctx->next_ctx ← ec->next_ctx;
6:         ec->next_ctx ← NULL;
7:     end if
8:     return
9: end procedure
```

Placing KnightSim in the simulation state simply requires obtaining a pointer to the first context in the initial context batch and performing a `longjmp()` to the context's starting position. Subsequently, each context resides in either an await, ready to run, or running state until the end of simulation. In the single-threaded version of KnightSim only one context is ever in the running state at a time. A transition between these states is accomplished with use of the `advance()`,

`await`(), and `pause`() functions. A running context executes its assigned tasks and advances one or more eventcounts as a product of its work by use of the `advance`() function. By advancing an eventcount, the designated eventcount's count is incremented and the eventcount's context pointer is checked. If the counts of both the context and eventcount are equal the context is removed from the eventcount and inserted next into the current context batch as a context that should run this cycle.

Algorithm 2.6 Context Scheduling: Await

```
 1: procedure AWAIT(eventcount* ec, count value, context* ctx)
 2:     if ec->count >= value then
 3:         return;
 4:     end if
 5:     ctx->count ← value;
 6:     ec->next_ctx ← ctx;
 7:     ctx ← ctx->next_ctx;
 8:     if !setjmp(ec->next_ctx->buf) then
 9:         if ctx then
10:             longjmp(ctx->buf);
11:         else
12:             sim_cycle++;
13:             longjmp(context_select());
14:         end if
15:     end if
16:     return
17: end procedure
```

After a context completes its tasks, the context then transitions to the await state by use of the `await`() function. The context will assign itself a count on which it will await, remove itself from the current context batch, assign itself to the designated eventcount's context pointer, and store the current position in its jump buffer. Simulation execution can then jump to the next context in the current context batch or, if this batch is finished, increment the global cycle count and select the next batch. A running context may also assess a latency with the use of the `pause`() function. Assessing a latency stops the current context from running until a future global cycle count is reached, where the context will then automatically resume execution. The pausing context

is removed from the current context batch and added to a context batch in the global hash table that is awaiting the same future global cycle count. If the addition to the global hash table results in a new context batch record the global hash table's count is incremented. Lastly, I store the current position in the pausing context's jump buffer. Simulation execution can then jump to the next context in the current context batch or, if this batch is finished, increment the global cycle count and select the next context batch.

Algorithm 2.7 Context Scheduling: Pause

```
 1: procedure PAUSE(count value, context* ctx)
 2:     value ← value+sim_cycle;
 3:     context* ctx_ptr ← ctx;
 4:     ctx ← ctx->next_ctx;
 5:     ctx_hash_insert(ctx_ptr, value&ROWS);
 6:     if !setjmp(ctx_ptr->buf) then
 7:         if ctx then
 8:             longjmp(ctx->buf);
 9:         else
10:             sim_cycle++;
11:             longjmp(context_select());
12:         end if
13:     end if
14:     return
15: end procedure
```

Algorithm 2.8 Context Scheduling: Context Select

```
 1: procedure CONTEXT_SELECT(void)
 2:     context* ctx_ptr ← NULL;
 3:     if table_count then
 4:         do
 5:             ctx_ptr ← table[sim_cycle&ROWS];
 6:         while !ctx_ptr and sim_cycle++;
 7:         table[sim_cycle&ROWS] ← NULL;
 8:         table_count-;
 9:     else
10:         sim_end();
11:     end if
12:     return ctx_ptr->buf;
13: end procedure
```

The next context batch is selected with the `context_select()` function. I select the next context batch by iterating through the global hash table until I obtain a valid pointer to a batch of contexts. The global cycle count is incremented with each required iteration and reference of the hash table. Each removal of a context batch from the hash table results in a decrement of the global hash table's count. As mentioned before, simulation ends when the global hash table's count reaches zero.

2.3 KnightSim Modeling Methodology

Using KnightSim to create computer architectural simulation models, like the one shown in Fig. 2.1, is straightforward. The figure shows a functional architecture block diagram of a modern processor comprising 18 CPU cores, a mesh switching fabric, and two memory controllers. To implement this in KnightSim the developer would establish the appropriate number of contexts necessary to model the desired system and assign each context the appropriate generic control function. The control function encapsulates the tasks that each particular type of simulation element performs at the developers desired level of granularity. For example, emulation elements, CPU pipeline stages, caches, IO controllers, switches, memory controllers, DRAM, and etc. The computer architectural model, in whole, is represented as the collection of these simulation elements cooperatively working together, like the real hardware.

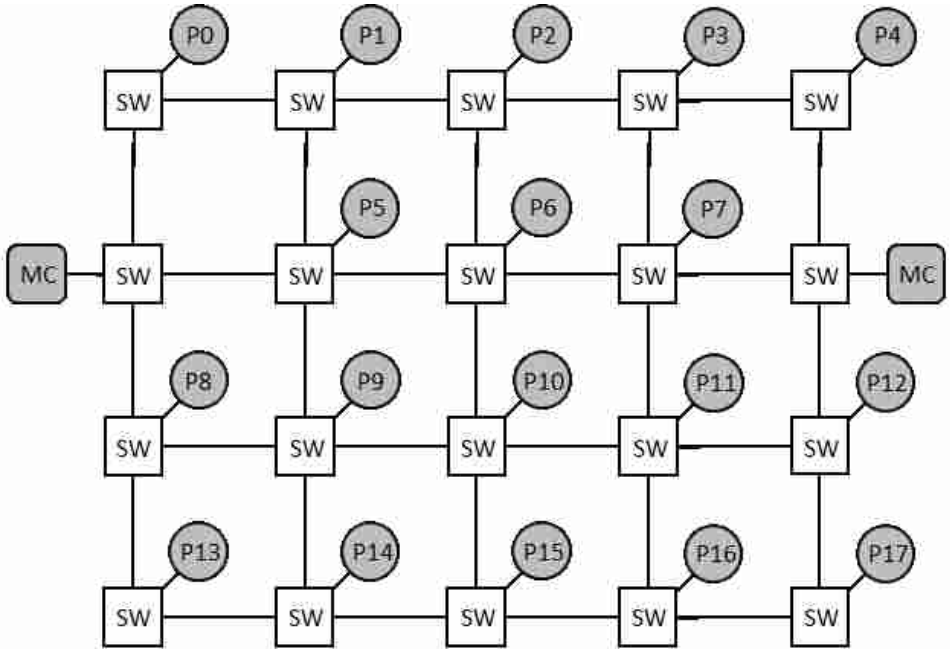


Figure 2.1: An 18 Core CPU Computer Architectural Model

In this example each processor core, cache, switch, and memory controller could be modeled as individual contexts that execute at the appropriate cycles. Alternatively, a more fine-grained approach would be to model each processor core, cache, switch, and memory controller as a collection of contexts that comprise the sub-elements of the modeled component. In all cases the desired level of simulation granularity is left to the developer to decide. In the presented approach, each modeled CPU core would fetch and then emulate a target instruction, then update the state of its modeled pipeline stages and associated resources. CPU flow control is provided based on the state of the CPU's modeled pipeline resources and the memory system. Interaction with the first and subsequent levels of the memory system is accomplished with an appropriate advancement and memory system element load or store. This process continues throughout the memory system and switching fabric. In a large computer architectural simulation model, long chains of contexts represent the interdependence between discrete hardware elements. During long stalls contexts

await and simulation execution time advances to the appropriate simulation cycle.

In the following section I present and discuss a simulation model of the switching fabric shown in Fig. 2.1. The example switching fabric model demonstrates advanced usage of KnightSim for modeling of complex system elements and highlights the ease of use and power of KnightSim's modeling methodology. In addition to the example switching fabric shown here, researchers can refer to CGM [3], where an entire memory system comprising configurable cache structures, cache directories, translation lookaside buffers, page table walkers, switching fabrics, crossbars, a system agent, memory controller, and other discrete system elements such as a GPU hub and IOMMU is modeled using the techniques presented in this dissertation.

2.3.1 Switching Fabric Implementation Methodology

Alg. 2.9 and Alg. 2.10 provides pseudocode showing the implementation of the mesh switching fabric illustrated in Fig. 2.1. The example demonstrates how to model a switch and crossbar with a user-defined number of ports and virtual lanes. The switch pseudocode is written with the intent of clearly showing KnightSim's modeling methodology. More fine-grained and sophisticated implementations are possible. The model consists of a single context and its main control function, `switch_ctrl()`, where all derivative switch-related tasks are encapsulated.

Algorithm 2.9 Switch Control Globals

```
1: globals
2:     #define CYCLE etime->count»1
3:     #define P_PAUSE(p_delay) pause((p_delay)«1)
4:     #define ENTER_SUB_CLOCK if(!(etime->count & 0x1)) pause(1);
5:     #define EXIT_SUB_CLOCK if (etime->count & 0x1) pause(1);
6:     int switch_pid = 0;
7:     eventcount ** sw_ec;
8:     struct switch_t ** sw;
9: end globals
```

Variables pertinent to the individual switch model are located along the top of the main `switch_ctrl()` function. Context variables are all initialized and stored within each individual contexts. Any number of switches can be created and chained together utilizing this single `switch_ctrl()` function. Therefore, each switch is assigned a global `my_pid` for dereferencing the correct switch structure and eventcount during execution. The `await()` function takes as an argument a count to assign to the context as it waits. So, a context await variable is needed to manage this interaction and is initialized with the value of one. This causes the switch to enter the await state at the beginning of the switch's main execution loop where it then awaits advancement by one or more connected system elements, usually a connected I/O controller.

Arguably, one of the most powerful features of KnightSim is the ability to easily model interactions in the sub-clock domain. In relation to computer architectural simulations, this provides an easy way to model arbitration in the system. After advancement by one or more connected elements the switch then enters the sub-clock domain. During which all elements seeking to advance the switch have completed their advancement and have entered the await or pause state. The switch can then perform arbitration and scheduling functions with complete knowledge of all its advancing elements. In modeling the sub-clock domain, the global cycle count can be divided any number of times with the use of macros intended to adjust the cycle time. As shown at the top of Alg. 2.9, `ENTER_SUB_CLOCK` and `EXIT_SUB_CLOCK` adjust the cycle count such that two cycles represent one cycle for the system at large. The sub-clock domain is then modeled on the odd cycle count which is the half cycle to the system.

After returning to the regular clock domain with use of `EXIT_SUB_CLOCK`, the switch then charges a latency for its work with the `P_PAUSE` macro. The switch will pause and then resume from this point after the specified number of cycles passes. The charged latency provides the mechanism to automatically model the occupancy of the switch. Upon resuming, the switch then records the number of successfully formed links, moves data from the specified input ports to the specified

output ports, and advances the specified output port I/O controller. If a specified output port is full the switch can not successfully form the link and will automatically retry each cycle until success. The switch's retry stalls form a part of the system wide contention model regarding these resources.

Algorithm 2.10 Switch Control

```
 1: procedure SWITCH_CTRL(context* ctx)
 2:     int my_pid = switch_pid++;
 3:     count step = 1;                                   ▷ Context await count
 4:     packet* net_packet = NULL;
 5:     loop
 6:         await(&sw_ec[my_pid], step, ctx);
 7:         calc_passive_power(&sw[my_pid], CYCLE);
 8:         ENTER_SUB_CLOCK
 9:         xbar_link(&sw[my_pid]);
10:         EXIT_SUB_CLOCK
11:         P_PAUSE(&sw[my_pid]->latency, ctx);
12:         for i ← 0 to sw[my_pid]->num_ports − 1 do
13:             if xbar_link_success(&sw[my_pid], i) then
14:                 sw[my_pid]->num_links++;
15:                 net_packet ← dequeue(&sw[my_pid], i);
16:                 enqueue(&sw[my_pid], i, net_packet);
17:                 advance(&sw_io_ec[xbar_out(&sw[my_pid], i, ctx)]);
18:             end if
19:         end for
20:         step += sw[my_pid]->num_links;
21:         calc_active_power(&sw[my_pid], &sw[my_pid]->num_links, CYCLE);
22:         sw[my_pid]->num_links = 0;
23:         switch_update_state(&sw[my_pid]);
24:     end loop
25:     return;                                           ▷ Should never return
26: end procedure
```

Prior to returning to the top of the switch's main execution loop the switch resets its state and prepares for the arrival of new work. The switch's step value must be incremented via the number of links made. This accounts for the number of advancements and the work performed that cycle. In the case that the switch fails to service a request, the main execution loop will continue to run until all outstanding requests are serviced and the step variable's value returns to one larger than

the switch's eventcount value.

2.3.2 Power Simulation Methodology

The methodology behind creating accurate power simulation models is also shown in the example switch model. This is accomplished by calculating the individual element's passive and active power expenditures throughout execution. Each element has a finite number of possible derivative states it can enter. In the modeled switch example, individual switch elements are either in the run/ready state or are in the await state. Calculations determining the expenditure of power during these two states would be performed with the equivalent `calc_passive_power`() and `calc_active_power`() functions as shown.

The await state is straightforward and represents the period of time the element passively utilizes or leaks power. After being advanced the switch wakes up and calculates the passive or leaked power usage over the period of cycles the switch was awaiting. This is accomplished by utilizing a variable value representing the passive power used by the element per cycle. Upon exiting simulation one final calculation must be made to reconcile the difference between the element's last await and the end of simulation.

During the run/ready state several derivative outcomes may occur that would influence the amount of power utilized by the element. This necessitates a user-provided state-based power profile for the modeled element. The power profile maps each possible outcome to an assigned value representing the active power used in that state. The nature of the switch's main execution loop makes it easy to determine the possible states of the switch model during the run/ready state. In the example, the number of links formed while in the run/ready state can be used to estimate the level of effort performed by the switch.

When added together the passive and active power usage determines the overall power expenditure for the modeled element. The expenditure for each modeled element can be aggregated to determine the power expenditure for the entire system or portions of the system thereof. Impacts to power from proposed architectural changes can then be accurately modeled and taken into account by researchers.

2.4 Parallel KnightSim Implementation Methodology

I developed KnightSim with an eye towards ultimately parallelizing it. Therefore, parallelizing KnightSim only requires a few changes which I highlight in this section. In general, the approach to parallelizing KnightSim is summarized best as splitting a given cycle's context batch into a balanced group of smaller context batches and then executing the group of context batches over an appropriate number of threads. This results in a discrete event-driven simulation methodology that automatically parallelizes event execution at the cycle level.

Algorithm 2.11 Thread Context Select

```
 1: procedure THREAD_CONTEXT_SELECT(int id)
 2:     if table_count then
 3:         context* ctx_ptr ← table[sim_cycle&ROWS][id];     ▷ Check for thread's context batch
 4:         if ctx_ptr then
 5:             table[sim_cycle&ROWS][id] ← NULL;
 6:             __sync_sub_and_fetch(&table_count, 1);
 7:             longjmp(ctx_ptr->buf);                  ▷ Jump to first context's last position
 8:         else                                                    ▷ Return to cycle barrier
 9:             longjmp(threads[pthread_self()%NUMTHREADS].buf);
10:         end if
11:     end if
12:     return
13: end procedure
```

Parallel KnightSim utilizes a pool of POSIX threads of configurable size and a 2D global hash table. The 2D global hash table's rows each represent a future cycle and each thread is assigned a

column. As shown in Algo. 2.11, at the start of each cycle each thread consults the appropriate row and column of the 2D global hash table to determine if there is a context batch to run this cycle or not. If a thread finds a context batch it removes the context batch from the 2D global hash table and then independently executes each context in its assigned context batch one after the other until the end of the context batch list is reached.

Threads maintain global cycle synchronization by use of a global cycle barrier, as shown in Algo. 2.12. After each thread completes the execution of its assigned context batch each thread will then return and enter the global cycle barrier. All returning threads but the last to arrive at the global cycle barrier spin while waiting for the global cycle count to be incremented. The last thread to arrive performs the global cycle count increment, which releases all threads and places overall execution in the next cycle.

Algorithm 2.12 Thread Control

```
1:  procedure THREAD_START(void* id)
2:      volatile bool lflag ← false;
3:      thread_set_affinity((long)id);
4:      while(gflag != lflag){};                          ▷ Wait for sim execution
5:      setjmp(thread_buf[pthread_self() mod SIZE]);
6:      lflag ← !lflag;                                   ▷ Invert the local status flag
7:      if __sync_sub_and_fetch(&threadnum, 1) then
8:          while(gflag != lflag){};                      ▷ wait for the last thread to arrive
9:      else
10:         sim_cycle++;
11:         threadnum ← SIZE;
12:         gflag ← lflag;                                ▷ Last thread resets the flags
13:     end if
14:     context_select((long)id);          ▷ Contexts select and run their respective batches
15:     return
16: end procedure
```

2.5 Parallel KnightSim Modeling Methodology

Using Parallel KnightSim for the creation of parallelized computer architectural models starts out in the same manner as described in Sec. 2.3. Individual system simulation elements are modeled as regular KnightSim contexts. The developer then specifies the number of threads he or she wishes to utilize during simulation execution and assigns the contexts to individual threads. A Parallel KnightSim execution with a single thread specified is an equivalent execution to the non-parallelized version of KnightSim. However, Parallel KnightSim incurs additional overhead due to the introduction of thread management that is not present in the non-parallelized version of KnightSim. Therefore, it is recommended that the parallelized version of KnightSim not be used for sequential (single-threaded) executions.

An optimized approach to parallelizing an 8 CPU core simulation model with accompanying switching fabric and L1, L2, and L3 caches is shown in Fig. 2.2. In the figure, each box represents a single computer architectural simulation element that can be simulated by a single KnightSim context type as discussed in Sec. 2.3. The developer then divides up the contexts and assigns them to a target thread as illustrated by "Thread 1" and "Thread 2". This is done by assigning a thread ID to each context at initialization time thereby splitting the contexts into a context batch for thread 1 and a context batch for thread 2. This process can be done for any number of threads, however performance gains are dependent on producing an optimized thread to context batch ratio.

KnightSim Contexts form chains as they advance neighboring context's eventcounts and modify shared data structures. In the scope of a single thread only one context is ever running at a time, so, a thread's execution of its context batch can be treated as sequential execution. This means that thread safety is not of concern to a thread's local context batch. However, contexts that share data or eventcounts between two or more threads can run into classic multithreaded programming data hazards. The following subsection discuss ways to workaround these issues.

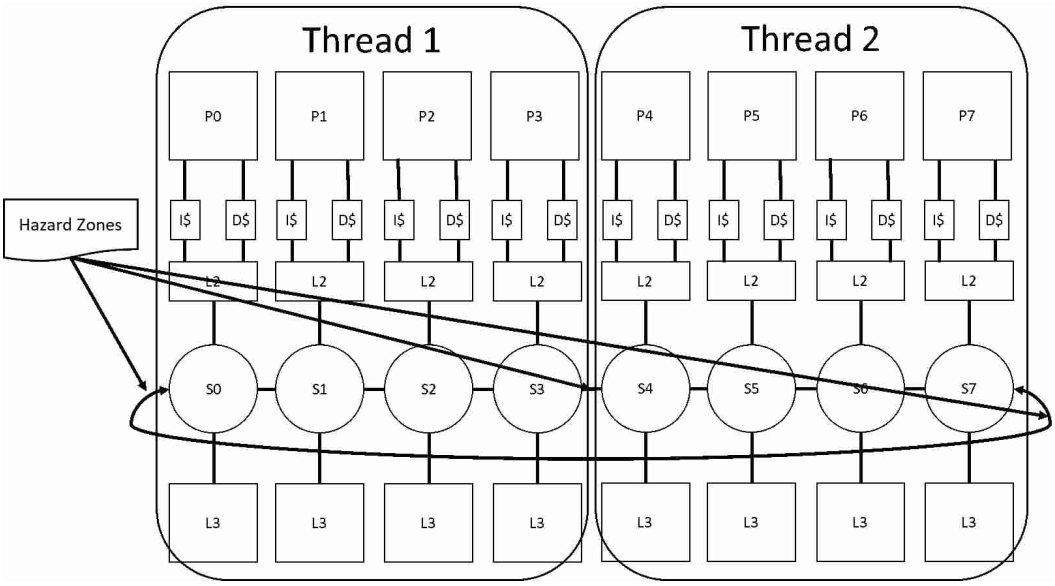


Figure 2.2: Parallel KnightSim Hazard Zones

2.5.1 Data Hazards

As discussed in Sec. 2.4, Parallel KnightSim automatically parallelizes event execution at the cycle level. However, data race conditions can occur between contexts that share data and that are being executed by two or more threads in a given cycle. Fig. 2.2 depicts where data races can occur and labels them as hazard zones. In this example the two threads share and modify data (e.g. eventcounts and input/output queues) related to the switches along the edge of each thread's context boundary as a result of one thread's switch advancing another thread's switch. Parallel KnightSim automatically accounts for internal thread safety issues by appropriately handling the advance and await of hazardous eventcounts. Developers are only required to specify at initialization time whether or not a particular context and its eventcount(s) are hazardous. The developer then only needs to be concerned about thread safety on the simulator side. For example, a modeled message queue that is shared between two contexts executed by two different threads could pose a thread safety issue.

On the simulation side, data race conditions can be avoided by assigning contexts that share data to the same context batch so that they are run by the same thread during execution. In places where a natural division of the contexts is not possible, the inclusion of fine-grained thread-safe techniques, such as mutexes and lock-free data structures, in only the hazard contexts will eliminate data race conditions. Another simple approach to avoid data hazards is to have paired hazardous contexts wait for execution on different sub-clocks. This approach guarantees that they do not execute at the same time and thus will not interfere with each other. Ultimately, optimized parallel simulation performance is gained by balancing the simulation model's size and simulated architectural structure with a properly specified number of threads and context-to-thread assignment. Developers should endeavor to reduce the number of serialization points in the architectural model, which will lead to better parallel performance gains.

2.6 KnightSim and Parallel KnightSim Performance Results

For performance evaluations, direct performance comparisons between KnightSim, Parallel KnightSim, and the discrete event-driven simulation engines found in Gem5 [21], Multi2Sim [15], and M2S-CGM [3] are made. Comparisons are made to this selection of discrete event-driven simulation engines because, at the time of authoring this dissertation, the simulators in which they are used are relevant, widely recognized, and have been used in recent computer architectural simulation related publications. For the purposes of these experiments, the discrete event-driven simulation engines found in Gem5, Multi2Sim, and M2S-CGM are referred to as Gem5-Event, Esim, and The Threads Package respectively. In the results, KnightSim and Parallel KnightSim are referred to as KS and PKS_N respectively. For PKS, the "_N" denotes the number of specified threads used in each PKS trial.

2.6.1 Experimental Setup

All experiments are conducted on a test system comprising a 16 core Intel Xeon E5-2697A v4 processor running at 2.6 GHz - 3.6 GHz with ample system memory running at 2400 MHz. In all test cases execution time is measured over the equivalent `simulate`() function. Measured execution times do not include time spent in regions of code associated with setup, initialization, or cleanup activities. Additionally, any non-essential code, like asserts, from each test application has been removed.

Gem5-Event and Esim employ a similar implementation approach that establishes an event list with associated callback functions upon initialization. During execution, events are scheduled to run in either the current cycle or a future cycle using an equivalent `schedule_event`() function. Scheduled events are placed in a data structure and removed for execution at a later simulation cycle. Gem5-Event declares class objects as sim objects whose member functions can be declared as events. Therefore, I implement an event in Gem5-Event as a single class member function that is initially scheduled to run by the class's constructor during initialization time. Esim declares domain event handlers that are meant to handle a number of domain specific sub events. Thus, I implement an event in Esim as a single domain level event that is registered and scheduled to run at initialization time. For both Gem5-Event and Esim, each time an event is executed the event schedules itself to run again in one cycle.

KnightSim, Parallel KnightSim, and The Threads Package implement events as contexts, however the implementation of The Threads Package is completely different from that of KnightSim and Parallel KnightSim and does not benefit from the extensions presented in this dissertation, see Sec. 2.2 and Sec. 2.4. As discussed and shown in Sec. 2.2.3, scheduling a pause of one cycle during context execution is functionally identical to scheduling an event to occur one cycle later, as in Gem5-Event and Esim. For KnightSim, Parallel KnightSim, and The Threads Package an event

is implemented as a single context that is registered and scheduled to run at initialization time. Each time the context is run, the context schedules itself to run again after one cycle by pausing one cycle. Before direct performance comparisons can be made developing an understanding of the use of event engines in computer architectural simulators is needed.

2.6.2 Experiment 1: Determining Event Engine Usage

To support the performance analysis of KnightSim, a determination of how a typical computer architectural simulator makes use of its event engine must be made. This is accomplished by determining what a realistic range is in terms of (1) the average and maximum number of executed events per simulated cycle and (2) the average and maximum number of physical cycles per event. These two measurements give us a sense of the amount of pressure placed on the event engine and how many physical cycles it takes to process an event on average. To measure these values, a sampling of the Rodinia OpenMP benchmarks [22] has been run on M2S-CGM. Measurements are taken over the benchmark's parallel section while varying the size of the simulated system. M2S-CGM provides a system wide model with a configurable number of CPU cores, L1, L2, and L3 caches, switching network, system agent, memory controller, and SDRAM.

The findings are shown in Table 2.1 and are used to form the basis of the performance analysis. As shown in the table a realistic range of expected events per simulated cycle is approximately 13, or fewer, for small simulation models to approximately 980, or more, for large simulation models. From the results, it is apparent that the predominance of computer architectural simulation models used in relevant research would fall in the category of approximately 113 events, or fewer, per simulated cycle. This is because most relevant research simulation models have had 16 or fewer CPU cores.

Table 2.1: Measure of Event Engine Usage

Sim Cores	Avg Events	Max Events
1	13	26
2	20	38
4	34	58
8	60	95
16	113	165
32	215	291
64	416	529
128	813	980

Physical Cycles	Avg Cycles	Max Cycles
	2680	9945

From the results it is also established that a typical computer architectural simulator utilizing a context-based event engine performs an average of 2680 physical cycles of work per event. However, it has also been observed that in some cases this can be considerably higher. The data gathered regarding the average number of physical cycles of work per event can be utilized to determine if the usage of Parallel KnightSim for the purposes of parallelizing computer architectural simulations is viable or not. It is important to mention that the information set forth in Table 2.1 is intended to give us some reasonable guidelines with which to design, conduct, and draw conclusions to results for the performance analysis. Other simulators may exhibit slightly different results, but it is not expected to be significantly different.

2.6.3 Experiment 2: Single-Threaded Performance Results

This experiment is designed to gauge the overall single threaded performance difference between KnightSim, The Threads Package, Esim, and Gem5-Event. Overall speedup measurements are taken for eight test cases. The test cases comprise the execution of 16, 32, 64, 128, 256, 512, 768, and 1024 events per cycle for one million cycles. The selected test cases provide a good

range in terms of varied simulated system size, as established in Sec. 2.6.2. The selected test cases also facilitate performance analysis comparisons with Parallel KnightSim executions in the following section. A verification that all test applications function correctly was performed by observing the final value of a global variable that was incremented by each event during execution. In the formally measured experiment, the verification feature was commented out so that all events perform no work. After several trials, It was determined that one million simulated cycles is more than sufficient to reach a steady state for final performance measurements.

The experimental results for the single threaded executions are shown in Fig. 2.3. The results for each test case are normalized to the execution results of The Threads Package. KnightSim demonstrated strong overall performance with an average speedup of 3.51x, 11.95x, and 2.8x over the execution results of The Threads Package, Esim, and Gem5-Event respectively. However, more importantly KnightSim showed an average speedup of 4x over both The Threads Package and Gem5-Event in the range of 16 to 128 events per cycle. This range represents a preponderance of the use cases for event engines in computer architectural simulation based research.

The results for KnightSim show that the extensions to The Threads Package have provided a significant boost in the methodology's overall performance and that the methodology also outperforms those of Gem5-Event and Esim. Gem5-Event proved to scale very well with larger simulated system sizes, but did not outperform KnightSim over the selected test case range. Esim provided a consistent performance baseline, but did not exceed the performance of KnightSim, Gem5-Event, and The Threads Package. An inspection of Esim's source code revealed that Esim performs a `calloc`() with the scheduling of each event and a `free`() at the end of each event's execution. This is the leading cause of the performance disparity between Esim and the other event engines.

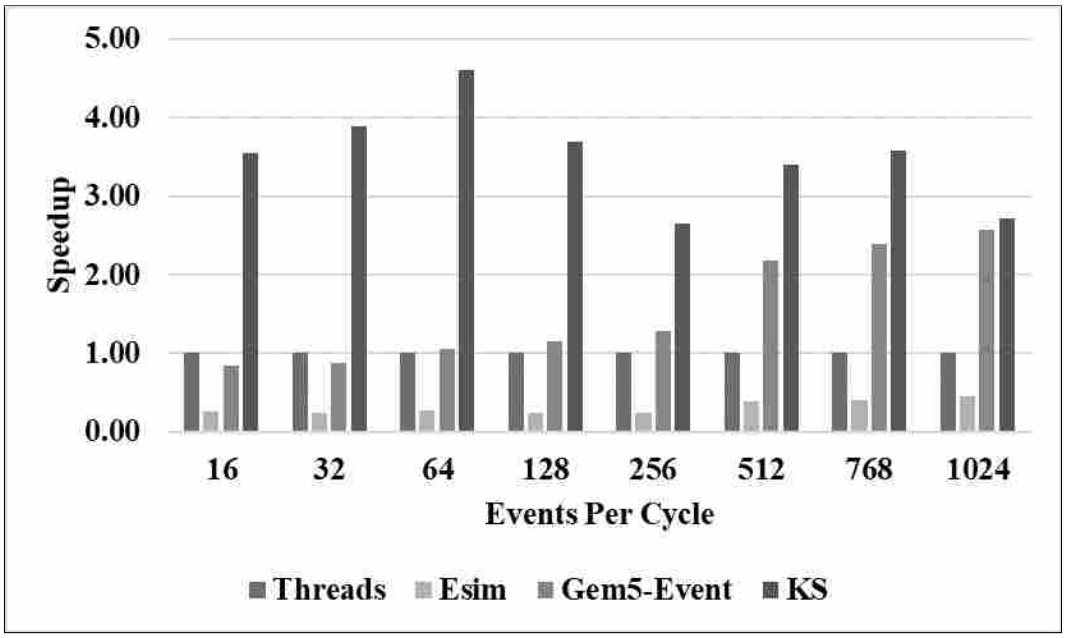


Figure 2.3: Single Threaded Performance Results

2.6.4 Experiment 3: Multithreaded Performance Results

The next experiment is designed to gauge the overall multithreaded performance difference between KnightSim and Parallel KnightSim. Before continuing on it should be noted that none of the previously presented event engines possess a parallel processing capability, therefore this experiment draws comparisons to KnightSim as the established performance baseline.

Drawing on the experiences gained by implementing Parallel KnightSim and conducting initial trials, it is apparent that the two most critical factors impacting overall parallel performance in computer architectural simulations are (1) the number of events executed per cycle and (2) the amount of work each event performs when executing. These two factors inversely affect the negative impacts to parallel performance imposed by the global cycle barrier and other pthread related overhead. Parallel speedups are achievable once the combined effects of these two factors amortize the cost of the serial section.

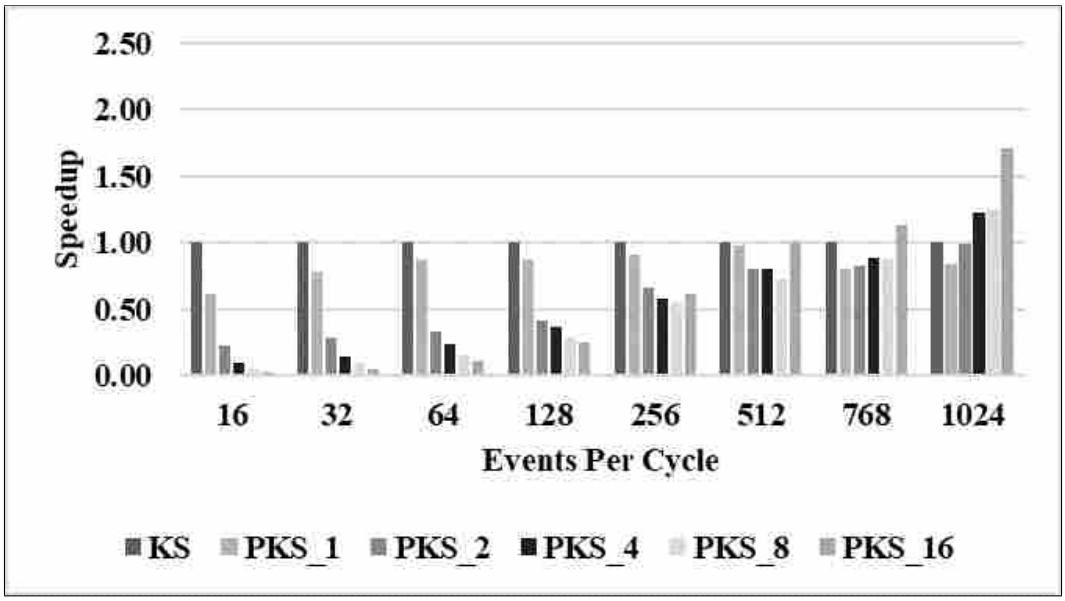


Figure 2.4: Multithreaded Performance Results Without Work

Experimental results regarding the impact of the number of events per cycle on overall parallel performance are shown in Fig. 2.4. For this experiment the configuration described in Sec. 2.6.3 is maintained, however the workload is now parallelized using Parallel KnightSim. When events perform no work the cost of the global cycle barrier and other pthread related overhead is apparent in the results. It is observed that Parallel KnightSim does not out perform KnightSim for small computer architectural models executing fewer than 512 events per cycle. However, increasing the number of events per cycle begins to amortize the cost of the serial section when executing approximately 256 events per simulated cycle in parallel. At approximately 512 events per simulated cycle, and higher, parallelization results in measurable speedups. These results show promise for parallelizing large computer architectural models with Parallel KnightSim because PKS can scale with computer architectural simulation size. However, it is unreasonable to base performance results solely on an experiment where each event performs no work. In the following experiment we provide each event with a simulated work load and rerun the experiments.

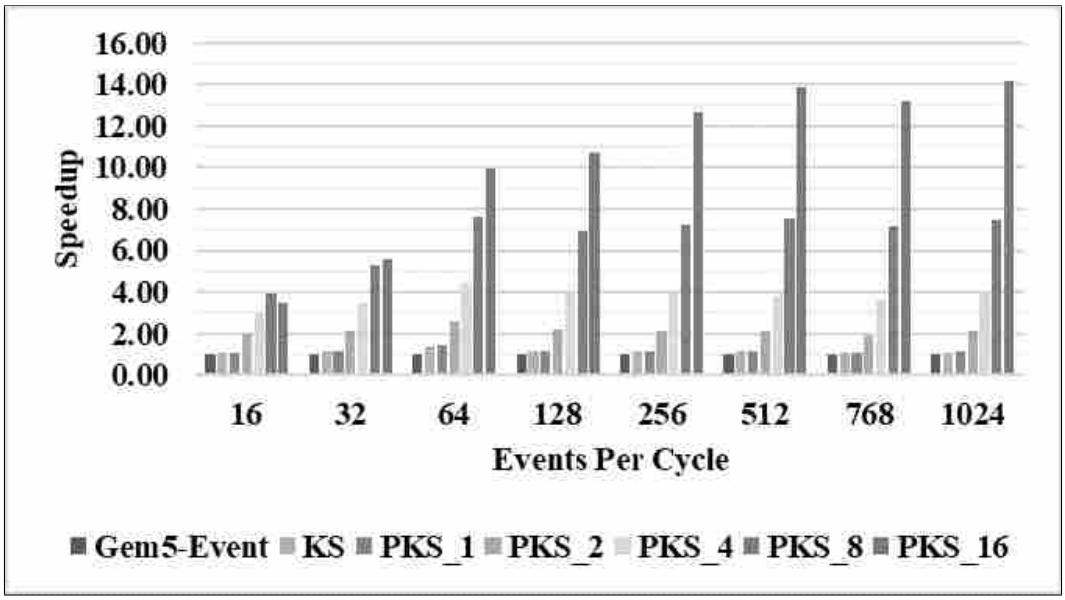


Figure 2.5: Multithreaded Performance Results With Work

The appropriate amount of work to simulate is determined by scaling the average workload each event must perform during execution until the cost of Parallel KnightSim's serial section is amortized. This is achieved when the overall performance of both KnightSim and single threaded Parallel KnightSim are equal. The results of our trials showed that, on average, events must work for approximately 1700 physical cycles to amortize the cost of the serial section in Parallel KnightSim. This is in conjunction with the number of events being executed per cycle. Fig. 2.5 shows the impact of the two factors combined. Again, the configuration described in Sec. 2.6.3 is maintained, however a workload of approximately 1700 cycles is to the events. Results for Gem5-Event, KS, and PKS are shown with all results normalized to those of Gem5-Event. For brevity I do not show results for Esim and The Threads Package because they do not outperform the results of Gem5-Event. The results show that for each test case the cost of the serial section in PKS is amortized as both KS and PKS_1 have the same overall performance. KS and PKS_1 outperform Gem5-Event with an average speed up of 1.16x, however it is apparent that as event workload scales up the ben-

efits of fast single-threaded event execution declines. This is due to the limited scalability found in all single-threaded event engines and strongly motivates parallelization with PKS.

Parallelizing the workload with PKS resulted in average speedups of 2.15x, 3.79x, 6.65x, and 10.44x over Gem5-Event and 1.89x, 3.33x, 5.84x, and 9.24x over KS for PKS_2, PKS_4, PKS_8, and PKS_16 respectively. Considering that PKS does not need to execute an unrealistically high number of events per cycle to gain parallel performance and that the imposed 1700 physical cycle threshold is lower than the number of physical cycles measured in Table 2.1, it is evident that utilizing PKS to parallelize small to large computer architectural simulations is viable and can result in measurable speedups over the established performance of other discrete event-driven simulation methodologies.

2.7 Summary and Conclusions

In the first half of this chapter I introduce KnightSim and discuss the benefits of KnightSim's event implementation approach regarding how KnightSim Contexts automatically model simulated occupancy and contention. I then provide detail regarding KnightSim's implementation methodology and discuss critical items pertaining to how KnightSim is used and how events are instantiated as KnightSim contexts. Then, I discuss the implementation methodology of Parallel KnightSim. In the second half of this chapter I reported the results of a detailed performance analysis of discrete event-driven simulation engines in computer architectural simulators and the results of a direct comparison between KnightSim, Parallel KnightSim, and the discrete event-driven simulation engines found in Gem5, Esim, and M2S-CGM. My study provides insight into the average number of events executed per simulated cycle and average number of physical cycles it takes to process an event in a typical computer architectural simulator. I establish that small simulation models execute approximately 13 events per simulated cycle, or fewer, and that large simulation models

execute approximately 980 events per cycle, or more. My overall performance results showed that on average KnightSim achieves speedups of 2.8x to 11.9x over the other discrete event-driven simulation engines presented in this paper. The results also show that, on average, Parallel KnightSim can achieve speedups over KnightSim of 1.78x, 3.30x, 5.84x, and 9.16x in 2, 4, 8, and 16 threaded executions respectively. Based on the performance results presented here and on the additional benefits of KnightSim's context-based approach I believe that KnightSim is a promising tool for use in the development of future computer architectural simulations. The next chapter highlights a new CPU-GPU heterogeneous architectural simulator called M2S-CGM. KnightSim was pervasively used in the creation of M2S-CGM and forms the basis of all of M2S-CGM's timing models.

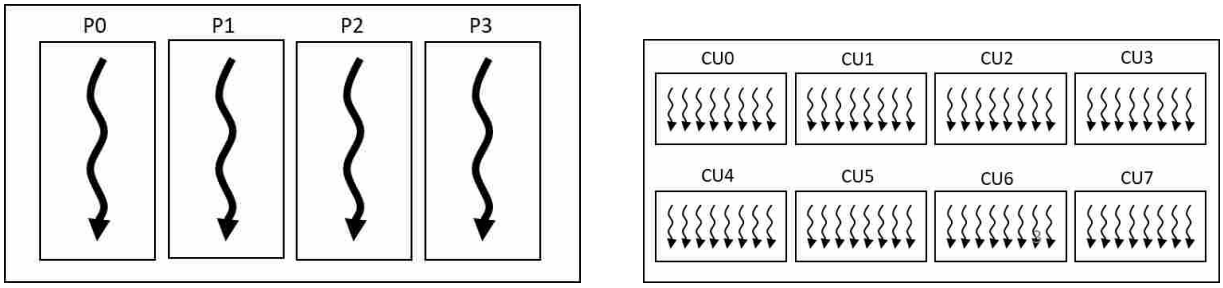
CHAPTER 3: HETEROGENEOUS CPU-GPU SYSTEM SIMULATION

This chapter discusses the implementation and modeling methodologies of M2S-CGM, as first presented in [3], with a significant expansion of discussion of the implementation methodology of M2S-CGM. M2S-CGM is a novel computer architectural simulator that provides end-to-end simulation of the system elements required to simulate non-coherent and coherent CPU-GPU heterogeneous workloads. M2S-CGM extends the CPU and GPU models of a previously established architectural simulator called Multi2Sim [15] and completely replaces Multi2Sim's existing memory system with a new custom memory system called CGM. CGM is built utilizing the KnightSim modeling methodology, as presented in chapter 2, and comprises coherence protocols, configurable CPU and GPU cache structures, directories, virtual memory mechanisms, switching fabrics, a system agent, a memory controller, and DRAMSim2 [23].

The chapter starts with providing a detailed background regarding the current state-of-the-art in GPGPU programming and heterogeneous CPU-GPU hardware configurations. Next, discussion of M2S-CGM's implementation methodology, software architectural makeup, and current simulation capabilities is provided. Finally, the chapter concludes with a validation study of M2S-CGM. The validation study utilizes a select group of the Rodinia OpenMP and OpenCL benchmark's and makes comparisons between the execution results of M2S-CGM and that of a physical test system [22, 24, 25]. The baseline results documented in the validation of M2S-CGM are then used to support the architectural experiments presented in Chapter 4.

3.1 Background and Motivation

The start of the GPGPU era, circa 2005, served as a tipping point in the mainstream use of GPUs as co-processing elements to the CPU—the crux of a CPU-GPU heterogeneous system [26]. Henceforth the GPGPU programming model started to gain momentum in its evolution regarding both its general processing performance and programmability. These advancements in the programming model then resulted in the rise of its popularity in its use for processing scientific workloads with problems that are heavily parallelizable. In comparison to CPU bound multithreaded applications, GPGPU applications are designed to offload extremely parallelizable code segments onto the GPU where the GPU's Single Instruction Multiple Data (SIMD) architecture can provide significant speedup over a CPU's Multiple Instruction Multiple Data (MIMD) or Single Instruction Multiple Thread (SIMT) equivalent implementations [27, 28].



(a) CPU MIMD depiction

(b) GPU SIMD depiction

Figure 3.1: CPU and GPU Architectural Differences

Fig. 3.1 depicts the architectural and processing differences between the CPU and GPU that are directly attributed to the differences in speedup between workloads suited for each processor type. In the CPU each core is designed to run "thicker" threads where higher levels of performance are attributed to higher instructions processed per clock. Conversely, each GPU compute unit is designed to run "thinner" fibers (micro threads) where only one instruction is processed per clock. In the GPU higher levels of performance are attributed to higher levels of data parallelism. In both

the CPU and GPU implementations, each thread/fiber works on different data elements. The CPU is also capable of processing SIMD instructions, the same way the GPU does, through the support of the SSE instruction set [29]. However, when comparing the two diagrams and taking a data set's size into consideration, it is immediately apparent that a CPU would require additional iterations to process a data set during execution as compared to the GPU.

At the time of writing this dissertation there are currently two mainstream approaches to CPU-GPU system architecture: (1) a traditional approach where one or more GPUs are located on discrete graphics cards connected through one of several peripheral component interconnect configurations (i.e. PCIe [30]) and (2) a more recent approach where the GPU is colocated with the CPU on-die as an integrated graphics chip and is connected to the rest of the system via the on-die switching fabric [31, 32]. Despite the differences between these two architectural approaches, the GPGPU programming model has remained the same with the GPU treated as a separate system element that operates independently and in its own memory address space.

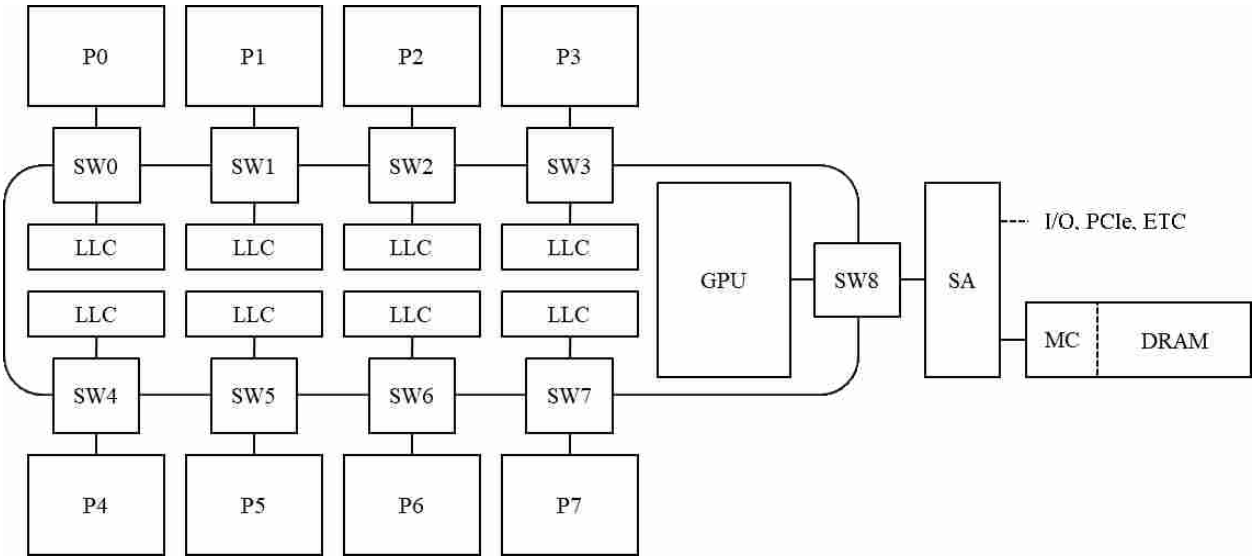


Figure 3.2: Simulated Heterogeneous System Node Architectural Block Diagram

In the GPGPU programming model the user must endeavor to partition the execution of the appli-

cation between the CPU and GPU such that the overall application provides higher performance to the equivalent multithreaded version. The performance of the GPGPU version of the application, as compared to a multithreaded equivalent, can then be subjective and depends on several system variables, such as, number of CPU cores, number of GPU compute units, memory system latency, contention, data bandwidth, and the number of interactions between the CPU and GPU. As a rule of thumb, research [3] shows that fewer interactions between the CPU and GPU and larger data parallel problem sizes for the GPU will yield higher levels of speedup over applications with more interactions between the CPU and GPU and smaller or lower data parallel problem sizes. For this reason it remains that some GPGPU implementations do not perform better than their multithread equivalents, which continues to compel further research in this area [33].

However, with the inclusion of the GPU on-die with the CPU new heterogeneous hardware and software design spaces can be explored and new levels of parallel system performance are theoretically achievable. This serves as the motivation for producing M2S-CGM. M2S-CGM provides the foundational infrastructure required to study system architectural interactions between two processing elements with two different instruction set architectures and very different processing capabilities. M2S-CGM allows for exploration of changes supporting performance improvements for heterogeneous workload executions and the study of the trade-offs those design choices impose. M2S-CGM allows us to experiment with both the GPGPU programming model and its supporting hardware design spaces allowing for higher levels of hardware and software co-design. The following section discusses the implementation methodology of M2S-CGM and provides a detailed overview of its software architectural makeup and construction.

3.2 M2S-CGM Implementation Methodology

Fig. 3.2 shows a full example configuration of the M2S-CGM simulator. In the example configuration, M2S-CGM is configured to simulate a realistic processor with integrated graphics and other ancillary System On Chip (SOC) components. The makeup of the simulated processor includes an x86 system emulator (not shown), multicore x86 CPU timing model, a Southern Islands system emulator (not shown), multi-compute unit Southern Islands GPU timing model, a detailed multi-level cache memory system, virtual memory mechanisms (not shown), switching fabric, system agent, memory controller, and SDRAM. The figure depicts a system architecture that is configured similarly to the Intel Haswell Core i7 Devil's Canyon architecture [34, 35]. The figure shows a configuration where the L2 caches, L3 caches, GPU, and system agent are colocated and connected together by a switching fabric with a ring bus topology. More complex configurations are possible, however this particular configuration provides a model of a real world chip architecture regarding commodity off-the-shelf processors.

3.2.1 x86 System Emulation

M2S-CGM extends the x86 System Emulator found in Multi2Sim—the most critical and complex component of the simulator. The extensions performed comprise:

- The addition of new simulated x86 system calls.
- New x86 CPU instructions enabling the successful execution of new OpenCL and OpenMP benchmarks on the CPU.
- New system call related statistics.
- CPU thread management changes.

- The instruction level emulation of specific system calls of interest that are performed by the operating system during OpenCL executions.

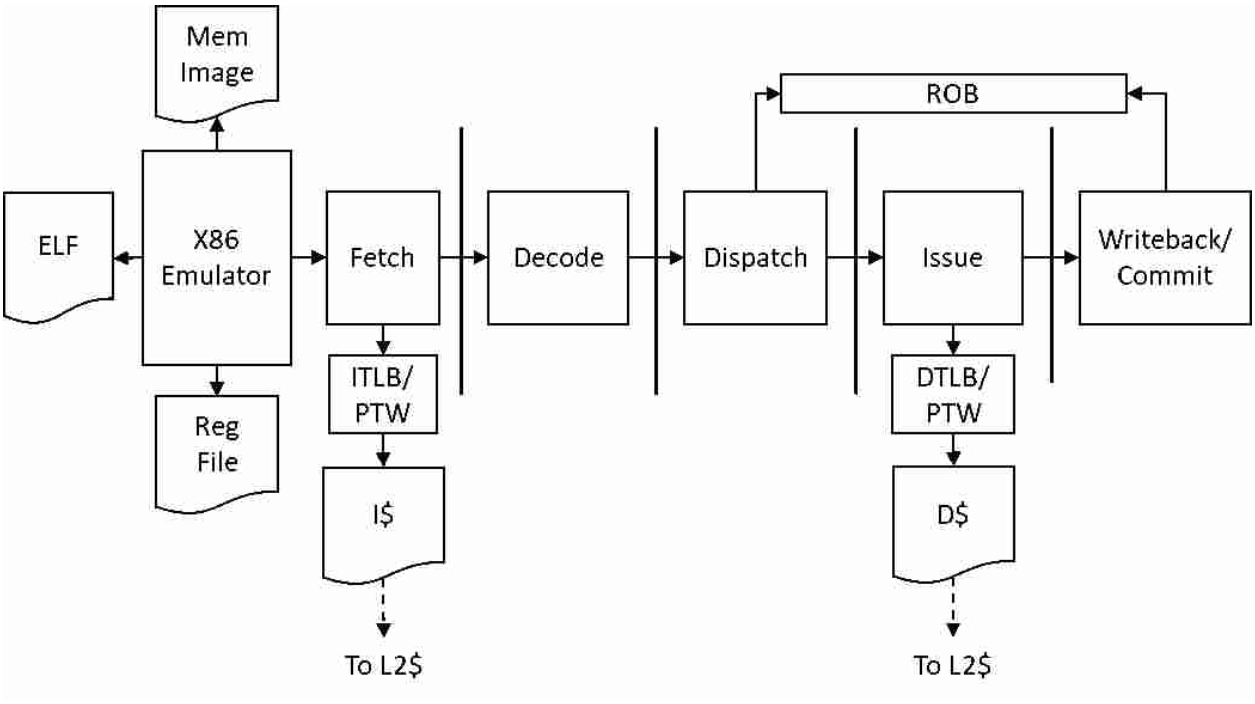


Figure 3.3: Simulated CPU Implementation Approach

Fig. 3.3 shows a summary of how the x86 system emulator works in conjunction with the x86 CPU Timing Model. In the figure, all of the components to the left of the fetch stage comprise the x86 system emulator. At initialization time, the x86 system emulator first sets up and configures the x86 runtime environment. This comprises the establishment of a memory image, stack, CPU register file, execution context (thread), and the assignment of the stack and instruction pointers for the entry context. During execution, the x86 system emulator opens and reads the instructions within the application binary file. The application binary is compiled into an Executable and Linkable Format file. The x86 system emulator is responsible for updating the instruction pointer position after successful execution of each instruction.

The process of executing instructions comprises the disassembly and emulation of the instruction itself. Emulation results in the appropriate updates to the simulated CPU register file and memory image. Before the x86 system emulator moves on to emulate the next instruction the emulator creates an appropriate set of pseudo instructions that represent the micro operations that will pass through the CPU pipeline and places them in the Fetch stage input buffer. The Fetch stage is the first stage of the CPU pipeline timing model.

When executing a multithreaded application the x86 system emulator creates the additional threads specified by the application binary, assigns them to a simulated core, and then begins executing instructions for each thread during each cycle. Despite simulating a multithreaded application, execution in the simulator is sequential, which makes keeping the simulated memory image from becoming corrupted easy. Lastly, the rate at which the x86 system emulator reads and executes instructions is determined by the state of the fetch stage provided by the x86 CPU timing model. This means that if a given CPU core is unable to fetch for any reason the x86 system emulator does not run that cycle.

During system calls the simulator traps to the x86 system emulator where it acts as the the operating system and performs the actions subject to the system call prior to returning to the application. The added instruction level emulation of system calls provides a means to incorporate OpenCL system related overhead into the x86 CPU timing model. In the native version of Multi2Sim there is no timing model for any of the simulator's emulated system calls, which makes the full study of CPU-GPU system interactions not possible. Examples comprise the system calls performed during OpenCL executions related to memory management and memory movement back and forth between the CPU and GPU.

3.2.2 x86 CPU Timing Model

M2S-CGM extends the x86 CPU timing model found in Multi2Sim. The extensions performed comprise:

- The addition of operating system related system call trap time.
- Memory access related functional corrections in the dispatch, issue, write-back, and commit pipeline stages.
- New CPU related statistics.
- Support for CPU and GPU cache flushes.
- CPU virtual memory mechanisms.
- Integration with the CGM memory system, see Sec. 3.2.5.

As shown in Fig. 3.3, all of the components to the right of the x86 system emulator form the x86 CPU timing model. The x86 CPU timing model comprises a generalized out-of-order 6 stage pipeline. The x86 CPU timing model can be configured as a hyper-threaded single or multicore CPU with configurable pipeline. Pipeline configuration allows for the specification of the maximum number of instructions to attempt to process each cycle and other performance settings specific to each pipeline stage. As mentioned in Sec. 3.2.1, the x86 system emulator will assign execution contexts to their applicable cores in the simulator. Then, the micro ops generated by the x86 system emulator for each context are passed to the fetch stage, which is the beginning of the x86 CPU pipeline timing model. Each stage runs once per CPU cycle and attempts to processes as many micro ops as possible up to the pipeline's configured maximum number of instructions. As the micro ops pass through the pipeline they are processed in an out-of-order manor, but are reordered prior to commit and write-back with use of the reorder buffer.

For memory system related statistics the state of the reorder buffer is used to determine if the CPU is busy running or stalled on a read or write request to the memory system. This is accomplished by checking if the reorder buffer is full and determining if a read or write is at the head of the buffer. However, despite the state of the reorder buffer, the x86 CPU timing model will continue to fetch until the fetch input queue is saturated, at that point the CPU core is fully stalled. The native CPU timing model in Multi2Sim removes memory system stores from the CPU reorder buffer before issuing the store to the memory system. In the M2S-CGM CPU model a store is first issued to the memory system in the issue stage with the hopes of bringing the memory system block to the CPU L1 cache early. Regardless, the store is only written into the L1 data cache, committed, written-back, and removed from the reorder buffer after the completion of all dependant and previously issued micro ops complete.

As mentioned in Sec. 3.2.1, the native x86 CPU Timing Model in Multi2Sim does not support timing for system calls. This means that system calls performed during the execution of an application in Multi2Sim will be charged zero cycles for the system call itself. This modeling approach is sufficient for applications that do not rely heavily on the performance of system calls. However, this approach severely limits the study of CPU-GPU related system interactions because the system interactions themselves are performed through a series of system calls. The system call related modifications to the CPU timing model provide a mechanism to ensure that all uops occurring before a system call are committed and written-back, a system call trap time is assessed, and then if required the micro ops related to the instruction level emulation of the system call pass through the appropriate CPU pipeline. After completion of the system call, execution is then resumed at the next instruction address pointed to by the application's next instruction pointer.

The CPU timing model interfaces with CGM at the fetch and issue CPU pipeline stages and is dependent on CGM's modeled memory system latency for correct execution timing. Before entering the memory system, virtual addresses are looked up in the translation lookaside buffer and on a

hit are translated from virtual to physical prior to accessing the respective L1 instruction or data cache. On a miss, execution then results in the eviction of the oldest entry in the translation lookaside buffer, a subsequent walk of the page table, a latency charged for the action, and finally an access of the respective L1 instruction or data cache. From there memory system accesses are execution driven and latency varies based on the memory system's configuration and the system wide contention and occupancy state. Access latency within the memory system, saturation of memory system elements, or lack of free miss status handling registers may result in any combination of an empty fetch queue, full reorder buffer, and full issue buffer which effectively results in measurable memory system stalls in the CPU.

3.2.3 Southern Islands GPU System Emulation

In addition to the previously mentioned x86 extensions, M2S-CGM also makes extensions to the Southern Islands GPU emulator found in Multi2Sim. The extensions comprise:

- New Southern Islands GPU instructions enabling the successful execution of new OpenCL benchmarks on the GPU.
- GPU virtual and physical address mechanism additions.

The implementation of the Southern Islands GPU system emulator is similar to that of the x86 system emulator, as discussed in Sec. 3.2.1. Fig. 3.4 shows a summary of how the Southern Islands GPU system emulator works in conjunction with the Southern Islands GPU timing model. In the figure all of the components to the left of the GPU fetch stage comprise the Southern Islands GPU system emulator. Unlike the x86 system emulator, the Southern Islands GPU system emulator does not automatically start with the execution of an application on M2S-CGM. Instead, the guest

application executing on M2S-CGM must first configure the GPU's kernel and data prior to execution on the GPU, see Chapter 4. This is accomplished with the use of an OpenCL runtime and the appropriate GPU driver simulation. Once the guest application has completed the configuration of the GPU's kernel and its input data, the Southern Islands GPU system emulator is directed to start execution with the first instruction of the GPU kernel by a system call. At the beginning of the execution the Southern Islands GPU system emulator makes modifications to its designated memory image and sets up the GPU register file.

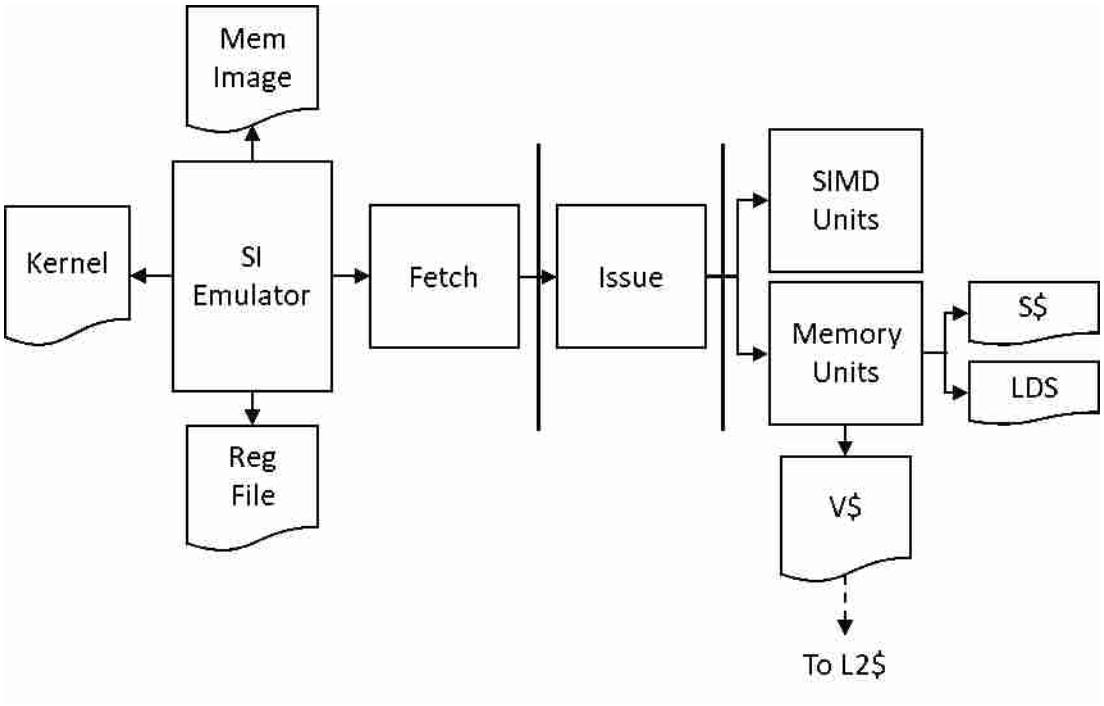


Figure 3.4: Simulated GPU Implementation Approach

Similarly to the x86 system emulator, the Southern Islands GPU system emulator opens a GPU kernel, compiled into an Executable and Linkable Format file, and begins to execute instructions. The process of executing GPU instructions comprises the disassembly and emulation of one instruction per cycle. Additionally, the emulator only executes one context (stream in GPGPU terms) on the GPU at a time. Each instruction emulation results in the appropriate updates to the simulated GPU

register file and the GPU's designated memory image. Then, the emulator creates and passes a pseudo instruction to the appropriate GPU Compute Unit fetch input buffer and updates the instruction pointer. Like the CPU, the Fetch stage is the point of entry to the GPU compute unit pipeline model. At the end of execution the GPU's designated memory image contains the data output from the execution of the GPU kernel and the CPU is notified that the GPU has completed its tasks by a GPU driver interaction.

Several new GPU instructions were added to the emulator to expand the range of benchmarks the Southern Islands GPU system emulator can execute. The Southern Islands instruction set architecture is specific to the Radeon HD 7000 series GPUs and is partially publicly known through a published instruction set architecture reference guide [36]. The new GPU virtual and physical address mechanism additions bring both the CPU and GPU into the same virtual memory system at large. When the GPU executes in a non-coherent mode the GPU is assigned its own unique physical memory pages and when executing in a coherent mode the GPU shares the same physical memory pages with the CPU.

3.2.4 Southern Islands GPU Timing Model

In addition to the previously mentioned x86 extensions, M2S-CGM also extends the Southern Islands GPU model found in Multi2Sim. The extensions performed comprise:

- Memory access related functional corrections to the GPU issue stage and vector unit pipeline.
- Vector memory access coalescing.
- New GPU related statistics.
- GPU virtual memory mechanisms and Input–Output Memory Management Unit

(IOMMU).

- Timing models for scalar caches and local data stores.
- A GPU hub for multiplexing memory request.
- Integration with the CGM memory system, see Sec. 3.2.5.

As shown in Fig. 3.4, all of the components to the right of the Southern Islands GPU system emulator form the Southern Islands GPU timing model which models an application specific in-order GPU multistage pipeline. The Southern Islands GPU timing model can be configured as a single context multi-compute unit GPU with each compute unit comprising a front end, SIMD lanes, a scalar unit, a branch unit, a vector memory unit, and a Local Data Share (LDS) unit. The pipeline itself is configurable, however the pipeline only supports a single instruction execution each cycle.

As mentioned in Sec. 3.2.3, the Southern Islands GPU system emulator creates and assigns a single GPU execution context to the Southern Islands GPU timing model. The context work-groups are then spread across the number of available compute units during execution. Kernel micro ops generated by the Southern Islands GPU system emulator are fetched by the compute unit front end and are then issued to their respective processing unit; either a scalar unit, a branch unit, a vector memory unit, or the SIMD unit. Memory accesses, depending on the type, take a route through the scalar, branch, and vector memory units, shown as "Memory Units" in the figure. Processing related instructions take a route to the SIMD unit, shown as "SIMD Units" in the figure. During execution memory accesses performed in either the scalar, branch, or vector memory units result in a local data share, scalar cache, or vector cache access. Vector cache accesses are the only cache accesses that leave the GPU core for the external memory system.

Functional corrections to the issue stage and vector unit pipeline in the native Multi2Sim GPU timing model include vector memory unit access coalescing and proper stall of the GPU's compute units on memory system resource saturation. In the M2S-CGM Southern Islands GPU timing model, vector memory accesses are coalesced into single block requests at the GPU L1 cache level. Once the GPU L1 cache coalescer is full or the miss status handling register is full the vector memory unit fills and then stalls. Once stalled the issue stage can no longer place memory requests in the vector memory unit's pipeline, which results in a GPU compute unit stall on memory access request.

Additionally, the native Southern Islands GPU timing model found in Multi2Sim lacks the simulated computer architectural components required to resolve virtual to physical address translations when communicating externally with the main memory system. The native Southern Islands GPU timing model assumes that all data addresses are virtual and are resident within a generic model of the GPU's internal memory. In effect this means that the GPU and CPU memory systems are incompatible and must be modeled as two disparate entities. This approach lends itself to studies of the internal mechanisms of the GPU, but severely limits its use in the study of the interactions between the CPU, GPU, and memory system at large. The modifications related to the Southern Islands GPU timing model's virtual memory mechanisms take these issues into account and provide timing for the computer architectural components involved with memory address translation. In conjunction with the modifications to the Southern Islands GPU system emulator, this creates a full system change that allows for a modeled shared memory system between the CPU and GPU; like the real hardware. A future addition of a direct memory access engine model would further enhance the fidelity of the CPU-GPU system model on whole.

The GPU timing model interfaces with CGM at the local data share, scalar cache, and vector cache entry points. There CGM provides latency for these internal memory elements and external memory system accesses. In addition to the caches and crossbar, the GPU memory system model

includes a hub and IOMMU (see Fig. 3.2). The hub multiplexes memory system accesses between the GPU's L2 cache banks and the switching fabric. The IOMMU performs address translations for the GPU and can alternately perform both forward and reverse address translations if utilizing virtual addressing within the GPU's caches. Additionally, memory system accesses destined for the external memory system include a configurable virtual or physical address schema. For experimentation purposes it is assumed that GPU virtual to physical address translation can occur at either the interface between each compute unit and the first level of cache or within the GPU's IOMMU. In a real system, a GPU TLB miss would result in a required intervention by the CPU to assist in the address translation. An efficient design of the GPU's hardware that enables the GPU to share a virtual address space with the CPU is still an open research question.

3.2.5 Memory System Timing Models

M2S-CGM's memory system timing model is provided by a stand alone memory system simulator called CPU-GPU Memory, or CGM for short. CGM is a new and custom memory system model designed to support the research performed and presented in this dissertation. As described in Sec. 3.2.2 and Sec. 3.2.4, CGM completely replaces Multi2Sim's disparate CPU and GPU memory system models with a fully cohesive CPU and GPU memory system model. This results in one comprehensive computer architectural system model that brings together the CPU, GPU, and memory system.

A depiction of the memory system simulation elements provided by CGM is shown in Fig. 3.2. CGM provides timing models of configurable CPU and GPU cache structures, input-output controllers, switching fabrics, crossbars, a system agent, a memory controller, SDRAM, and other discrete system elements such as the GPU's scalar cache, local data share cache, and hub. In addition, CGM also includes timing models for virtual memory mechanisms, such as memory management

units, translation lookaside buffers, page table walkers, and includes a directory based CPU MESI coherence protocol and a GPU MESI coherence protocol.

All of the memory system timing model elements within the scope of CGM are implemented as KnightSim contexts using the KnightSim modeling methodology as discussed in Sec. 2.3. This means that each memory system timing model element is advanced when given work to perform. Advanced elements wake up and try to process their assigned work until success. Once complete, the next element is given work to perform and it is in turn advanced by the previous element. On each advance, memory system simulation elements assess a latency for the performance of their work by pausing until that latency has expired. The assessed latency provides the modeled occupancy of that element. Contention is modeled as resources begin to fill or saturate creating element stalls and longer access latency.

The start of a memory system access begins at the interface between CGM and the CPU or GPU. Once the CPU or GPU completes an address translation a load or store will then be issued to the memory system. In execution, the CPU or GPU first checks the status of the L1 cache top request Rx queue. If there is space for a request, the CPU or GPU will then proceed to insert the memory request into the queue and advance the appropriate L1 cache. If the top request Rx queue is full the CPU or GPU will try again each cycle until success. Once enqueued, the memory request will wait in the top request Rx queue until retrieved for processing by the L1 cache. The number of cycles the memory system request must wait is determined by the current state of the memory system at large. The L1 cache has been advanced by the CPU or GPU and will wake up each cycle in an attempt to process the memory request. Once processed by the L1 cache, the result will either be a hit or a miss depending on the state of the memory block in the cache. Hits are returned to the CPU core or GPU compute unit immediately. However, a miss will result in the lower level memory system's MESI coherence protocol coming into play, see Sec. 3.2.6. The following subsections summarize the implementation methodology of each memory system timing model and provides

expanded discussion regarding their specific tasks.

3.2.5.1 Cache Timing Model

Fig. 3.5 shows the basic elements that all CPU and GPU cache structures comprise within CGM. The basic architectural components of the cache are summarized as:

- A cache level control function (i.e. L1, L2, L3, scalar, local data share, and vector cache control).
- Top and bottom request, reply, and coherence receive (Rx) and transmit (Tx) queues.
- A Miss Status Handling Register (MSHR) and associated coalescer.
- A directory based cache table.
- Write-back, retry, and join queues.
- Input-output controllers for request, replies, and coherence messages leaving the cache.

All cache structures comprise a configurable cache size, memory block size, MSHR size, coalescer size, set associativity, and latency. Additionally, all cache structures support a write-back schema, a least recently used cache replacement policy, and a directory based MESI cache coherence protocol. All caches are implemented with an inclusive caching schema and are connected via a modeled system bus or by a modeled switching fabric, see Sec. 3.2.5.2. Furthermore, each cache level operates at the same frequency as its respective CPU or GPU and shares a common cache control function that is implemented as a single cache type KnightSim context.

When a cache controller is advanced the cache controller will first invoke its scheduler. The scheduler maintains a record of the current state of the various sub elements within the cache and determines what action the cache controller will take in a given cycle. The cache's state record comprises the current sizes of the MSHR, coalescer, and the state of various queues within the cache during execution. The scheduler gives first priority to new CPU or GPU memory requests and second priority is given to lower level memory system replies and other coherence related traffic.

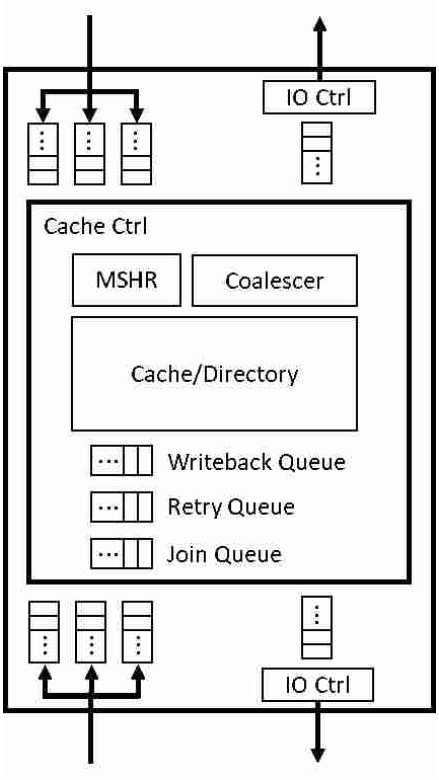


Figure 3.5: Cache Architecture

The scheduler is responsible for performing request flow control in the cache. If the cache's scheduler finds that either the MSHR, coalescer, write-back queue, retry queue, or bottom request Tx queue are saturated the cache is unable to process any new requests and must stall that cycle. The

cache will continue to stall until a previously processed memory request is returned and can complete. Once a previous memory request completes resources within the cache itself become free which then allows the cache to process a new memory request. In addition, the scheduler is also responsible for determining if new memory requests are to be coalesced with outstanding memory requests. Coalescing is required if the subject memory block is in the transient sate, there is an associativity conflict, or if there is a directory conflict. The scheduler will retry coalesced memory requests when the appropriate memory block is brought to the cache and resources in the cache table become free.

When processing a memory request, the cache controller will first probe the address for tag, index, and offset. Fig. 3.6 shows an example two way set associative cache table with write-back. The cache controller will simultaneously look for the memory block in both the cache table and in the write-back queue. Memory blocks are never in the write-back queue and cache table at the same time. After the address probe the cache controller will locate the appropriate set in the cache table and then compare the tags of each available way from the cache table to determine if the block is in the cache or not. The cache controller will also compare the tags of all entries in write-back to determine if the memory block is in write-back or not.

The cache's write-back queue significantly complicates things. Memory blocks that are present in the write-back queue are always valid and are either in a modified or exclusive state. Memory blocks in the exclusive state remain exclusive to the core until invalidated or flushed by the cache controller at an opportune time. When processing a memory request, it is possible for the cache controller to find that a memory block is uncached in the cache table, but still valid, dirty, and sitting in the write-back queue. CGM implements an optimized approach where all caches will attempt to write the memory block back into the cache table and process the memory request as a hit. If the cache controller is unsuccessful in finding a victim due to an associativity or directory conflict the write-back is immediately flushed from the cache and the memory request is coalesced

or NACKED back to the requester as appropriate.

The cache controller processes a miss by first freeing a cache line by evicting the least recently used non-transient cache line. Then after eviction, storing the requested memory block's tag in the free cache line, setting the free line to the transient state, and sending a memory request down to the next level in the memory system. When transmitting requests, replies, or coherence messages to another memory system component the cache controller constructs the appropriate request, reply, or coherence message and places the message in the top or bottom input-output controller's outgoing message queue. Sec. 3.2.6 discusses the implementation of the CPU and GPU MESI coherence protocols and provides additional detail regarding the types of messages transmitted between the various memory system components. Then, the cache's built in input-output controllers are responsible for all transmittal actions on behalf of the cache and will transmit the message along the appropriate virtual lane based on the message type. Message transactions between caches are bidirectional and are asynchronous.

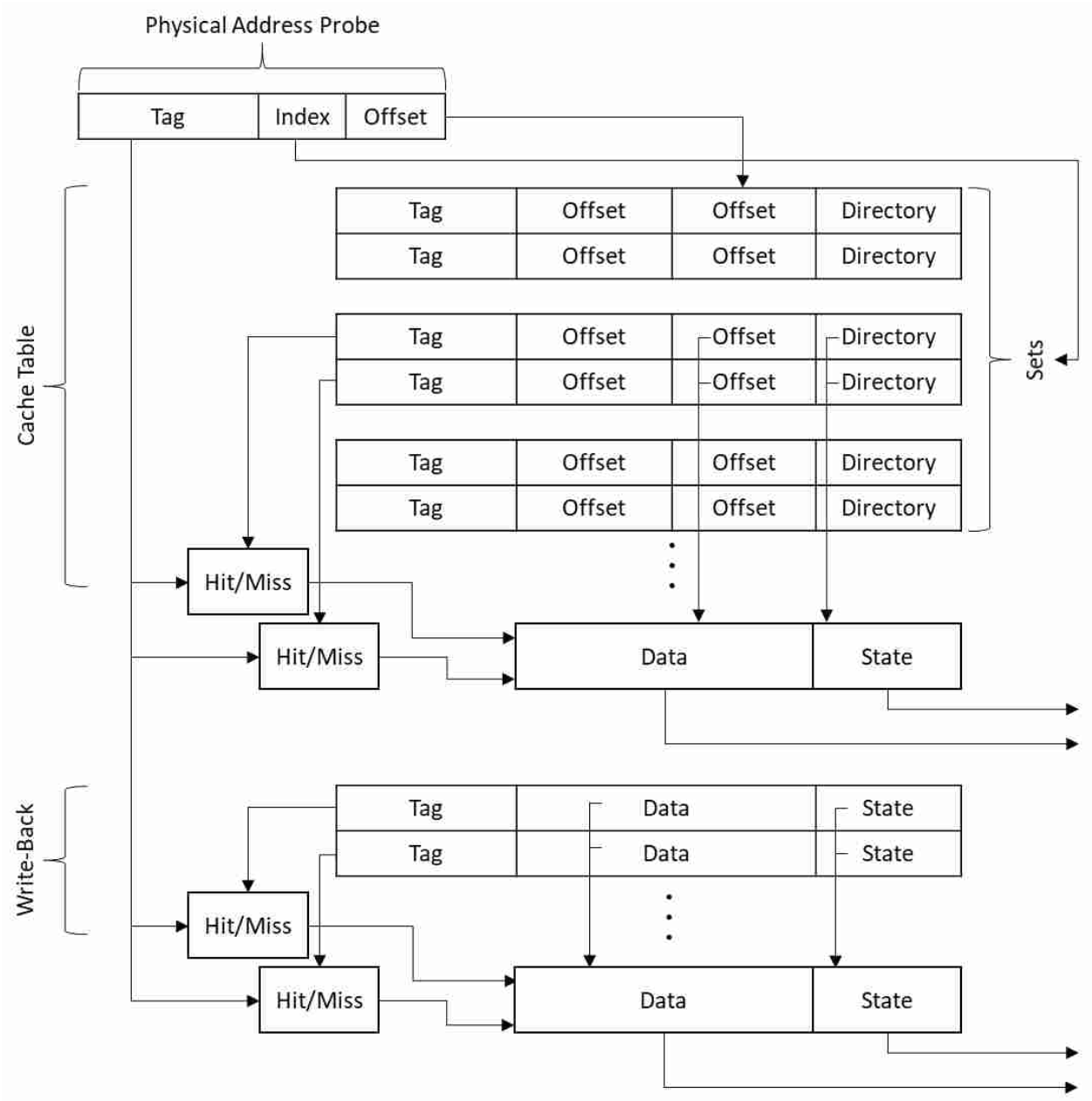


Figure 3.6: Architecture of a Two Way Set Associative Cache With Write-Back

3.2.5.2 Switch Timing Model

CGM connects major memory system components through a modeled on-die switching fabric. The on-die switching fabric is modeled by chaining together multiple individual switch models. Fig. 3.7 shows the architecture of a single switch model. The basic architectural components of the switch are summarized as:

- A switch control function.
- North, East, South, and West ports comprising virtual lanes for request, reply, and coherence Rx and Tx messages.
- An internal crossbar.
- A routing table.

An example KnightSim switch control function was previously discussed in detail in Sec. 2.3. The switch model comprises a configurable number of switches, configurable request, reply, and coherence Rx and Tx lane queue sizes, and a configurable latency. After specifying the desired number of switches the switching fabric automatically generates itself as a ring bus topology and utilizes a shortest path routing schema for the routing of message packets. Additionally, the switch model supports a round-robin schema when selecting a starting port during each execution. Currently the switch model's available ports include North, East, South, and West ports, however work has been done to make the switch model "N" ported. The same round-robin schema is used when selecting a starting request, reply, or coherence Rx lane at the beginning of each execution.

The round-robin schema increments the starting port and lane position at the end of each switch controller execution. Incrementing the starting position provides a mechanism for deadlock prevention in the switch model. The switch's crossbar model is shown in Fig. 3.8. The crossbar links

all possible inputs to all possible outputs. Despite having physical queues for request, reply, and coherence messages, the crossbar only has one set of wires per port and will only route one virtual lane type at a time. In CGM it is considered an error if a message packet being routed has the same destination and source.

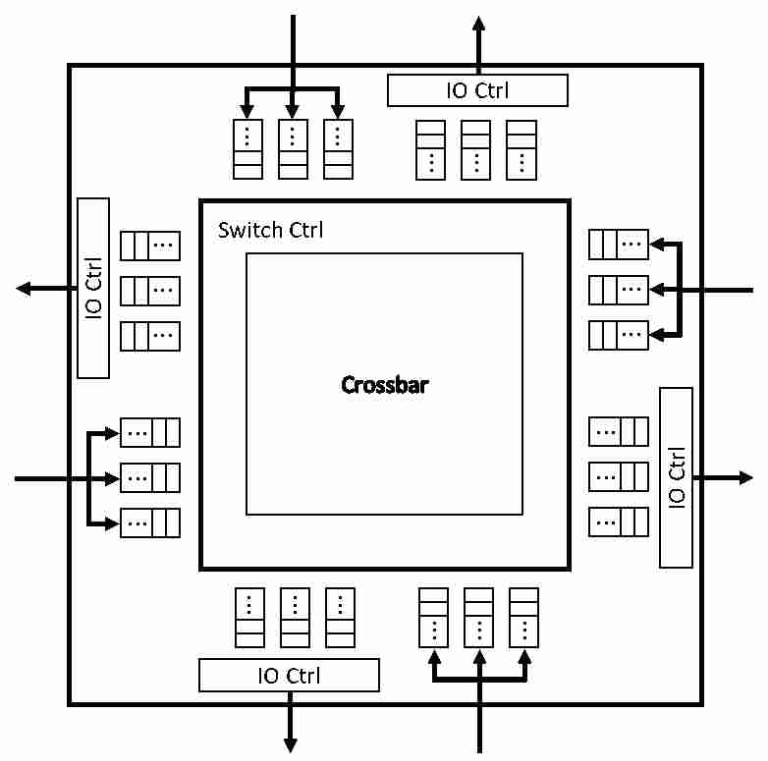


Figure 3.7: Switch Architecture

In comparison to the complexity of the cache control function the switch control function is relatively straight forward. A switch can be advanced by a connected memory system component (i.e. cache, switch, system agent, or GPU hub) by up to the number of ports modeled by the switch in a given cycle. After advancement the switch will start with the current lane and port provided by the round-robin schema. The switch will go through all ports looking for the presence of a message packet. If the switch controller finds a packet the switch will consult the destination and attempt to link the input port and lane to the correct output port and lane. The switch controller will link

the pair if the output port and lane queue have not previously been linked during the current execution and there is room for the message packet in the appropriate output port's lane queue. The switch controller will be unsuccessful at forming a link if the appropriate output port lane queue is full which results in stalls and contention along that virtual lane. The switch controller repeats this process for all ports during an execution epoch. After the switch controller goes through all available ports the message packets are then transferred via the crossbar to their respective output queues with their successfully linked routes. The switch model's step variable is then incremented by the number of successfully formed links.

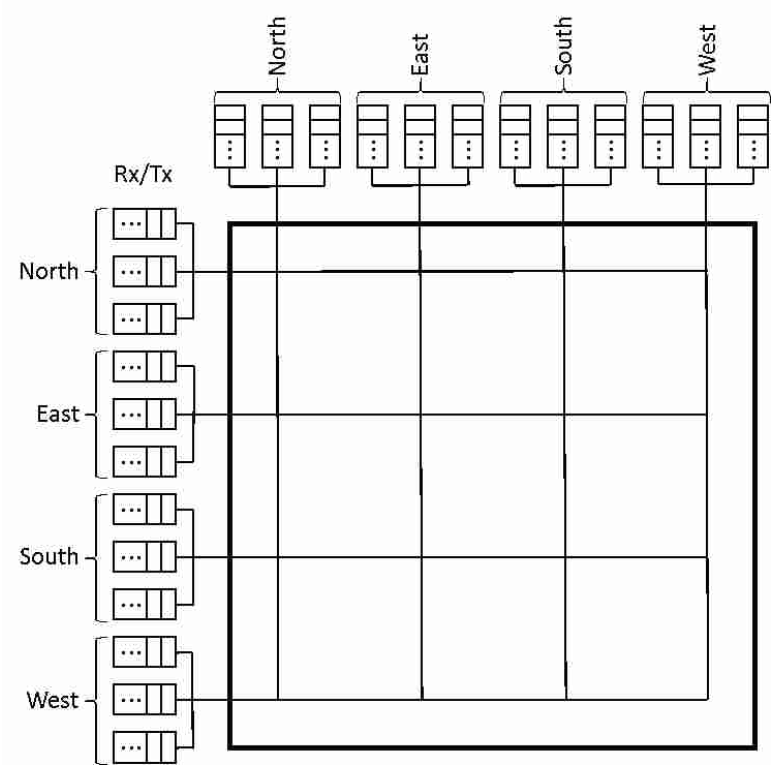


Figure 3.8: Switch Crossbar Architecture

Similar to the cache model, the switch model's built in input-output controllers are responsible for all transmittal actions on behalf of the switch and will transmit message packets along the appropriate virtual lanes to the next memory system component. The switch model's input-output

controllers also follow a round-robin schema for selection of the virtual lane which provides a mechanism to prevent deadlock in the system. Message packet transactions between each switch and all other memory system components are bidirectional and are asynchronous.

The switching fabric operates at a frequency that is independent of the CPU and GPU. In CGM there are currently three timing domains which comprise the CPU, GPU, and system frequency domains. The switching fabric sits in the system frequency domain and operates at a frequency that is specified as a ratio of the CPU's frequency. This is performed with use of a KnightSim macro that adjusts the number of cycles that a `pause()` will invoke. For example, if the CPU is operating at 4 GHz and the switching fabric is operating at 2 GHz the macro will adjust the `pause()` by a factor of two, which results in the switching fabric running at half the frequency. Bandwidth within the switching fabric is modeled by the input-output controllers. The input-output controllers divide the simulated message packet size in bytes by a variable flit size to determine transfer time latency. The latency is inclusive of the data register fill time.

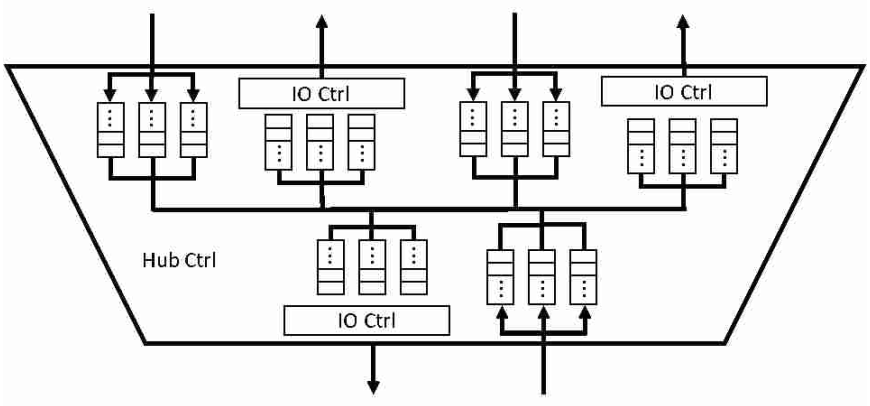


Figure 3.9: GPU HUB Architecture

3.2.5.3 GPU Hub Timing Model

CGM connects the Southern Island GPU model to the on-die switching fabric with the use of a GPU hub. Fig. 3.9 provides an architectural representation of the GPU Hub. The GPU's hub interfaces the GPU with the reset of the memory system by multiplexing memory requests, replies, and coherence messages between the GPU's internal memory hierarchy and the external memory system. The implementation of the GPU hub timing model is straightforward and comprises a hub control function, configurable request, reply, and coherence Rx and Tx lane queue sizes, and a configurable latency. Similar to the switch timing model, the hub controller uses a round-robin schema for selection of a starting input queue. The hub controller then polls each queue until a message packet is found. The hub controller then explicitly moves the message packet to the correct output port and lane. The hub controller schedules the transfer of one message packet per execution. Message packets bound for lower level memory are directed to the network right away. Message packets bound for higher level GPU memory must be routed to the appropriate L2 cache bank. The hub resides in the GPU frequency domain with bidirectional and asynchronous message packet transactions.

3.2.5.4 System Agent, Memory Controller, and SDRAM Timing Models

Fig. 3.10 shows the architecture of the system agent, memory controller, and SDRAM. Modern chip designs have moved the architectural components of the northbridge on-die with the CPU. These holistically designed CPUs are called System On Chip (SOCs) and include a system agent that comprises the interfaces provided by the system's northbridge [37]. These interfaces include the display interface, Direct Media Interface (DMI), PCI Express (PCIe), and memory controller. CGM implements a system agent, but does not implement the display, DMI, and PCIe interfaces because the components these interfaces would integrate are out of the scope of the research at

hand. However, these interfaces can be added to the system agent timing model with ease if future research work necessitates.

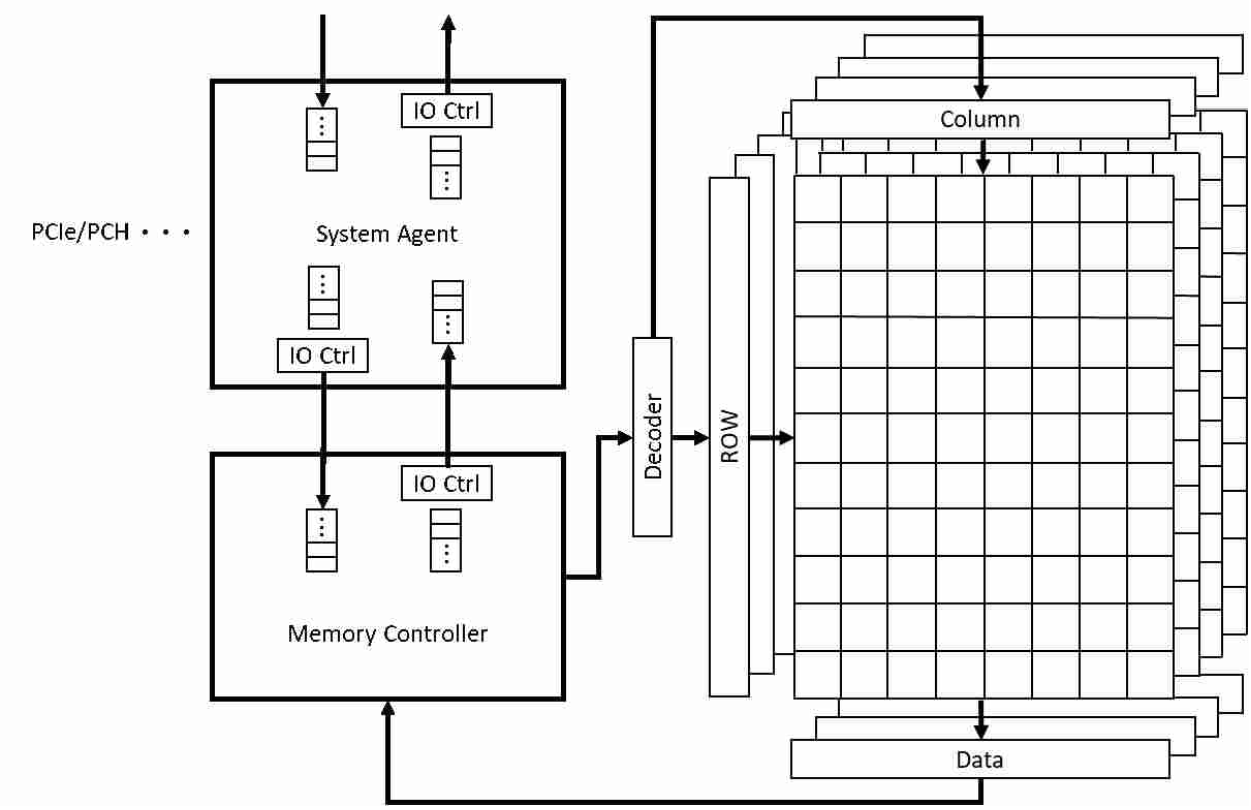


Figure 3.10: System Agent, Memory Controller, and SDRAM Architecture

The system agent timing model comprises a system agent control function, configurable top and bottom Rx and Tx queues, and a configurable latency. The memory controller can be considered a part of the system agent, but is independently modeled in CGM. The memory controller timing model comprises a memory controller control function and configurable top Rx and Tx queues. The SDRAM timing model is provided by DRAMSim2 [23]. DRAMSim2 provides detailed timing models for several state-of-the-art SDRAM and DDR memory modules. The memory controller timing model integrates with DRAMSim2 with use of DRAMSim2's built in Application Programming Interface (API). Main memory is simulated as a multi-channel system with memory

striped over multiple memory modules. Message packet transactions between the system agent, memory controller, and SDRAM are bidirectional and are asynchronous.

When advanced by the attached switch node input-output controller, the system agent controller will poll its input request queue for a message packet. After finding the message packet the system agent controller will then route the packet to the memory controller. The system agent's input-output controller will place the memory packet in the memory controller's outstanding memory request queue and advance the memory controller. Once advanced the memory controller will then perform the appropriate load or store to SDRAM. DRAMSim2 provides functions for loading and storing to the simulated SDRAM modules. Additionally DRAMSim2 models contention and occupancy within the SDRAM modules and includes several selectable scheduling schemas. When a load or store is complete, DRAMSim2 will initiate a memory controller call back function notifying the memory controller that the load or store has completed. Stores to SDRAM are complete at that point, however loads require a reply message with data from the memory controller back to the requesting system cache.

3.2.6 Memory System Coherence Protocols

CGM currently implements a detailed directory based cache coherence protocol for both the CPU and GPU. Directory based cache coherence protocols for homogeneous chip multiprocessors have been around for a long time and take many forms, as described in [38, 39, 40, 41, 42, 43]. However, CGM introduces a directory based cache coherence approach for heterogeneous CPU-GPU systems. The schema comprises a detailed directory based MESI coherence protocol throughout the CPU's and GPU's memory hierarchy. The GPU's MESI protocol is implemented differently in comparison to the CPU's MESI protocol, however it is designed to connect and work seamlessly with the CPU's MESI protocol and effectively binds the two disparate memory hierarchies. This

design provides a straightforward and optimized coherence protocol for the CPU and GPU when operating individually, but also supports coherent interoperation of the CPU and GPU on shared memory objects in a collaborative processing environment.

The CPU and GPU MESI coherence protocols both operate using a directory located in the lowest level of cache in the system [44]. As shown in 3.2, the lowest level of cache for the CPU is the L3 cache bank and the lowest level of cache for the GPU is the GPU L2 cache bank. When configured to operate individually in a non-coherent mode the CPU L3 cache bank and the GPU L2 cache bank both arbitrate access to memory blocks between processors in their respective memory hierarchies. In the non-coherent mode, loads and stores go directly to main memory from the CPU L3 cache bank and similarly go directly to main memory from the GPU L2 cache bank over the switching fabric. When configured to operate in a coherent mode, memory requests are routed from the GPU L2 cache bank to the CPU L3 cache bank. The CPU L3 cache bank then arbitrates memory accesses between both the CPU and GPU in whole and will maintain a coherent memory system between the two processor types.

In modern x86 CPUs the L1 instruction, L1 data, and L2 cache banks are private to each core and in modern GPUs the L1 instruction and L1 data caches are private to each compute unit. However, the CPU L3 and GPU L2 cache banks are shared among all processing units within their respective memory hierarchies. In CGM the CPU and GPU L2 cache banks are implemented as "smart caches" and perform several important memory coherence related functions on behalf of their respective processors. These task comprise macro core memory block evictions and optimized L2 cache level coherence mechanisms, such as request forwarding and joins.

Table 3.1: CPU L1 Instruction Cache MESI Coherence Protocol State Table

CPU L1 Instruction Cache	
State	**Fetch**
M	N/A
E	N/A
S	Hit
I	Miss ->GetS

As described in 3.2.5.1, each cache comprises a cache control function that is implemented as a KnightSim context with a specific set of coherence protocol functions. After a cache has been advanced the cache controller then checks the message packet type. Valid message types comprise request, reply, and coherence message types. Once the cache controller decodes the message type the corresponding coherence protocol action is taken. Coherence protocol actions are implemented as call-back functions that are assigned to each cache structure at initialization time. The following subsections provide expanded detail regarding the implementation of the CPU and GPU MESI coherence protocols in CGM and how they work in an integrated operational environment. The following protocol sate tables assume that generalized cache scheduling functionality, like stalling and coalescing, has previously occurred as described in 3.2.5.1.

3.2.6.1 CPU L1 Instruction and Data Cache MESI Protocol State Tables

Table 3.1 shows the protocol state table for the CPU L1 instruction cache. The CPU L1 instruction cache is only required to support a shared or invalid state for cached memory blocks and does not utilize a write-back queue. This simplified design was chosen because the CPU L1 instruction cache is only required to cache read only memory blocks from the `.text` memory segment of the application. Additionally, CPU L1 instruction cache accesses are denoted as fetches instead of loads. This allows the rest of the memory system to easily determine if a memory request is for

read only memory or not. As shown in the table, the instruction cache controller will process a fetch to a memory block that is cached and in the shared state as a hit. A fetch to a memory block that is uncached or that is in the invalid state will be processed as a miss resulting in a fetch (GetS) request being sent down to the next level cache in the memory system.

Table 3.2 shows the protocol state table for the CPU L1 data cache. In comparison the data cache protocol state table is more complex than the instruction cache protocol state table. The data cache supports all states of the MESI coherence protocol and additionally supports in cache merges of write-back data stored in a write-back queue, see generalized cache functionality in Sec. 3.2.5.1. As shown in the table, the CPU L1 data cache controller will process a load or store to a memory block that is cached and in the modified or exclusive states as a hit. Memory blocks that are in the exclusive state are set modified as a result of a successful store requests. Additionally, the CPU L1 data cache controller will process a load request to a memory block that is cached and in the shared state as a hit.

Table 3.2: CPU L1 Data Cache MESI Coherence Protocol State Table

CPU L1 Data Cache			
State	**Substate**	**Load**	**Store**
M	M	Hit	Hit
E	E	Hit	Hit ->M
S	S	Hit	Miss ->Upgrade
	S_Pending_Upgrade	Upgrade_Miss ->Coalesce	Upgrade_Miss ->Coalesce
I	I	Miss ->Get	Miss ->GetX
	I_WB_Writable	Hit ->E/M	Hit ->M
	I_WB_Non_Writable	Miss ->Coalesce	Miss ->Coalesce

In a special case, the data cache controller will process a store request to a memory block that is cached and in the shared state as an upgrade miss. This is considered a miss because memory blocks in the shared sate cannot be written to because they are shared across one or more processor cores. Therefore, the data cache controller must then set the shared memory block pending and

send an upgrade request to the last level cache (directory) in the memory system. Memory blocks that are waiting on an upgrade to complete are considered transient. A successful upgrade request will result in the memory block's eviction by all other holding cores and then the subsequent upgrade of the requesting core's transient memory block to the exclusive state.

The data cache controller will process a load or store to a memory block that is uncached, not in write-back, or that is in the invalid state as a miss resulting in a load (Get) or store (GetX) request being sent down to the next level cache in the memory system. If the cache controller finds that the requested memory block is in write-back the cache controller will process the request according to the state of the cache table set.

3.2.6.2 CPU L2 Cache MESI Protocol State Table

Table 3.3 shows the protocol state table for the CPU L2 cache. The CPU L2 caches are implemented as smart caches that perform several macro level coherence functions on the behalf of the CPU core it supports. Like the CPU L1 cache, the CPU L2 cache supports all states of the MESI protocol and additionally supports in cache merges of write-back data stored in the write-back queue. The CPU L2 cache handles core level block evictions on behalf of the last level cache (directory). All memory requests arriving at the CPU L2 cache are unique because the CPU L1 instruction and data caches perform block level coalescing at the L1 level. This intrinsic property of the memory hierarchy helps to simplify the CPU L2 protocol state table somewhat.

Table 3.3: CPU L2 Cache MESI Coherence Protocol State Table

CPU L2 Cache					
State	**Substate**	**Fetch**	**Load**	**Upgrade**	**Store**
M	M	N/A	Hit ->PutX	N/A	Hit ->PutX
E	E	N/A	Hit ->PutClnX	N/A	Hit ->M ->PutX
S	S	Hit ->PutS	Hit ->PutS	Miss ->Upgrade_Fwd	Miss ->Upgrade
I	I	Miss ->GetS	Miss ->Get	Miss ->GetX	Miss ->GetX
	I_WB_Flush_Conflict	N/A	Miss ->Get_Nack	N/A	Miss ->GetX_Nack
	I_WB_Writable	N/A	Hit ->E/M ->PutClnX/PutX	N/A	Hit ->M ->PutX
	I_WB_Non_Writable	N/A	Miss ->Coalesce	N/A	Miss ->Coalesce

The CPU L2 cache controller will process a fetch request to a memory block that is cached and in the shared state as a hit resulting in a reply (PutS) back to the requester. Likewise, the cache controller will processes a fetch request to a memory block that is uncached or invalid as a miss resulting in a fetch request (GetS) being sent down to the next level cache in the memory system. The CPU L2 cache controller will process a load or store to a memory block that is cached and in the modified or exclusive state as a hit. The cache controller will also process a load request to a memory block that is cached and in the shared state as a hit. After processing a hit the L2 cache controller will build the appropriate reply message (PutS, PutClnX, or PutX) and send back to the requesting L1 cache.

The CPU L2 cache controller will forward an upgrade request to the last level of cache (directory) if the memory block is cached in the shared state. In a special case it is possible for the CPU L1 data cache to drop the memory block and for the the L2 cache controller to receive a store to a memory block in the shared state. If this occurs the L2 cache controller processes the request as an upgrade in the same manor as the L1 data cache, but will reply with a PutX to the L1 data cache when the memory block is upgraded.

An eviction of a memory block in the L2 cache automatically results in a flush of the memory block at the L1 cache level. If the memory block is shared, the L2 cache does not wait for the L1 cache to acknowledge the flush and will continue to process requests, however if the memory block is exclusive or modified the L2 cache must wait for an acknowledgement from the L1 cache. An L2 eviction of a block in the exclusive or modified state can lead to two special cases at the L2 cache level that can occur while waiting for the eviction to complete. First, the L2 cache controller will process a new load or store to a memory block that is pending a data cache flush as a flush conflict and issue the appropriate NACK (Get/GetX NACK) going back to the requester. Second, the L2 cache controller will go ahead and process an upgrade request of a memory block that is pending a data cache flush as a miss resulting in a store request (GetX) being sent down to the next

level cache in the memory system.

The L2 cache controller will process a load or store to a memory block that is uncached, not in write-back, or that is in the invalid state as a miss resulting in a load (Get) or store (GetX) request being sent down to the next level cache in the memory system. If the cache controller finds that the requested memory block is in write-back the cache controller will process the request according to the state of the the cache table set. The L2 cache performs several other coherence related functions regarding coherence joins on the behalf of the last level of cache (directory). The additional coherence functions of the CPU L2 cache are covered in the following section.

3.2.6.3 CPU L3 Cache MESI Protocol State Table

Tables 3.4 and 3.5 show the various protocol states for the CPU L3 cache. As shown in 3.2, the CPU L3 cache is the lowest level cache in the system and therefore is responsible for arbitrating memory requests between the CPU cores when set in the non-coherent processing mode and between the CPU and GPU when set in the coherent processing mode. The L3 cache performs arbitration by utilizing a directory with each cache line, as shown in Fig. 3.6.

The L3 cache supports all states of the MESI coherence protocol and additionally supports in cache merges of write-back data stored in the write-back queue. When processing requests as a hit, the L3 cache will set a bit in the directory that indicates which cores hold the subject memory block. This creates several new MESI sub states, which are discussed here. The CPU L3 cache also uses an additional bit in the directory to indicate a pending state for multi-part coherence related interactions. Also, the CPU L3 cache services all processors and can receive multiple requests for the same memory block from different processors which requires some additional sophistication in the cache's coalescing mechanisms.

Table 3.4: CPU L3 Cache MESI Coherence Protocol State Table: Fetches and Loads

L3 Cache Loads			
State	**Substate**	**Fetch**	**Load**
M	M	N/A	Hit ->Set_Core_Holder ->PutX
	M_Owned	N/A	Hit ->Set_Core_Holder ->PutX
	M_Pending_Owned	N/A	Hit ->PutX
	M_Pending	N/A	Miss ->Get_Nack
	M_Held	N/A	Hit ->Set_Pending ->GetFwd
E	E	N/A	Hit ->Set_Core_Holder ->PutClnX
	E_Owned	N/A	Hit ->Set_Core_Holder ->PutClnX
	E_Pending_Owned	N/A	Hit ->PutClnX
	E_Pending	N/A	Miss ->Get_Nack
	E_Held	N/A	Hit ->Set Pending ->GetFwd
S	S	Hit ->Set_Core_Holder ->PutS	Hit ->Set_Core_Holder ->PutS
I	I	Miss ->SDRAM_Load	Miss ->SDRAM_Load
	I_WB_Flush_Conflict	N/A	Miss ->Get_Nack
	I_WB_Non_Writable	N/A	Miss ->Coalesce
	I_WB_Writable	N/A	Hit ->Set_Core_Holder ->E/M ->PutClnX/PutX
	I_WB_Dir_Conflict	N/A	Miss ->WB_Flush ->Get_Nack

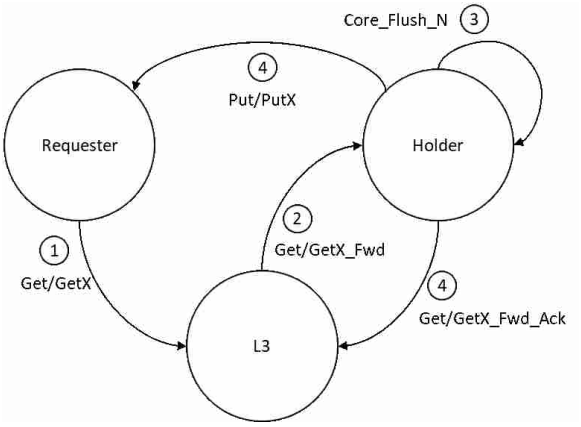
Table 3.5: CPU L3 Cache MESI Coherence Protocol State Table: Upgrades and Stores

L3 Cache Stores			
State	**Substate**	**Upgrade**	**Store**
M	M	N/A	Hit ->Set_Core_Holder ->PutX
	M_Owned	N/A	Hit ->Set_Core_Holder ->PutX
	M_Pending_Owned	N/A	Hit ->PutX
	M_Pending	Miss ->Upgrade_Nack	Miss ->GetX_Nack
	M_Held	Hit ->Set_Pending ->GetXFwd	Hit ->Set_Pending ->GetXFwd
E	E	N/A	Hit ->Set_Core_Holder ->PutX
	E_Owned	N/A	Hit ->Set_Core_Holder ->PutX
	E_Pending_Owned	N/A	Hit ->PutX
	E_Pending	Miss ->Upgrade_Nack	Miss ->GetX _Nack
	E_Held	Hit ->Set_Pending ->GetXFwd	Hit ->Set_Pending ->GetXFwd
S	S	Hit ->M ->Set_Core_Holder ->Upgrade_Ack (Requester) ->Inval_N (Holders)	Hit ->M ->Set_Core_Holder ->PutX_N (Requester) ->Inval_N (Holders)
I	I	Miss ->Upgrade_Nack	Miss ->SDRAM_Load
	I_WB_Flush_Conflict	N/A	Miss ->GetX_Nack
	I_WB_Non_Writable	N/A	Miss ->Coalesce
	I_WB_Writable	N/A	Hit ->Set_Core_Holder ->M ->PutX
	I_WB_Dir_Conflict	N/A	Miss ->WB_flush ->GetX_Nack

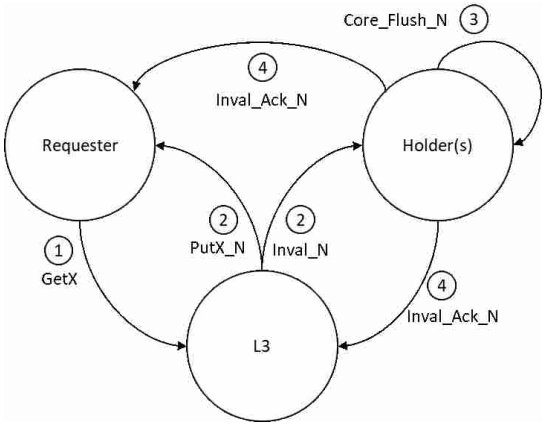
The CPU L3 cache controller will process a fetch request to a memory block that is cached and in the shared state as a hit. Likewise, the cache controller will also process a fetch request to a memory block that is uncached or in the invalid state as a miss resulting in a SDRAM load request being sent to the memory controller. The cache controller will process a load or store to a memory block that is cached and in the modified or exclusive state as a hit. The cache controller will also process a load request to a memory block that is cached and in the shared state as a hit. After processing the hit the cache controller will update the cache line directory with information showing which processor holds the memory block, build the appropriate reply message (PutS, PutClnX, or PutX), and send the reply message to the requesting processor.

The inclusion of the requirement to arbitrate between processors introduces four new exclusive and modified substates. The CPU L3 cache controller will process a load or store to a memory block that is owned by the requesting processor as a regular hit and will reply with the appropriate message (PutClnX or PutX). This occurs when the owning processor drops the memory block and subsequently requests the memory block again from the directory. When the cache controller processes a load, upgrade, or store and finds that the memory block is cached, in the exclusive or modified state, and is held by a processor other than the requesting processor the cache controller sets the coherence pending bit in the directory and then sends a Get/GetX_Fwd message to the holding processor. On a Get_Fwd the holding processor's L2 cache will downgrade the holding processor's memory block to shared and forward the memory block in the shared state to the requesting processor, see Fig. 3.11a. On a GetX_Fwd the holding processor's L2 cache will evict the processor's memory block and forward the memory block in the modified state to the requesting processor. In both cases the holding processor's L2 cache also simultaneously sends an acknowledgement to the CPU L3 cache. In the case of a Get_Fwd that downgrades a modified memory block in a processor the acknowledgement will contain dirty data as a sharing write-back. After receiving either acknowledgement the L3 cache clears the coherence pending bit in the directory

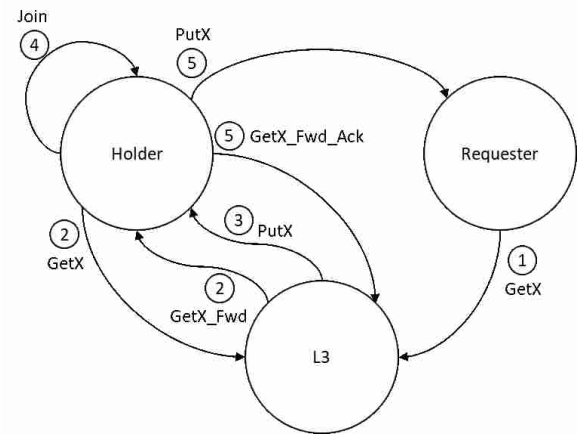
and changes the states of the memory block in its cache table appropriately (shared or modified).



(a) Successful Get/GetX_Fwd



(b) Successful PutX_N



(c) Successful Join

Figure 3.11: Common Network-Based Coherence Protocol Cases

The cache controller will process a subsequent load, upgrade, or store to a memory block in the coherence pending state as a NACK back to the requesting processor. In a special case, it is possible for the CPU L3 cache controller to send a Get/GetX_Fwd message to a holding processor that has simultaneously sent a request to the CPU L3 cache for the subject memory block, see Fig 3.11c. This occurs when the holding processor previously drops the memory block. In this case, the CPU L3 cache receives a memory request from the owning processor for a memory block that is pending

a coherence action (Get/GetX_Fwd). The CPU L3 cache controller will processes the request as a hit and send an appropriate reply message (PutClnX or PutX) back to the owning processor. Once received, the owning processor will exhaust all outstanding requests for the memory block and then join the outstanding Get/GetX_Fwd request from the L3 cache.

A similar mechanism to the Get/GetX_Fwd is used when the CPU L3 cache controller processes an upgrade or store to a memory block that is in the shared state. When processing an upgrade, the L3 cache controller will immediately upgrade the memory block to modified, send invalidation requests to all holding processors other than the requesting processor, update the cache line directory with the requesting processor as the single holding processor, and send an upgrade acknowledgement to the requesting processor. When processing a store, the cache controller will immediately upgrade the memory block to modified, send invalidation requests to all holding processors other than the requesting processor, update the cache line directory with the requesting processor as the single holding processor, and send a PutX_N reply to the requesting processor. In the case of an upgrade the coherence interaction ends when the CPU L3 cache sends the upgrade acknowledgement to the requesting processor. In the case of store, the coherence interaction does not end until both the CPU L3 cache and requesting processor L2 cache receive invalidation acknowledgements from all previously holding processors other than the requesting processor, see Fig. 3.11b. The requesting processor's L2 cache will join the coherence interaction by storing the memory block in the cache table in the modified state after receiving all expected invalidation acknowledgements. In all forward-based protocol cases, if the holding processor no longer holds the requested memory block a NACK will be sent to the L3 cache. The L3 cache will then update the directory and send the block to the requester to complete the memory request.

The CPU L3 cache controller will process a load or store to a memory block that is uncached, not in write-back, or that is in the invalid state as a miss resulting in a SDRAM load request being sent to the memory controller. At the CPU L3 level upgrade requests are NACKED back to the requester if

the memory block has been evicted prior to the arrival of the upgrade. If the cache controller finds that the requested memory block is in write-back the cache controller will update the directory and process the request according to the state of the the cache table set. If a Write-back is flushed by the L3 cache controller it is flushed as a SDRAM Store. If a write-back is waiting on a core flush from the holding processor the L3 cache controller will processes a request as a NACK back to the requesting processor.

3.2.6.4 GPU L1 Vector and L2 Cache MESI Protocol State Tables

Tables 3.6 and 3.7 show the protocol state tables for the GPU L1 vector cache and GPU L2 cache. The initial design choice to use a MESI coherence protocol for the GPU was made because it provides a lightweight and optimized coherence protocol for the GPU in the non-coherent mode of operation, but also bridges the gap between the CPU and GPU by intrinsically interconnecting with the CPU's MESI coherence protocol [45, 46]. In comparison to the CPU, the GPU's data parallel architecture operates as a "stream" processor [47, 48]. In general the GPU runs thousands of fibers in lockstep which results in the GPU attempting to simultaneously pull in large numbers of memory blocks and modify them only to subsequently drop them to then pull in more memory blocks. For performance, this necessitates very high levels of bandwidth in both bringing the memory blocks to the GPU's processors and flushing the memory blocks out of the GPU's processors [49, 50, 51, 52, 53, 54]. The MESI coherence protocol lends itself to this operating environment by simplifying cache design and helps to optimized the GPU's data parallel memory access patterns [55].

The GPU L1 vector cache implementation is straightforward and supports all states of the MESI protocol and in cache merges of write-back data stored in the write-back queue. The vector cache controller will process a load to a cached memory block that is in the modified, exclusive, or shared state as a hit. The vector cache controller will process a store to a cached memory block that is

in the modified or exclusive state as a hit. A memory block in the exclusive state will be set to modified on a hit. The cache controller will process a load or store to memory blocks that are uncached, not in write-back, or that are in the invalid state as a miss resulting in a load (Get) or store (GetX) request being sent down to the next level cache in the memory system. Similarly, the cache controller will also process a store to a cached memory block that is in the shared state as a miss and will send a memory request (GetX) down to the next level cache in the memory system. If the vector cache controller finds that the requested memory block is in write-back the cache controller will update the directory and process the request according to the state of the the cache table set resulting in either a hit or a coalesce of the memory request. GPU L1 scalar cache requests are always processed as a hit by the cache model. This occurs because the data cached by the scalar cache is read only and prepositioned in the cache table prior to running a GPU kernel [56]. Therefore a flat timing is provided for GPU scalar cache accesses.

Table 3.6: GPU L1 Vector Cache MESI Coherence Protocol State Table

GPU Vector Data Cache			
State	**Substate**	**Load**	**Store**
M	M	Hit	Hit
E	E	Hit	Hit ->M
S	S	Hit	Miss ->Invalidate ->GetX
I	I	Miss ->Get	Miss ->GetX
	I_WB_Writable	Hit ->E/M	Hit ->M
	I_WB_Non_Writable	Miss ->Coalesce	Miss ->Coalesce

The GPU L2 cache is required to include additional functionality that makes it more of a hybrid of both the CPU L2 and L3 caches. The GPU L2 cache is directory based because it must arbitrate memory block accesses between GPU compute units and additionally requires the same implementation of the CPU L3 cache's coalescing mechanisms in the cache controller. The GPU L2 cache supports all states of the MESI protocol and in cache merges of write-back data stored in the write-back queue. When processing requests, the GPU L2 cache will set a bit in the directory

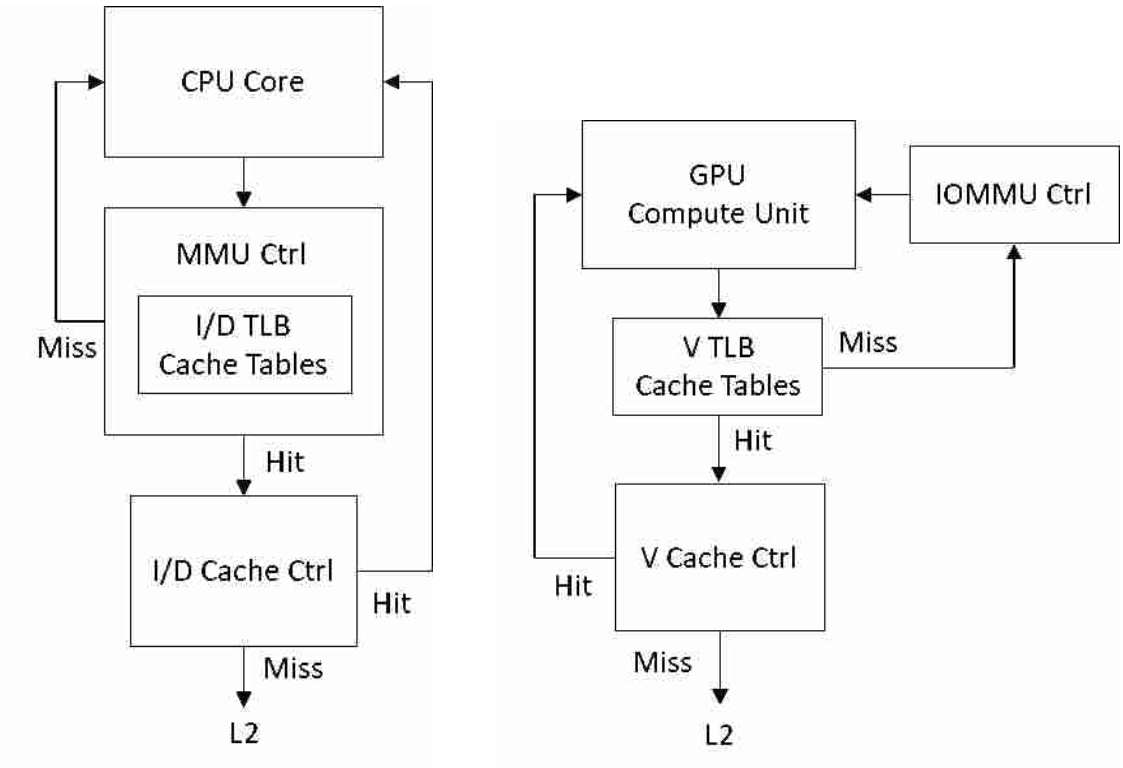
that indicates which GPU compute unit holds the subject memory block. This requires the GPU L2 cache to handle the same modified and exclusive substates as the CPU L3 cache. Furthermore, the GPU L2 cache must help in the completion of coherence related interactions when working collaboratively with the CPU similarly to the operation of the CPU L2 cache.

The GPU L2 cache controller will process a load or store to a memory block that is cached and in the modified, exclusive or shared state in the same manner as the GPU L1 cache controller. Similarly to the CPU L3 cache, the GPU L2 cache controller will update the cache line directory with the holding compute unit, build the appropriate reply message (PutS, PutClnX, or PutX), and send to the requesting compute unit. The cache controller will flush a memory block out of a holding compute unit and then send it to the requesting compute unit as a PutClnX or PutX if the cache controller finds that a load or store request is to a memory block that is cached and held in another compute unit. If the cache controller finds that a load or store request to a memory block that is cached and currently held by the requesting compute unit the cache controller will process as a PutS, PutClnX, or PutX. In a special case, it is possible for the GPU L2 cache controller to processes a core flush to a holding compute unit that has simultaneously sent a request to the GPU L2 cache for the same memory block. In this case the GPU L2 cache receives a request from the owning compute unit for a memory block that is pending the flush. The GPU L2 cache controller then processes the request as a hit, builds the appropriate reply message (PutClnX or PutX), and sends the request back to the owning compute unit. Once received, the owning compute unit will exhaust all outstanding requests for the memory block and then flush the memory block down to the GPU L2 cache. Similarly to the CPU L3 caches, the GPU L2 cache will downgrade blocks at the GPU L1 and L2 level from exclusive or modified to shared when receiving a Get request from an L1 cache. If the memory block is dirty when downgraded the GPU L2 will mark a bit in the cache line signifying that it is dirty in core.

Table 3.7: GPU L2 Cache MESI Coherence Protocol State Table

GPU L2 Cache			
State	**Substate**	**Load**	**Store**
M	M	Hit ->Set_Core_Holder ->PutX	Hit ->Set_Core_Holder ->PutX
	M_Owned	Hit ->PutX	Hit ->PutX
	M_Pending_Owned	Hit ->PutX	Hit ->PutX
	M_Pending	Miss ->Get_Nack	Miss ->GetX_Nack
	M_Held	Miss ->Core_Flush ->Putx	Miss ->Core_Flush ->Putx
E	E	Hit ->Set_Core_Holder ->PutClnX	Hit ->Set_Core_Holder ->PutX
	E_Owned	Hit ->PutClnX	Hit ->PutX
	E_Pending_Owned	Hit ->PutClnX	Hit ->PutX
	E_Pending	Miss ->Get_Nack	Miss ->GetX_Nack
	E_Held	Miss ->Core_Flush ->PutClnX	Miss ->Core_Flush ->PutX
S	S_Clean	Hit ->Set_Core_Holder ->PutS	Miss ->Invalidate ->L3_GetX Or SDRAM_Load
	S_Dirty_In_Node	Hit ->Set_Core_Holder ->PutS	Hit ->M ->Set_Core_Holder ->PutX (Requester) ->Inval (Holders)
	S_Pending_Owned	Miss ->Get_Nack	Miss ->GetX_Nack
	S_Pending	Miss ->Get_Nack	Miss ->GetX_Nack
I	I	Miss ->L3_Get Or SDRAM_Load	Miss ->L3_GetX Or SDRAM_Load
	I_WB_Flush_Conflict	Miss ->Get_Nack	Miss ->GetX_Nack
	I_WB_Non_Writable	Miss ->Coalesce	Miss ->Coalesce
	I_WB_Writable	Hit ->Set_Core_Holder ->PutClnX	Hit ->Set_Core_Holder ->PutClnX
	I_WB_Dir_Conflict	Miss ->WB_flush ->Get_Nack	Miss ->WB_flush ->GetX_Nack

The CPU L2 cache controller will process a load or store to a memory block that is uncached, not in write-back, or that is in the invalid state as a miss. If the LLC is not shared memory requests will be routed directly to the memory controller and if the LLC is shared memory request will be routed to the CPU L3 caches. If the cache controller finds that the requested memory block is in write-back the cache controller will update the directory and process the request according to the state of the the cache table set. If a Write-back is flushed by the L2 cache controller and the LLC is not shared the write-back will be flushed as a SDRAM Store and if the LLC is shared the write-back will be flushed to the CPU L3 cache. The L2 cache controller will processes a request as a NACK back to the requesting compute unit if a write-back is waiting on a core flush by the holding compute unit.



(a) CPU Address Translation Process

(b) GPU Address Translation Process

Figure 3.12: CPU and GPU Virtual Memory Mechanisms

3.2.7 Virtual Memory System

M2S-CGM extends the virtual memory system found in Multi2Sim by incorporating new functionality supporting a CPU-GPU shared memory address space and a GPU specific forward and reverse address translation capability. Additionally, CGM adds timing models for CPU and GPU Translations Lookaside Buffers (TLBs) and page table walkers. Fig. 3.12 shows the architectural makeup of the virtual memory mechanisms employed by both the CPU and GPU and how they interact with their respective processors and the rest of the memory system. Prior to issuing a memory system request each CPU core first consults its Memory Management Unit (MMU) in an effort to make a virtual to physical address translation. The MMU is implemented as a KnightSim context similarly to how a cache is implemented in the memory system. As shown in Fig. 3.13, on advancement the MMU controller probes the virtual address for tag, index, and Physical Page Number (PPN) offset and then consults the TLB cache table. If the MMU controller finds a matching tag in the respective set the entry state and PPN is read out and joined with the PPN offset to form the fully translated physical address. The physical address is then subsequently used by the CPU to issue a memory system request. If the MMU controller finds that the address translation is uncached it is a TLB miss and charges a latency as it raises an exception for the CPU to start the Page Table Walk (PTW).

The PTW is typically performed by the operating system and may or may not be hardware accelerated with use of a PTW cache [57]. Modern operating systems use a four tiered page table and when performing the PTW will make four distinct memory system accesses to resolve the memory page. CGM implements the PTW as a KnightSim context and assumes that the PTW is done in software by the operating system, meaning no hardware based cache for acceleration of the PTW. In an abbreviated simulation approach, after advancement the PTW makes a direct translation of the address in simulation and checks to see if a simulated memory page has been created to support

the translation. If a simulated memory page supporting the address translation does not exist the PTW charges a configurable latency as a hard page fault and if a simulated memory page does exist the PTW charges a configurable latency as a soft page fault. After the address translation completes the PTW caches the translation in the appropriate CPU TLB cache table and returns operation to the guest application for a retry of the address translation.

CGM implements a design approach in support of GPU address translation that is similar to the approach discussed in [58]. In the GPU, TLB caching and page table walking are performed identically to the CPU as discussed above. However, the architectural approach is slightly different. After each GPU vector memory unit coalesces its vector memory accesses the memory access unit then attempts to make a virtual to physical address translation using its private TLB. If successful the physically addressed memory access is passed to the vector cache for servicing. If the address translation is unsuccessful it is a TLB miss that incurs a latency. The GPU's memory access unit then consults with the system's Input-Output Memory Management Unit (IOMMU) for an address translation. The IOMMU performs the PTW and incurs a significant latency, as compared to the CPU. If the IOMMU's PTW finds that the simulated memory page does not exist a latency for a hard page fault is charged and if it finds that the page does exist a latency for a soft page fault is charged. The IOMMU also provides a reverse translation service on behalf of the GPU so that the GPU's caches can be virtually addressed. This provides a benefit in that the caches act to filter memory system accesses until the L2 cache level which results in the TLBs and IOMMU experiencing fewer translation requests from the GPU [59, 60].

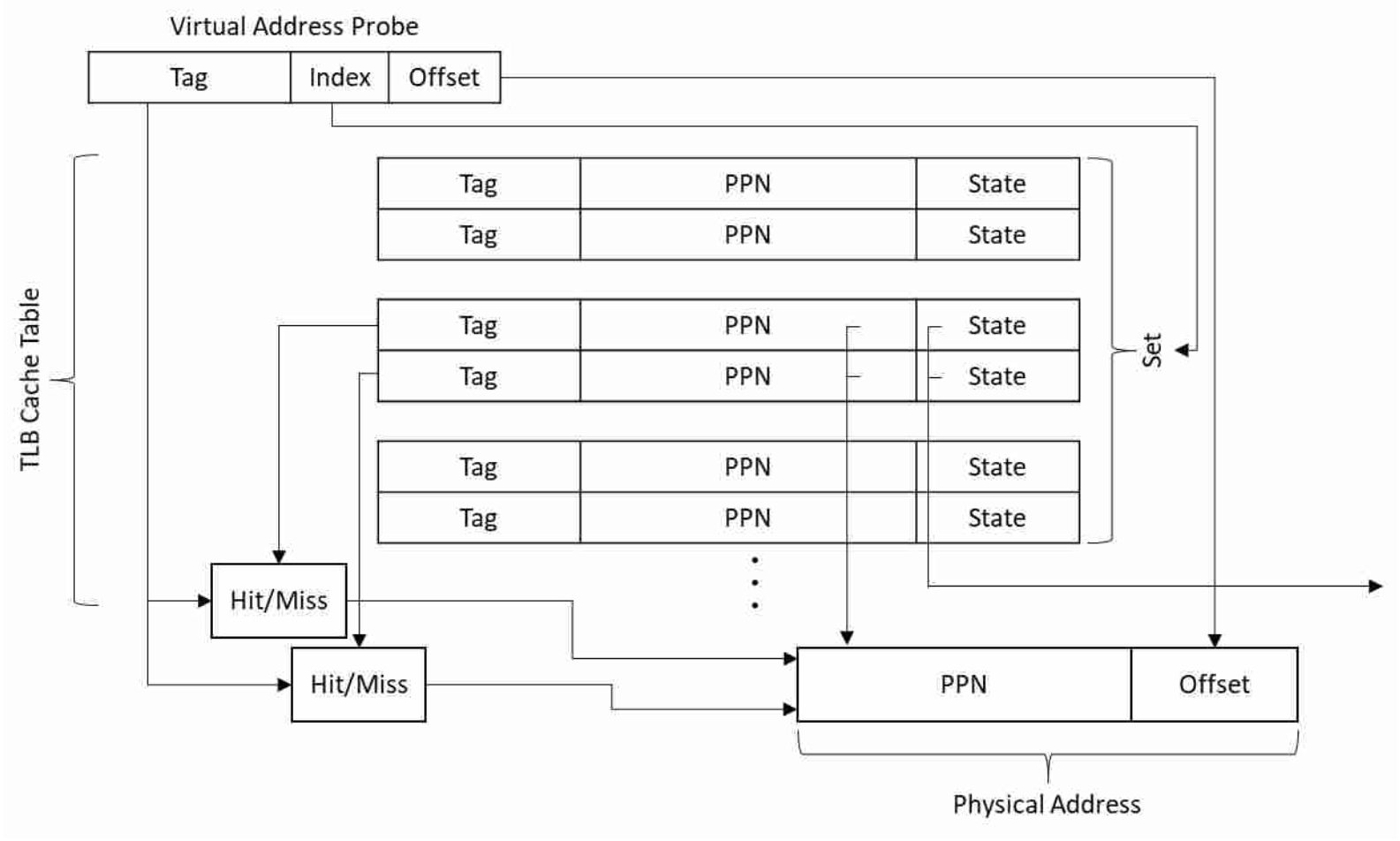


Figure 3.13: Architecture of a Two Way Set Associative TLB Cache

3.3 M2S-CGM Benchmarks

To support the experiments presented in this dissertation I implemented two different sets of benchmarks which I describe in further detail over the following two subsections.

3.3.1 Benchmark Set 1: OpenMP and Non-Collaborative GPU-Only OpenCL Benchmarks

For my first set of benchmarks I extend the Rodinia OpenMP and OpenCL Backprop (BP), Lower Upper Decomposition (LUD), Kmeans, Hotspot, Needleman Wunsch (NW), and Breadth First Search (BFS) benchmarks. Detailed information regarding this set of benchmarks and their related performance metrics can be found in [22, 61, 62, 63]. I chose this set of benchmarks because they provide a good spectrum of processor intercommunication aggressiveness for OpenMP executions and a good spectrum of inter CPU-GPU communication through GPU kernel invocations and memory copies for OpenCL executions. My extensions include making modifications to the non-collaborative GPU-only OpenCL benchmarks so that the benchmarks (1) utilize the M2S-CGM OpenCL runtime and (2) can be set to pass memory objects between the CPU and GPU by pointer.

3.3.2 Benchmark Set 2: Collaborative CPU-GPU OpenCL Benchmarks

For my second set of benchmarks I implement a set of collaborative CPU-GPU benchmarks that are relevant and well-suited for execution on a GPU. The chosen set of benchmarks includes Backpropagation, Block Matrix Multiply, Edge Detection, Nearest Neighbor, and Write. Block Matrix Multiply, Edge Detection, and Write are hand implemented for my experiments and Backpropagation and Nearest Neighbor are ported from the Rodinia benchmark suite [16]. Each benchmark comprises a non-collaborative GPU-only and collaborative CPU-GPU OpenCL implementation and utilizes the M2S-CGM OpenCL runtime. The non-collaborative GPU-only OpenCL version

is used as the point of reference for performance comparisons to the collaborative CPU-GPU OpenCL implementation. My collaborative CPU-GPU OpenCL benchmarks take in arguments that specify the number of CPU threads to use and a CPU workload percentage in addition to other normal benchmark-specific arguments. The following subsections give a brief background for each benchmark and provide detail regarding each benchmark's collaborative CPU-GPU memory system access behavior.

3.3.2.1 Backpropagation

Backpropagation is an algorithm relevant to machine learning applications and is used to train neural networks in two phases. In the first phase, a forward propagation is made which results in an input-driven neuron activation calculation for each neuron in the neural network. In the second phase, a reverse propagation of the resultant error is used to adjust neuron activation weights throughout the neural network. During execution the majority of compute time occurs in the second phase and is where I focus my experimental measurements. The collaborative OpenCL NDRange is divided between the CPU and GPU based on the size of the specified neural network layer, a specified CPU workload percentage, and a specified chunk size. During second phase execution the CPU and GPU both read from arrays containing calculated errors and neuron weights. Then a resultant new weight is stored in the neuron weights array for each neuron. Memory system interactions include the proliferation of cache lines in the shared state between the CPU and GPU on the read of the calculated error and neuron weight arrays. Finally, stores to the calculated weight array lead to a significant number of core evictions and results in a moderately high level of coherence interactions between the CPU and GPU.

3.3.2.2 Block Matrix Multiply

Block Matrix Multiply is an algorithm relevant to linear math and has wide ranging usage in many scientific fields. Block Matrix Multiply has a single calculation phase and kernel where the blocked dot product of two input matrices is calculated and then stored in a solution matrix. The collaborative OpenCL NDRange is divided horizontally between the CPU and GPU based on the number of rows in the input matrices, a specified CPU workload percentage, and a specified chunk size. During execution the CPU and GPU both read from two input matrix arrays, calculate a dot product, and store the result in an output matrix array. Memory system interactions include the proliferation of cache lines in the shared state when the CPU and GPU read from the two input matrix arrays and proliferation of cache lines in the modified state when storing the result. The vast majority of CPU and GPU interactions include the downgrade of cache lines to the shared state on the initial read of the two input matrix arrays and in a few edge cases arbitration of cache lines between the CPU and GPU when storing the result to the output matrix array.

3.3.2.3 Edge Detection

Edge Detection is an algorithm relevant to image processing and computer vision and is often used as an intermediate step in the process of extracting data from an image. Edge Detection has a single calculation phase and kernel where a filter is convolved with an input image that results in isolating the input image's edges. The collaborative OpenCL NDRange is divided horizontally between the CPU and GPU based on the number of rows in the input image array and a specified CPU workload percentage. During execution the CPU and GPU both read from a small array containing the filter's data and an input array containing the input image data. After each pixel has been convolved, the CPU and GPU store the value in the original input image array. Memory system interactions include a short-lived proliferation of cache lines in the shared state when reading from the filter

array and little to no CPU/GPU interactions when reading and storing to the original input image array. In a few edge cases block arbitration between the CPU and GPU occurs when reading and storing to the original input image array.

3.3.2.4 K-Nearest Neighbor

K-Nearest Neighbor is an algorithm relevant to pattern recognition, machine learning, and data mining. K-Nearest Neighbor has a single calculation phase and kernel where K-nearest neighbors are located within an input data set based on a specified position. The collaborative OpenCL NDRange is divided between the CPU and GPU based on the number of records in the input data set, a specified CPU workload percentage, and a specified chunk size. During execution the CPU and GPU both read position data from different locations in a record array, calculate the record's distance to an input position, and store the resultant distance in a record array. This benchmark is a highly data parallel benchmark with no CPU-GPU interactions over the execution of the kernel.

3.3.2.5 Write

The Write benchmark is an algorithm designed with utility in mind. Write comprises a single storage phase and kernel where all elements of an array are assigned a value. Write is intended to be a brutal benchmark in terms of creating high contention and latency in the system and has utility in finding memory system bottlenecks. The collaborative OpenCL NDRange is divided between the CPU and GPU based on the array size and a specified CPU workload percentage. During execution the CPU and GPU both store a value to each element in the array. This is a highly data parallel benchmark, but exhibits a huge demand for bandwidth and low latency. In one edge case block arbitration between the CPU and GPU occurs when the CPU stores near the end of its array segment.

3.4 M2S-CGM Validation Results

As mentioned in Chapter 1, previous work has introduced and established the CPU and GPU models of Multi2Sim [64, 65]. This only leaves the validation of M2S-CGM with its detailed memory system model. The validation experiment is designed to demonstrate the parallelism and correctness of the memory system's timing models, CPU and GPU MESI coherence protocols, and system call timing models. Therefore, the validation of M2S-CGM, is performed by comparison between the results of benchmarks executed on M2S-CGM and the results of benchmarks executed on a physical test system. The following subsections discuss the experimental setup and results.

3.4.1 Experimental Setup

The test system comprises an Intel Core i7-4790K Devil's Canyon Quad-Core CPU and AMD Radeon HD 7990 64 compute unit GPU. M2S-CGM is configured similarly as shown in Fig. 3.2 and is configured to match the test system's hardware profile. M2S-CGM and test system frequencies for the CPU, GPU, and memory system are 4 GHz, 1 GHz, and 2 GHz respectively. For M2S-CGM, the system-wide cache block size is 64B and I assume an 8 byte header on all memory system messages. CPU L1, L2, and L3 cache sizes are configured as 32KB, 256KB, and 2 MB respectively with L3 caches operating in a striped configuration. GPU vector and L2 cache sizes are configured as 16KB and 64KB respectively. Main memory is configured as 4GB of SDRAM operating in a dual channel memory configuration.

The benchmarks chosen for use in the validation comprise the Rodinia OpenMP and OpenCL Backprop (BP), Lower Upper Decomposition (LUD), Kmeans, Hotspot, Needleman Wunsch (NW), and Breadth First Search (BFS) benchmarks as discussed in Sec. 3.3.1. This set of benchmarks was chosen because they provide a good spectrum of processor intercommunication aggressiveness for

OpenMP executions and a good spectrum of inter CPU-GPU communication through GPU kernel invocations and memory copies for OpenCL executions. For all benchmarks measurements are taken across the benchmark's parallel section. For OpenCL benchmarks, the parallel section is defined as the beginning of the first OpenCL-related memory buffer creation and the ending of the parallel section to be the completion of the final memory copy to the host device. Benchmark problem sizes are selected based on the maximum obtainable speedup measured in the simulated system and range from medium to large.

3.4.2 Experiment 1: CPU OpenMP Parallel Performance Results

In validating M2S-CGM, a comparison of measured speedup on the test system to measured speedup on M2S-CGM for the Rodinia OpenMP benchmarks is shown in Fig. 3.14. Additionally, a comparison of heterogeneous-workload percentage breakdown for the Rodinia OpenCL benchmarks is shown in Fig. 3.15. Comparisons of absolute total cycles between the target test system and simulator are not made because the simulated CPU and GPU are generalized and represent a wide range of possible processor configurations. Instead, by correctly modeling and observing system behavioral results conclusions on the influence of system level design changes can be drawn and applied to more than just a single processor's architecture.

Fig. 3.14 shows the measured speedup for 2 and 4 threads for the Rodinia OpenMP benchmarks on the test system and on M2S-CGM. The results show good correlation between the test system and M2S-CGM and highlight expected performance differences. For the OpenMP benchmarks, simulated execution had an average difference of 10.4% for the two threaded runs and 22% for the four threaded runs. These differences are expected and show correct simulation behavior as compared to a physical system that is running many other system processes in addition to the benchmarks themselves. These results also highlight the inherent parallelism and correctness of

the memory system's CPU and GPU MESI coherence protocols.

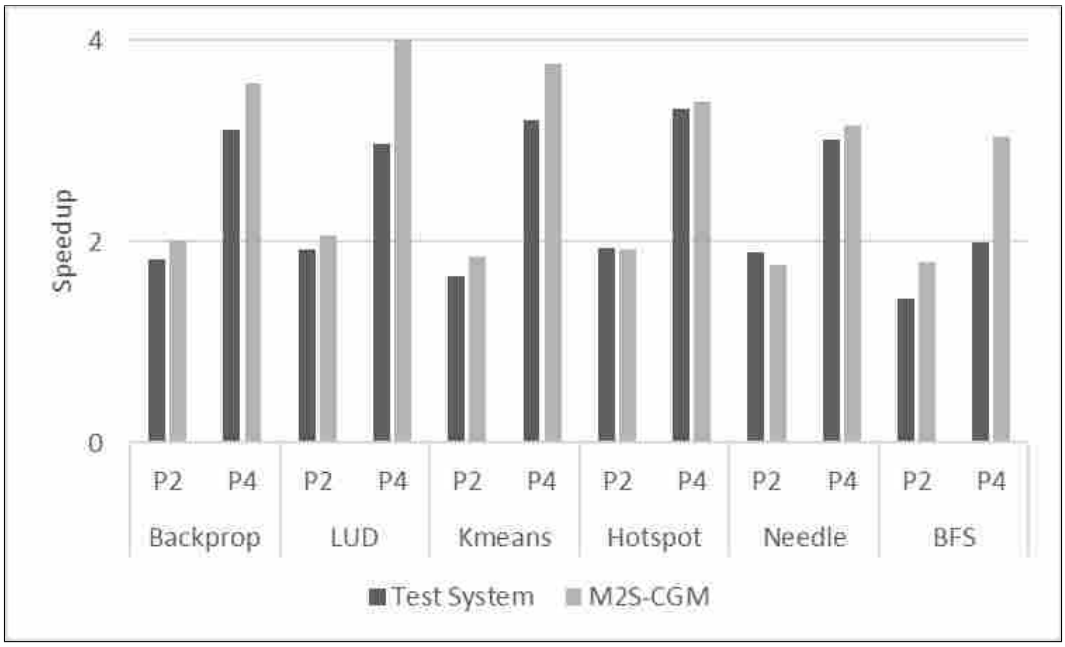


Figure 3.14: Rodinia OpenMP Benchmark Results

3.4.3 Experiment 2: CPU-GPU OpenCL Parallel Performance Results

Results for OpenCL benchmarks are shown in Fig. 3.15. Heterogeneous workload execution time is measured and broken down into GPU kernel time, CPU time, and OpenCL related system call time. In the results CPU time and OpenCL related system call time are combined and denoted as "CPU+SC". Again, the results show close correlation between the test system and M2S-CGM and highlight the delicate simulation of the interplay between the CPU and GPU over the parallel section. For the OpenCL benchmarks, simulated execution time breakdown between the CPU and GPU is within 6.4% on average. We also note that the Rodinia OpenCL benchmarks themselves do not make use of the CPU for processing during the execution of the benchmark. Instead, the CPU only performs setup and management of problem data and GPU execution through a series

of OpenCL calls and ultimately OS system calls. During GPU invocation the CPU goes dormant until the GPU completes and signals the CPU to wake up.

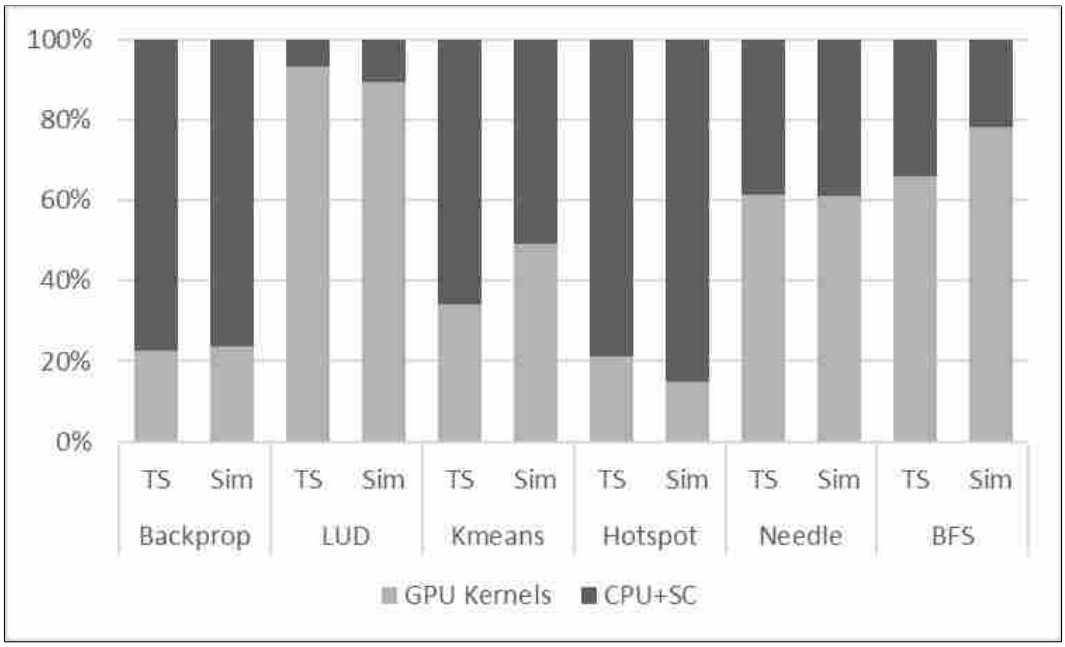


Figure 3.15: Rodinia OpenCL Benchmark Results

Based on the results shown here, and by successful comparison of M2S-CGM's simulated results to the test system, it is established that M2S-CGM provides a valid and realistic multicore and heterogeneous system model and can therefore serve as a strong platform for future heterogeneous system research.

3.5 Summary and Conclusions

In this chapter I introduced M2S-CGM, a detailed architectural simulator that models the interactions between CPUs and GPUs operating in heterogeneous compute environments. I presented the motivation and need for M2S-CGM and provide in-depth details about its software architectural makeup and provided a validation of M2S-CGM's multithreaded and heterogeneous system

simulation capabilities via the comparison of executions of select Rodinia OpenMP and OpenCL Benchmarks on a test system and the simulator. My validation results show that M2S-CGM provides an accurate simulation model of a modern multicore and heterogeneous system with differences ranging from 10.4% and 22% for two and four threaded OpenMP runs and 6.4% for OpenCL runs. The results show that M2S-CGM provides good correlation to modern computing systems and that information ascertained from experimentation is reliable and can be used for trade-off decisions in proposed architectural implementations.

CHAPTER 4: HETEROGENEOUS CPU-GPU SYSTEM ARCHITECTURAL EXPERIMENTS

This chapter presents the results of three heterogeneous CPU-GPU processor and application architectural studies, as first presented in [3, 19], and adds a significant expansion of discussion related to the heterogeneous CPU-GPU programming model. These studies aim to explore the heterogeneous CPU-GPU processor and application design space with the goal of answering interesting research questions, such as, (1) what are the architectural design trade-offs in heterogeneous CPU-GPU processors and (2) how do we best maximize heterogeneous CPU-GPU application performance on a given system. The experiments presented in this chapter are entirely conducted with the use of the M2S-CGM computer architectural simulation system and its compatible benchmarks.

This chapter starts with providing a detailed background regarding the current state-of-the-art in the heterogeneous CPU-GPU programming model. Then, I present the results of three architectural studies. The first set of experiments studies the impacts of added coherency between the CPU and GPU with and without shared LLC. The second set of experiments studies how to determine what the maximum optimization point is in collaborative CPU-GPU applications. The third set of experiments studies the performance impacts of four key architectural changes in heterogeneous CPU-GPU processors. Important observations and take-a-ways from each set of experiments are provided which helps to inform researchers on ways to best optimize collaborative CPU-GPU executions and provides new directions for future research regarding heterogeneous CPU-GPU processor design.

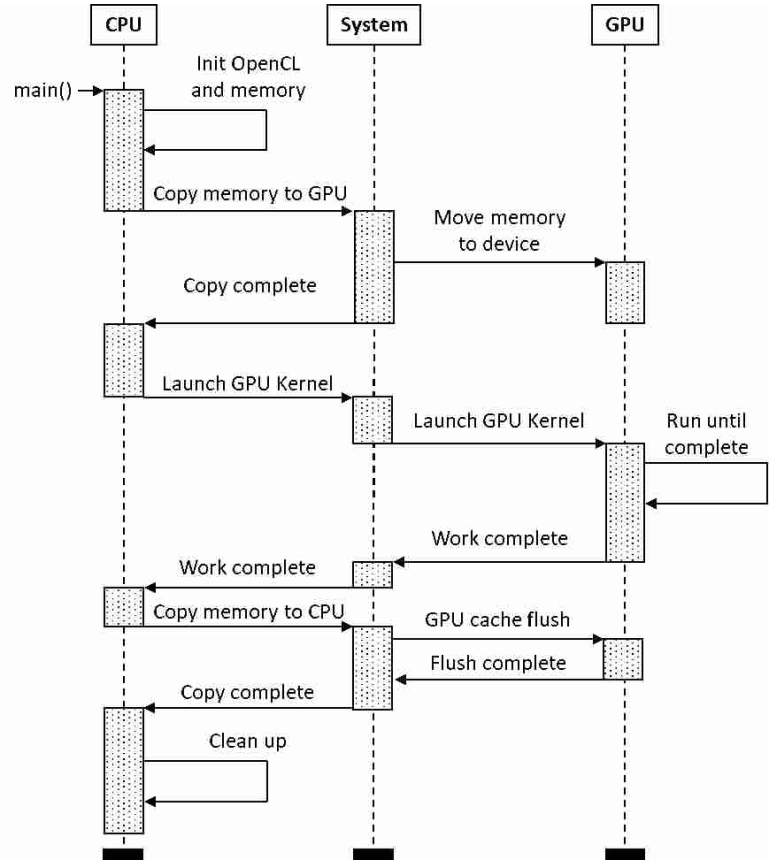


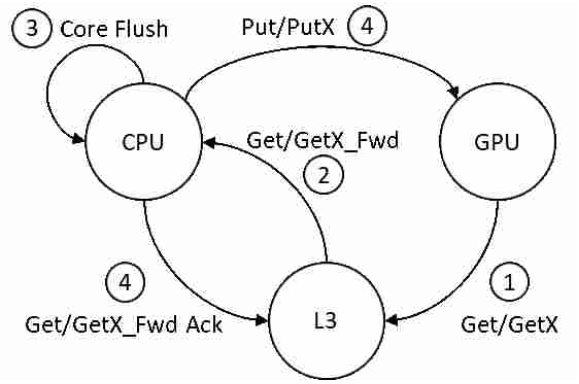
Figure 4.1: Non-Coherent CPU-GPU Execution Sequence

4.1 Background and Motivation

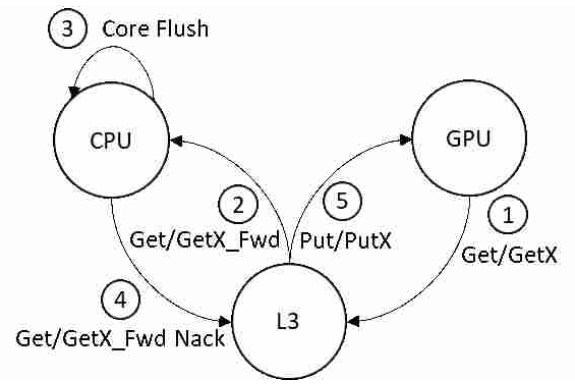
As mentioned in 3.1, the heterogeneous CPU-GPU programming model first debuted circa 2005 and provided a way to utilize discrete graphics card shaders as general purpose devices in an effort to speed up massively parallelizable computations [48]. At the time, heterogeneous CPU-GPU programming was touted as difficult to do, but still continued to draw interest from academia and industry. Since then many improvements have been made towards the generalization of GPU hardware and the overall simplification of the heterogeneous CPU-GPU program model. These improvements have helped to make heterogeneous CPU-GPU programming a mainstream pro-

gramming model. However, despite these recent usability improvements the general approach to executing a heterogeneous CPU-GPU program has remained the same. In the heterogeneous CPU-GPU programming model developers must treat the CPU and GPU as separate devices and must endeavor to partition the execution of the application between available CPU threads and GPU kernels so that the application exhibits an overall gain in performance over the equivalent multi-threaded only version. Figure 4.1 highlights how the CPU and GPU interoperate by showing a *summarized* execution sequence of a typical heterogeneous CPU-GPU application with a single kernel execution on a GPU.

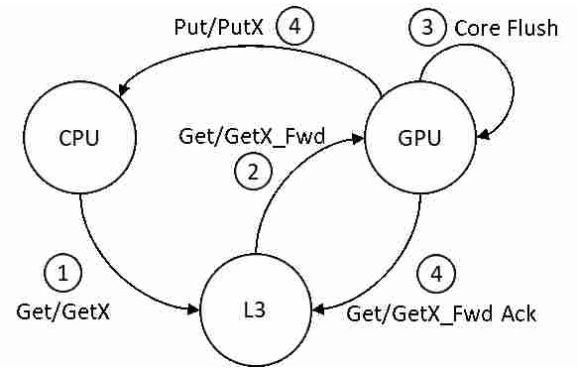
Following the figure from-top-to-bottom. When utilizing either a discrete graphics cards or GPU integrated on-die with the CPU, the application that is running on the CPU must first configure and setup the GPU's execution code and copy all data elements to the GPU prior to the execution of the selected GPU kernel. Then the application must explicitly invoke the GPU's execution of the kernel when programmatically ready to do so. Subsequently, at the end of GPU kernel execution the application running on the CPU must recopy the resultant data back from the GPU's memory hierarchy and address space to the CPU's memory hierarchy and address space so that the CPU can make use of the computed result. This execution schema is accomplished via a series of system calls that invoke several OS, GPU driver, and in some cases direct memory access interactions. This approach is required because the GPU is treated as a physically disparate I/O device, unequal to the CPU, with a different instruction set architecture, memory system structure, and virtual memory address space. It is important to note that the generalized heterogeneous CPU-GPU programming model does not preclude developers from using both the CPU and GPU simultaneously, however, in practice this is not performed much due to the complexities of fine gain workload and data partitioning and the overhead cost of copying data back and forth between the CPU and GPU.



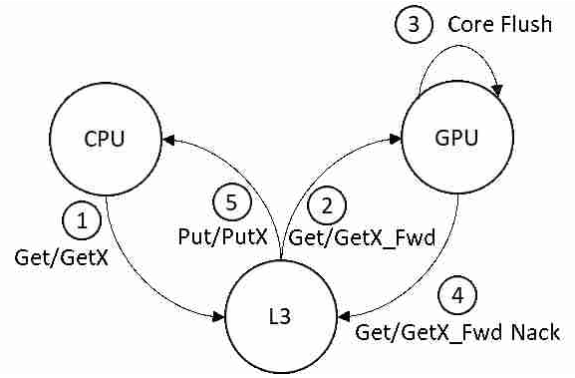
(a) Successful CPU Get/GetX_Fwd



(b) Unsuccessful CPU Get/GetX_Fwd



(c) Successful GPU Get/GetX_Fwd



(d) Unsuccessful GPU Get/GetX_Fwd

Figure 4.2: Coherent Memory Block Movement Between CPU and GPU

When studying the presented heterogeneous CPU-GPU program execution sequence it is apparent that there are several areas where further research could potentially unlock new performance improvements in the heterogeneous CPU-GPU programming model. These areas comprise researching architectural support leading to the reduction of required CPU-GPU system interactions, reduction or elimination of required memory copies, and shared (collaborative) processing of workloads between the CPU and GPU. This insight, forms the basis of the motivation for the research and experimentation presented in this chapter which aims to achieve better performance in heterogeneous CPU-GPU processor and application designs. In studying the heterogeneous CPU-GPU processor and application design trade space more cohesive systems that make better

use of all processing components can be achieved. The results of which would be beneficial in the development of scientific applications by providing higher levels of parallel performance that previously would be unobtainable.

4.2 Coherent Heterogeneous CPU-GPU System Implementation Methodology

This section builds upon the discussion of the M2S-CGM implementation methodology provided in Sec. 3.2 by presenting additional detail regarding how M2S-CGM simulates coherent interoperation between the CPU and GPU. Simulating coherent interoperation between the CPU and GPU concerns the entire software stack and the underlying hardware itself. The software stack comprises the simulated operating system, GPU driver, OpenCL runtime, and OpenCL application (i.e. the benchmark). At the software level the heterogeneous CPU-GPU programming model must be changed to take advantage of the shared virtual address space between the CPU and GPU. This requires changes to the simulated interoperation between the CPU and GPU. First, operating system and GPU driver memory management is changed for the GPU. When the CPU and GPU are made coherent data is no longer copied to and from the GPU's memory address space. Therefore, the operating system and GPU driver are configured to pass memory between the CPU and GPU by pointer. Second, heterogeneous CPU-GPU applications and the OpenCL runtime must be configured to support the operating system and GPU driver changes. The OpenCL runtime is updated to reflect the changes in the programming model and the application is configured to use the modified OpenCL runtime. However, the CPU is still required to perform all memory allocations and initialization prior to GPU kernel executions. The GPU is then passed pointers to memory in the CPU's address space as a part of device setup.

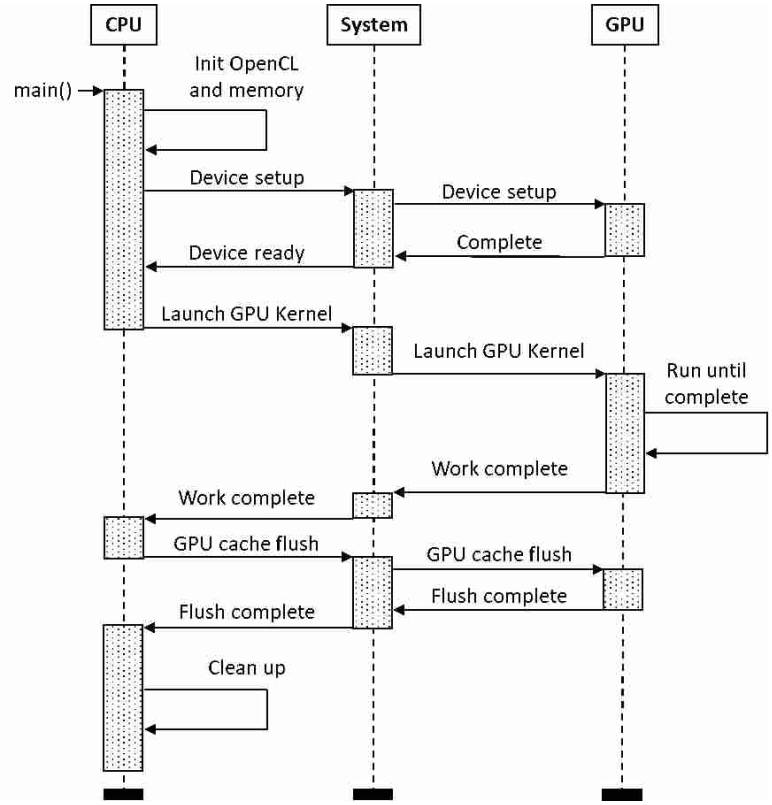


Figure 4.3: Coherent Non-Collaborative Without Shared LLC CPU-GPU Execution Sequence

There are several key hardware changes required to enable the coherent CPU-GPU environment. First, memory system coherency between the CPU and GPU is accomplished by extending the memory system's coherence directories to support servicing the GPU as an "Nth" core. This comprises the inclusion of an additional bit in the directory that represents the GPU and the required directory controller functionality to handle memory request and other coherence related messages to and from the GPU at the directory. Additionally, the CPU and GPU L2 caches are extended to support coherence protocol forwarding request (3-way hops) from the the L3 caches (directory) to a requesting CPU or GPU L2 cache and to support coherence related joins as needed. As discussed in Sec. 3.2, despite having different MESI coherence protocol implementations, the CPU and GPU intrinsically interface well because they both support all MESI cache line states.

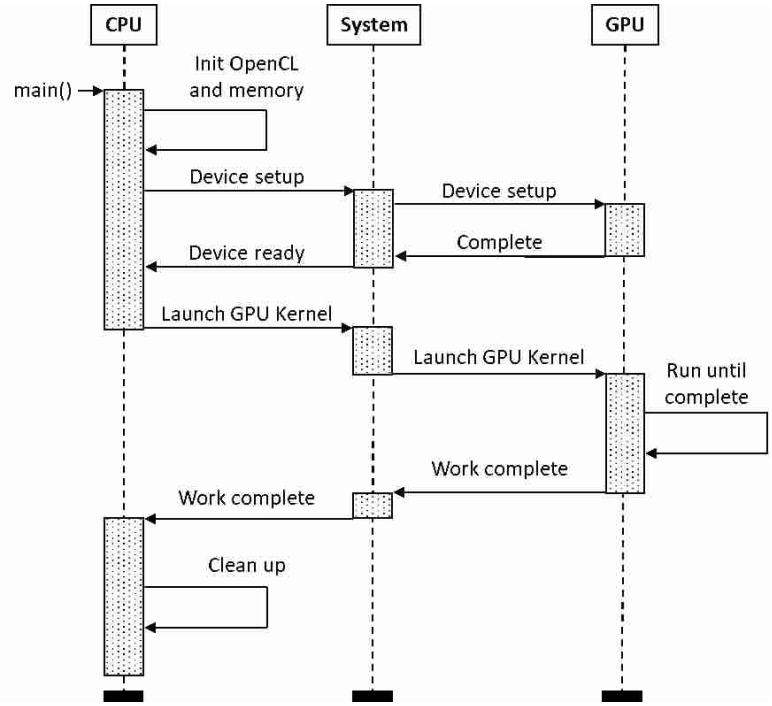


Figure 4.4: Coherent Non-Collaborative With Shared LLC CPU-GPU Execution Sequence

M2S-CGM implements a fine grained approach to integrating the MESI coherence protocols of the CPU and GPU and uses the memory block forwarding schema as the coherence mechanism between the CPU and GPU. Fig. 4.2 shows the process of memory block movement between the CPU and GPU. In essence the entire GPU is treated as a single additional CPU core. This design choice was made taking into account the streaming nature of the GPU. To move a memory block the CPU or GPU first initiates a Get/GetX request to the appropriate L3 cache bank. When consulting the appropriate cache line directory if the L3 cache determines that the memory block is held by a different CPU core or the GPU on whole it will then forward the request to the appropriate CPU or GPU L2 cache. Once received by the L2 cache the holding CPU core or GPU compute unit is either flushed or downgraded as required. Once complete, the L2 cache simultaneously forwards the memory block to the requesting CPU or GPU L2 cache and replies to the L3 cache with an acknowledgement. The forwarded request is unsuccessful if the CPU or GPU L2 cache finds that

the block is no longer in the CPU or GPU. This can occur if the CPU core or GPU has dropped the memory block and results in a NACK sent to the L3 cache. The L3 cache then must update the cache line directory and complete the coherence interaction with a Put/PutX back to the requesting CPU core or GPU.

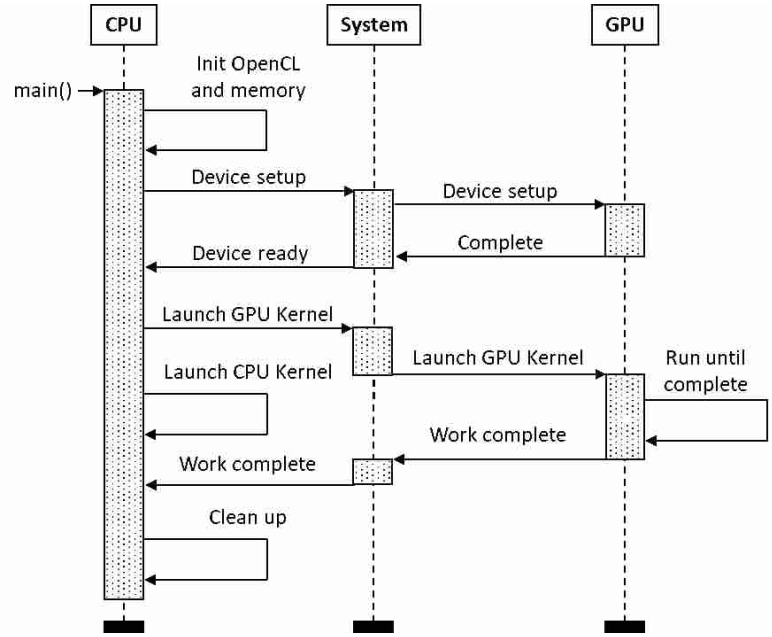


Figure 4.5: Collaborative CPU-GPU Processing Execution Sequence

The functional hardware mechanisms necessary to support virtual memory are already built into the GPU's memory system. This includes GPU specific TLBs, an IOMMU, a PTW, and optionally forward and reverse memory address translation functionality that is incorporated into the GPU hub. I assume that GPU address translation occurs within the IOMMU on a GPU compute unit TLB miss and then by the CPU on an IOMMU PTW miss, as presented in [58]. M2S-CGM's simulated page tables record multiple memory address spaces and page types (i.e. .text, .data, and .gpu) to support separate and combined CPU and GPU virtual address spaces. When sharing an address space the CPU and GPU both resolve pointer address translations to the same physical page in memory which results in the desired equivalent virtual to physical address translations.

The implementation details discussed in this section result in allowing M2S-CGM to support the following four CPU-GPU system wide configurations:

- A traditional GPGPU configuration with the CPU and GPU operating in disparate virtual address spaces and disparate memory systems.
- A modified traditional GPGPU configuration with the CPU and GPU operating in disparate virtual address spaces, but with shared lower level caches.
- A half CPU-GPU heterogeneous configuration with the CPU and GPU operating in a shared virtual address space, but disparate memory system.
- A full CPU-GPU heterogeneous configuration with the CPU and GPU operating in both a shared virtual address space and with shared lower level caches.

The traditional and modified GPGPU configurations serve as the heterogeneous execution baseline and model a modern non-coherent heterogeneous CPU-GPU processor. Note that the first configuration implements the CPU-GPU execution sequence previously shown in Fig. 4.1 and was used in the validation of M2S-CGM, see Sec. 3.4. As discussed in Sec. 4.1, in the traditional and modified traditional configurations the GPU is treated as a disparate device from the CPU and must work in its own address space. This requires the CPU to copy memory back and forth to and from the GPU prior to and after kernel executions through a series of OS systemcalls and GPU driver executions. Additionally, the OS must manage the state of the memory system, and ensure that CPU and GPU caches are fully flushed to main memory prior to a memory system read at the beginning and end of GPU kernel executions. In the modified GPGPU configuration the memory system is set to share the L3 caches between the CPU and GPU which effectively gives the GPU a larger and faster third cache level, however, the CPU and GPU use different physical addresses when accessing data which still requires cache flushes.

In the half and full coherent configurations the CPU and GPU coherently operate in a shared memory space and access the same physically addressed data. The CPU-GPU execution sequence implemented in the half configuration is shown in Fig. 4.3. In this implementation CPU and GPU memory requests join at the memory controller which means that the data must be completely flushed to main memory before either processor accesses the other's data. This is shown in the figure as a required CPU-GPU interaction occurring before the CPU can make use of the GPU's computed data. The CPU-GPU execution sequence implemented in the full configuration is shown in Fig. 4.4. In the full configuration sharing the LLC results in cache flushes being no longer compulsory because the directory arbitrates accesses and forwards memory block requests between the CPU and GPU as required. Therefore, in a non-collaborative execution the CPU can simply issue memory requests without needing to issue a GPU cache flush. Fig. 4.5 depicts the same CPU-GPU execution sequence, however in terms of a collaborative CPU-GPU execution. This is shown in the figure as the CPU an GPU simultaneously executing a kernel. In this approach the CPU and GPU can take advantage of the coherency, shared LLC, and shared address space and speed up overall application execution over executing the workload on the GPU only.

The rest of this chapter presents the results of three sets of experiments conducted with the goal of determining what the architectural design trade-offs in heterogeneous CPU-GPU processors supporting CPU-GPU coherency, shared LLC, and shared virtual memory address spaces are and how to best maximize heterogeneous CPU-GPU application performance in these types of systems.

4.3 Study 1: Architectural Affects of Shared LLC and CPU-GPU Coherence On Non-Collaborative GPU-Only Execution Performance

This section presents experiments and results regarding an architectural study with the goal of reducing interoperation overhead in non-collaborative GPU-only applications. The experimental

results presented in this section show the impacts of the removal of the requirement for CPU-GPU memory copies and shows the impacts of sharing the LLC between the CPU and GPU in coherent heterogeneous CPU-GPU systems.

4.3.1 Experimental Setup

For the conduct of the experiments presented in this section, I maintain the computer architectural system wide configuration parameters, Rodinia OpenCL Benchmarks, and OpenCL parallel section definition as outlined in Sec. 3.4.1. I then execute each benchmark and vary M2S-CGM's CPU-GPU coherency and shared LLC architectural configuration parameters.

4.3.2 Experimental Results

Experimental results for the selected Rodinia OpenCL Benchmark executions on each system configuration are shown in Fig. 4.6. In the figure, "NC" and "C" stand for noncoherent and coherent and "MC" and "L3" stand for memory controller and L3 cache. "MC" and "L3" represent the GPU memory request destination when entering the memory system from the GPU.

For each OpenCL benchmark all results are normalized to the benchmark's NC-MC configuration results. The results show execution breakdowns for CPU Busy, CPU Stall, GPU Busy, GPU Stall, and System Time in percentage of cycles. CPU and GPU busy time is the time the CPU and GPU performed work over the parallel section. CPU and GPU stall time is the time the CPU and GPU were stalled while waiting on outstanding memory system requests. System time is the time spent trapped to the OS while performing an OpenCL related system call or memory system related CPU interrupt, such as a cache flush. The results show that the modeled half coherent CPU-GPU heterogeneous system achieves speedups of 3.27x, 1.06x, 0.94x, 6.51x, 1.21x, and 1.19x and that

the fully coherent CPU-GPU heterogeneous system achieves speedups of 3.67x, 1.06x, 0.95x, 8.83x, 1.23x, and 1.40x for Backprop, LUD, Kmeans, Hotspot, Needleman, and BFS respectively.

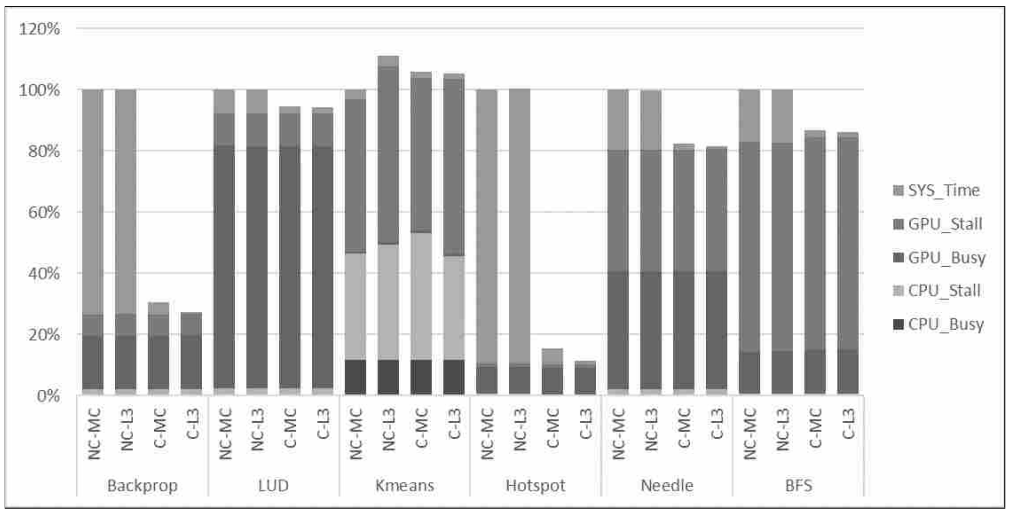


Figure 4.6: Coherent Non-Collaborative Experimental Results

The measured system time comprises all of the overhead associated with executing workloads in the CPU-GPU heterogeneous environment and includes all of the intercommunication required between the CPU and GPU. From comparison of the results between noncoherent and coherent executions, it is apparent that the extent to which speedup is achievable is dependent on the amount of overhead incurred by the application. Thus applications that require significant CPU intervention, like Backprop and Hotspot, see significant improvements and others, like LUD, Needleman, and BFS see slight improvement. However, all benchmarks with the exception of Kmeans did show improvement in speedup. Kmeans is an exception because the parallel section in Kmeans contains a significant amount of CPU setup between kernel executions which effectively defeats the purpose of parallelizing the benchmark over the GPU. This is evident in the nearly negligible GPU busy time for Kmeans.

CPU and GPU busy and stall time show the benchmark's utilization of the CPU and GPU together.

The native Rodinia OpenCL benchmark's are non-collaborative benchmarks and allocate no work to the CPU during GPU Kernel execution. This means that the current Rodinia OpenCL Benchmarks do not fully exploit the complete level of parallelism available between the CPU and GPU. As shown in the results, the Rodinia OpenCL benchmarks effectively delegate all processing to the GPU while the CPU remains idle and is relegated to only performing coordinating and setup functions between GPU kernel executions. In an ideal system the CPU would effectively share 50% of the execution time with the GPU. Performance gains in these measurements rely on better utilization of the CPU and GPU. This topic will be explored in greater extent in the next set of experiments. Despite this, CPU and GPU busy time remains consistent between configuration experiments which is expected and correct.

In the noncoherent configuration the results show that sharing the L3 caches between the CPU and GPU is ineffective and actually slightly hurts overall performance. This is due to the streaming nature of the GPU and low temporal reuse of memory system blocks. Thus, forwarding memory access requests to the L3 caches resulted in a predominance of L3 cache misses, which are then subsequently sent to the memory controller. The results suggest that in a noncoherent configuration it is better to directly forward the memory request to the memory controller and bypass the latency of the L3 cache access. However, in the heterogeneous configuration shared L3 caches can provide a measurable performance boost. This is due to the coherence protocol, where GPU cache flushes are no longer required on account of the supported LLC directory request forwarding between the CPU and GPU.

4.4 Study 2: Optimizing Collaborative CPU-GPU Execution Performance

This section presents experiments and results regarding an architectural study with the goal of profiling and analyzing collaborative CPU-GPU benchmarks. The analysis helps to determine how

best to optimize collaborative CPU-GPU applications for maximum collaborative performance. The results identify how to achieve each benchmark's maximum optimization point and provides a few rules of thumb regarding how best to reason about what an unprofiled collaborative CPU-GPU application's optimized settings should be.

4.4.1 Experimental Setup

To support the conduct of this set of experiments I configure M2S-CGM for the simulation of the modern heterogeneous CPU-GPU processor architecture shown in Fig. 3.2. The architectural model comprises a multicore out-of-order general purpose x86 CPU, a multi-compute unit in-order AMD Southern Islands GPU, a set of LLCs, switching fabric, system agent, multi-channel memory controller, and DRAM. In addition, the architectural model supports coherency, shared LLC, and shared virtual memory address spaces between the CPU and GPU. A detailed list of pertinent system configuration settings is provided in Table 4.1. I believe that this simulated architectural model and its detailed configuration is a reasonable estimate of current modern heterogeneous CPU-GPU processor designs and allows me to make future looking system configuration changes.

In the architectural model each CPU core connects to the memory system via a set of private L1 instruction, L1 data, and L2 caches. The GPU includes private local data share, scalar, and vector caches for each compute unit and one L2 cache for every four compute units. The GPU's L2 caches are set in a striped configuration and are shared among all of the GPU's vector caches. Each compute unit's vector cache connects to each of the GPU's L2 caches by a shared crossbar. The GPU's L2 caches connect to the memory system through a GPU hub/IOMMU. The GPU hub/IOMMU connects the GPU to the external memory system and multiplexes memory system messages going in-and-out of the GPU L2 caches. The CPU, GPU, LLC, system agent, and memory controller are connected together by a switching fabric configured in a ring topology. The LLCs are set in a

striped configuration and are shared among the CPU and GPU with directory and request forwarding (three way hop) block arbitration between the CPU and GPU. Both the CPU and GPU continue to utilize their MESI coherence protocols. The GPU is positioned close to the system agent and memory controller so that significant changes in bandwidth between the GPU and memory controller can be made without directly impacting the rest of the switching fabric and the CPU. In this design approach it is not unreasonable for the on-die GPU to enjoy much higher bandwidth than in previous generations of heterogeneous CPU-GPU processors [32].

I utilize the set of benchmarks presented in Sec. 3.3.2 and take performance measurements across the benchmark's parallel section. For both the non-collaborative GPU-only and collaborative CPU-GPU versions of each benchmark I define the start of the parallel section to be immediately before the first call to `clSetKernelArg()` and the end of the parallel section to be immediately following the join after `clFinish()`. I select benchmark problem sizes based on the maximum obtainable speedup in my simulated system where problem sizes range from medium to large. Additionally, I verified that the non-collaborative GPU-only and collaborative CPU-GPU implementations of each benchmark are equivalent to each other by setting the collaborative CPU-GPU benchmark's CPU workload percentage to zero, executing both versions, and observing that both versions have the same execution time. For the experiment I execute each benchmark and vary CPU cores/threads from one to eight and CPU workload percentage from 20% to 80%.

Table 4.1: Initial System Configuration

CPU	
Core	8 cores, 4GHz, 4 inst/cyc, out-of-order, 64 entry ROB
L1 I/D Cache	2-way, 32KB, private, 1 cyc latency, MSHR 16, coalescer 64, 32GB/s
L2 Cache	4-way, 256KB, private, 2 cyc latency, MSHR 16, coalescer 64, 32GB/s
L3	16-way, 2MB, shared, striped, 4 cyc latency, MSHR 16, coalescer 64, 32GB/s
GPU	
Core	8 CUs, 1GHz, 16-wide SIMD, 64 wavefronts, round robin scheduling
L1 Cache	4-way, 16KB, private, 1 cyc latency, MSHR 16, coalescer 32, 32GB/s
L2 Cache	16-way, 64KB, private, striped, 3 cyc latency, MSHR 16, coalescer 64, 32GB/s
Hub	1 cyc latency, 32GB/s I/O
Uncore	
Switches	2GHz, 4-port, round robin scheduling, ring topology, 32GB/s
SA	2GHz, 2 cyc latency, 32GB/s I/O
MC	2GHz, 2 cyc latency, 32GB/s I/O
DRAM	4GB, 8 channels, striped
Global	
64B line sizes, 8B headers, LRU replacement policy	

4.4.2 Experimental Results

The results for my collaborative CPU-GPU executions are shown in Figs. 4.7, 4.8, 4.9, 4.10, and 4.11. For this first set of experiments I configure M2S-CGM as discussed in Sec. 4.4.1 and execute both the non-collaborative GPU-only and collaborative CPU-GPU implementations of each of my benchmarks. The non-collaborative GPU-only results are shown as "GPU" on each of the graphs and are used as the point of comparison for the collaborative GPU-GPU executions. I ex-

ecute the collaborative CPU-GPU benchmarks while varying the number of CPU cores/threads from one to eight and varying CPU workload percentage from 20% to 80%. All results shown are normalized to those of the appropriate non-collaborative GPU-only benchmark execution results. Smaller is better in the graphs. The graphs provide a performance profile for each benchmark by showing overall measured execution time for each benchmark's configuration and showing how much of the overall time the CPU and GPU were executing. CPU and GPU time are shown as two separate overlaid line graphs. A difference between measured CPU time and GPU time tells us that the benchmark's execution was unbalanced and gives us the CPU or GPU idle time. This means that either the CPU or GPU finished executing early and then sat idle waiting for the other processor to complete its work. The benchmark's execution is therefore balanced when CPU time and GPU time intersect on the graph.

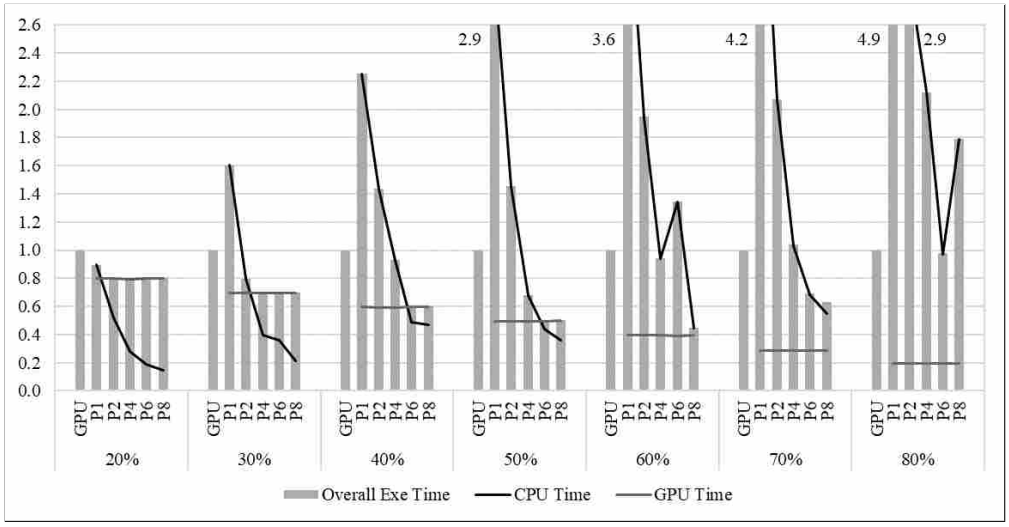


Figure 4.7: Backprop Collaborative Execution Profile

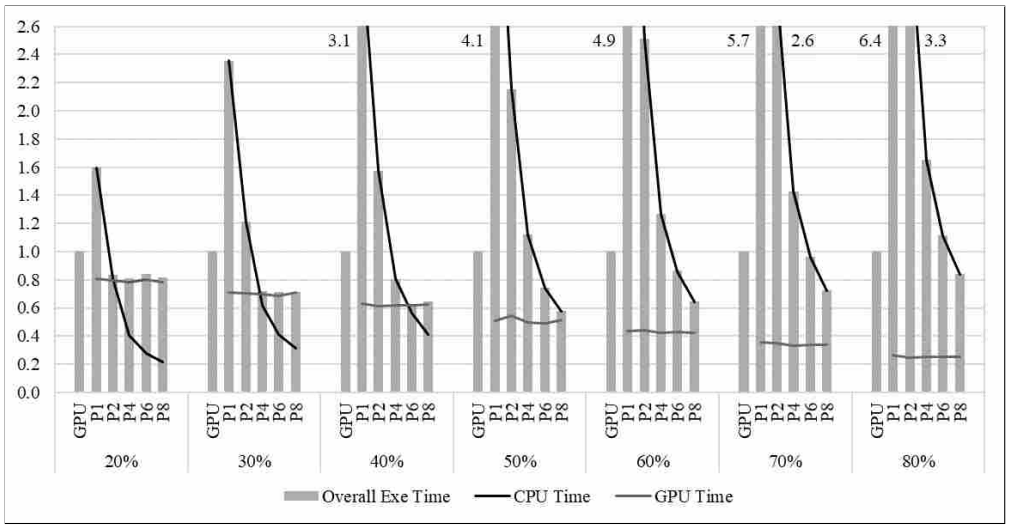


Figure 4.8: Block Matrix Multiply Collaborative Execution Profile

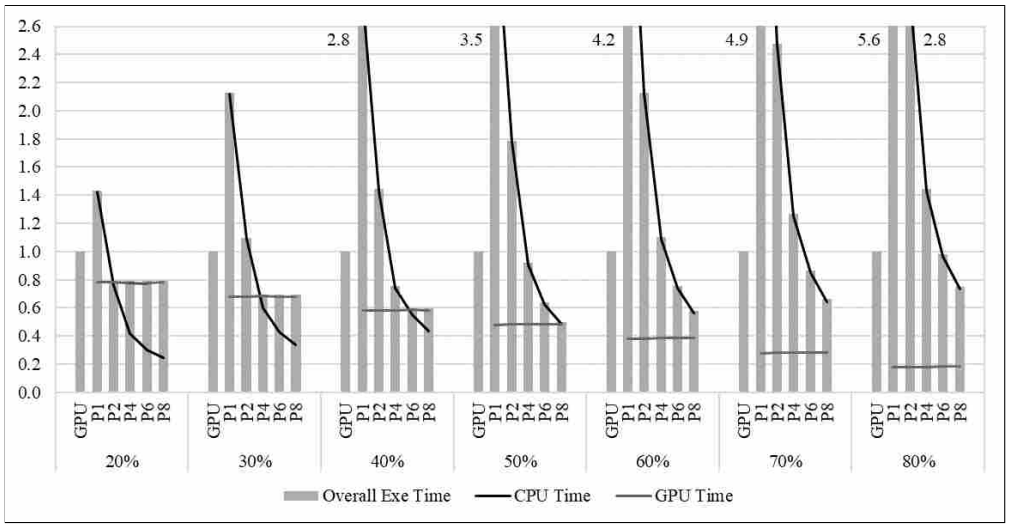


Figure 4.9: Edge Detection Collaborative Execution Profile

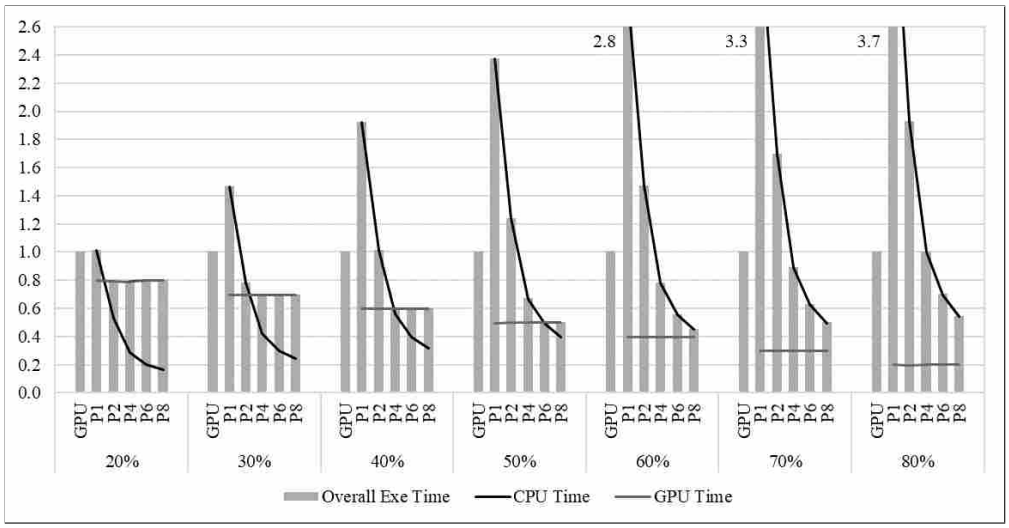


Figure 4.10: Nearest Neighbor Collaborative Execution Profile

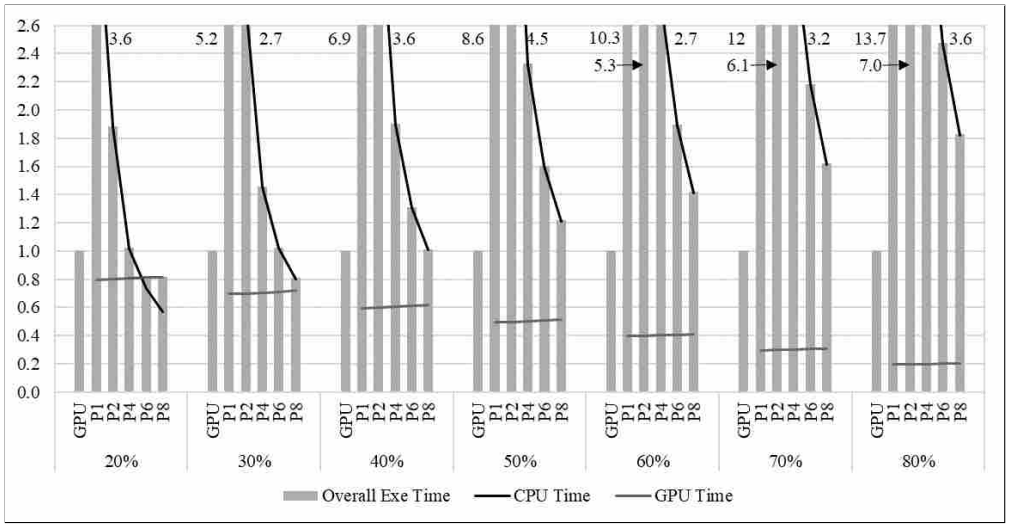


Figure 4.11: Write Collaborative Execution Profile

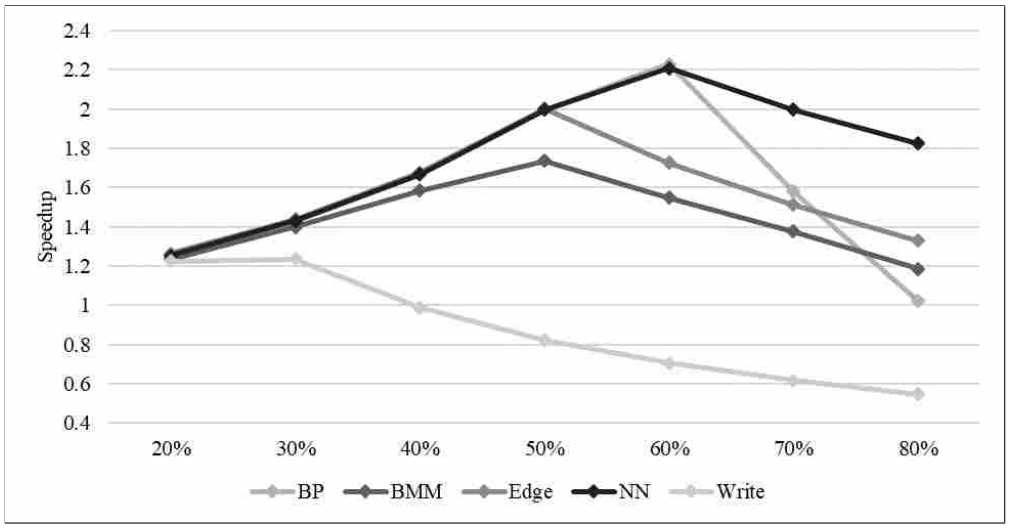


Figure 4.12: Best Case Overall Speedups

First some global observations on collaborative CPU-GPU execution behaviour. The graphs demonstrate that overall collaborative performance, for any of the benchmarks, trends towards becoming increasingly unbalanced as the CPU's workload percentage increases from 20% - 80%. This holds true no matter what number of CPU cores/threads are utilized. I also observe that, with the exception of Write, there exists multiple benchmark configurations resulting in balanced executions. However, a balanced collaborative CPU-GPU execution does not alone mean it is the optimal configuration leading to maximum performance. For example, as shown in the graph for Nearest Neighbor balanced configurations exists in the 20%, 30%, 40%, 50%, and 60% configurations, however, only one configuration, 60%, provides maximum execution performance. All together this means that the combination of using too little or too much CPU workload percentage and using too few or too many CPU cores/threads lead to performance degradation. This makes predicting an unprofiled collaborative CPU-GPU application's optimal configuration a nontrivial task.

For this set of benchmarks specifically, the graphs suggest that (1) giving the CPU more than 60% of the workload percentage will consistently lead to unbalanced collaborative executions and re-

sult in overall degradation in maximum performance, (2) with the exception of Write, giving the CPU less than 40% of the workload will lead to potential performance loss despite the existence of balanced configurations in lower CPU workload percentage ranges, and (3) again with the exception of Write, the sweet spot appears to be giving the CPU 40%-60% of the workload and allocating 6-8 CPU cores to process the workload in parallel. This configuration range leads to the highest in performance gains in our heterogeneous CPU-GPU architectural model. It is important to point out that in most cases performance does increase even in unoptimized collaborative execution and means that even unoptimized collaborative executions are still better than the equivalent non-collaborative GPU-only execution. In the following set of experiments I will show how architectural changes can change these observations.

Table 4.2: Optimal Collaborative Benchmark Configurations

Benchmark	**CPU Cores**	**CPU Share**
Backprop	8	60%
BMM	8	50%
Edge Detection	8	50%
Nearest Neighbor	8	60%
Write	6	20%

Fig. 4.12 summarizes all of the results as the selection of the best case overall speedups under each benchmark configuration. Overall, Backpropagation, Block Matrix Multiply, Edge Detection, Nearest Neighbor, and Write exhibit average speedups over the non-collaborative GPU-only execution of 1.52x, 1.36x, 1.47x, 1.69x, and 0.86x respectively. However, I can observe from the graph that each benchmark exhibits a single point of maximum performance that is significantly better than the average. Table 4.2 summarizes the configuration settings that result in my maximum performance measurements. For each benchmark I measured maximum performance gains over the non-collaborative GPU-only execution of 2.23x, 1.74x, 2.0x, 2.2x, and 1.24x for Backpropagation, Block Matrix Multiply, Edge Detection, Nearest Neighbor, and Write respectively.

The first and second set of experiments demonstrates M2S-CGM's capability and flexibility to successfully simulate various non-collaborative GPU-only and collaborative CPU-GPU executions. Additionally, the benchmarks presented in this dissertation have successfully allowed me to explore the trade space regarding optimizing collaborative CPU-GPU execution. The results shown here are promising and show that if collaborative CPU-GPU applications are appropriately configured, it is evident that collaborative CPU-GPU executions can outperform their non-collaborative counterparts by as much as 2.23x. In the following set of experiments I present a study on how varying key architectural parameters affect the CPU and GPU during collaborative executions and observe the resultant impacts to collaborative CPU-GPU performance.

4.5 Study 3: Future Architectural Impacts to Collaborative CPU-GPU Execution

This section presents experiments and results regarding an architectural study with the goal of determining the impact of varying four key architectural parameters on collaborative CPU-GPU performance by varying GPU compute unit coalesce size, GPU to memory controller bandwidth, GPU frequency, and system wide switching fabric latency. The analysis shows what types of architectural changes are critical to heterogeneous CPU-GPU processor performance and provides new directions for future research regarding heterogeneous CPU-GPU processor design and profiling tools meant to aid in predicting optimal collaborative CPU-GPU configurations.

4.5.1 Experimental Setup

For the conduct of the experiments presented in this section I maintain the computer architectural system wide configuration parameters, benchmarks, and OpenCL parallel section definition as outlined in Sec. 4.4.1. I then vary the M2S-CGM architectural model's GPU compute unit coalescer

size from 32 to 96, vary GPU to memory controller bandwidth from 32GB/s to 76GB/s and then to 288GB/s, vary GPU frequency from 1GHz to 4GHz, and vary latency within the switching fabric.

4.5.2 Experimental Results

The results for my future architectural impacts study are shown in Figs. 4.13, 4.14, 4.15, and 4.16. The graphs are set up in the same manner as discussed in Sec. 4.4.2. However, all results shown here are now normalized to each benchmark's best case execution and not to that of the non-collaborative GPU-only execution.

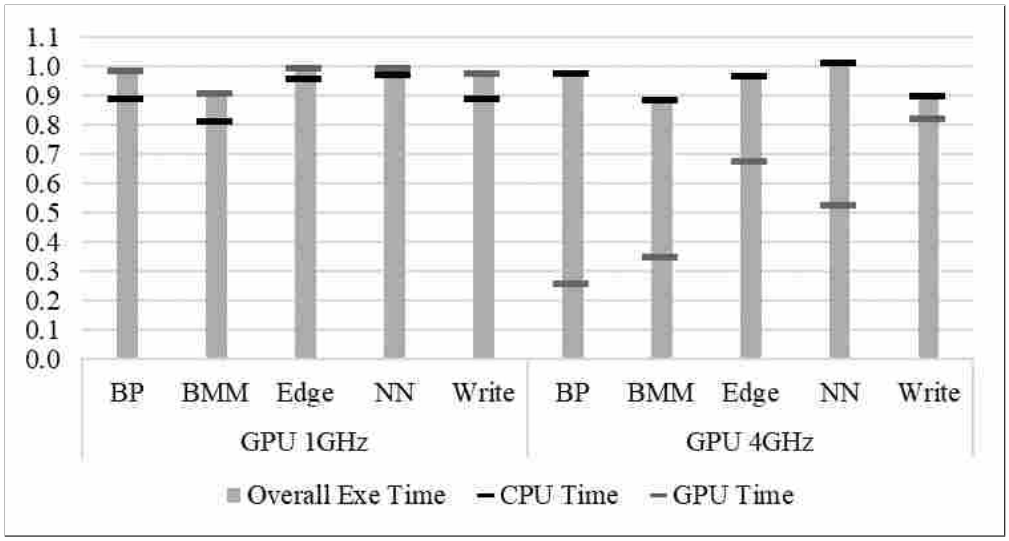


Figure 4.13: Increasing GPU Bandwidth and Frequency

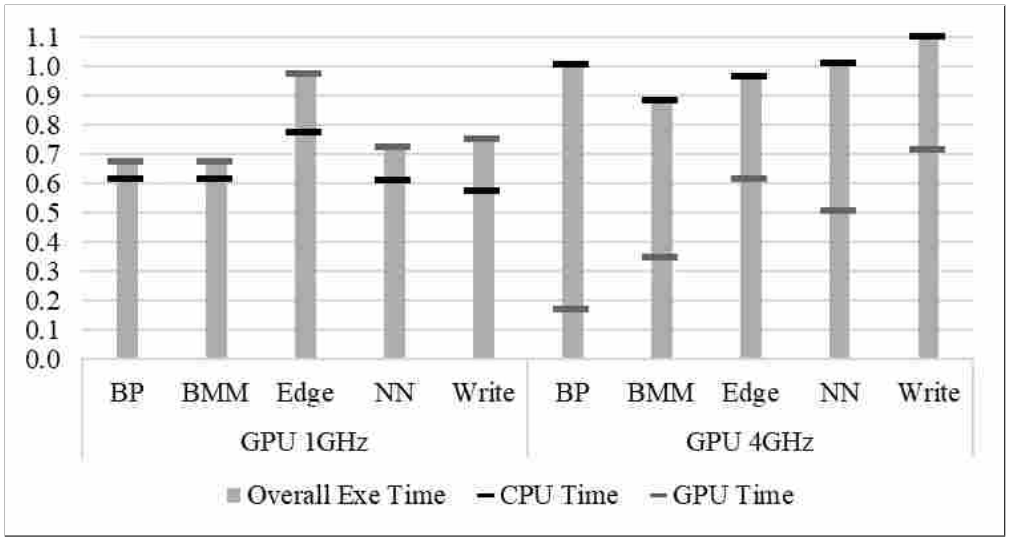


Figure 4.14: Increasing GPU Bandwidth, Frequency, and Coalesce Size

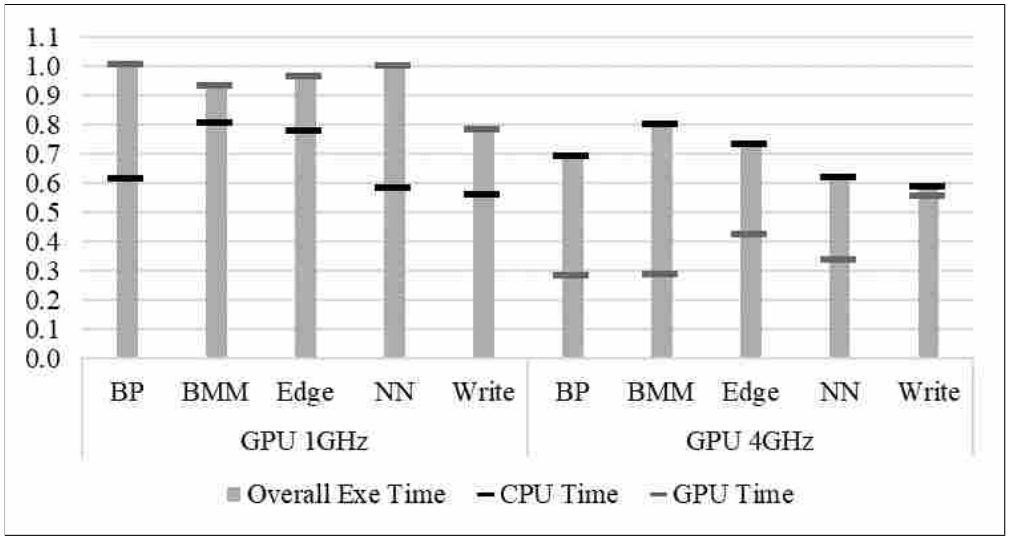


Figure 4.15: Decreasing Switching Fabric Latency

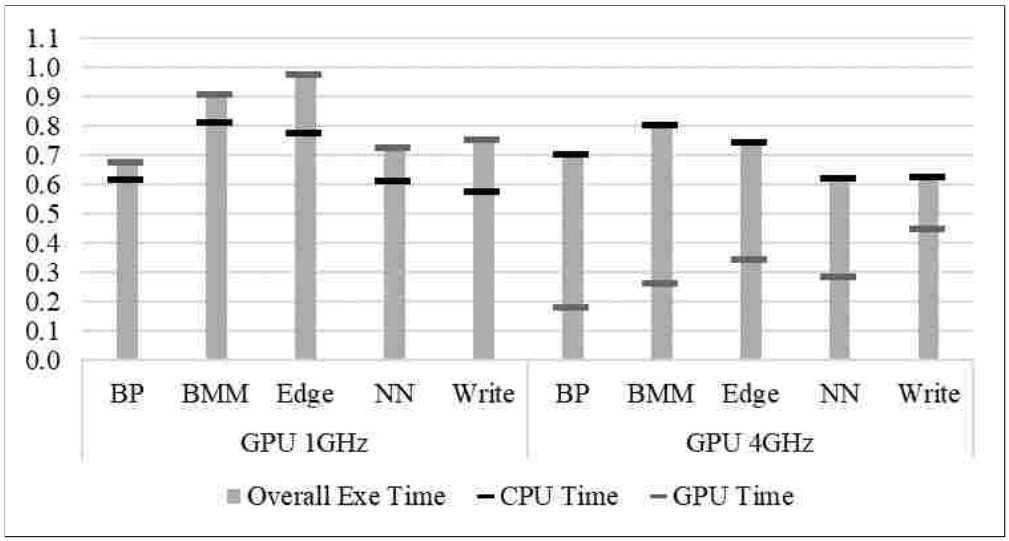


Figure 4.16: Decreasing Switching Fabric Latency and Increasing Coalesce Size

I decided to vary GPU compute unit coalescer size because it has been shown to be a significant determinant in overall GPU performance [66]. I varied GPU compute unit coalescer size by 16, 32, 64, 96, and 128 over a set of test executions and found that the performance benefits of GPU compute unit coalescing peak at approximately 96 for my benchmarks. I modify GPU to memory controller bandwidth, which comprises flit size increases throughout the GPU memory hierarchy all the way down through the memory controller. Increasing the GPU to memory controller bandwidth results in a significant reduction of contention in the I/O controllers of the GPU memory system, GPU/system agent switch node, system agent, and memory controller. I vary GPU frequency to understand how an increase in GPU frequency places pressure on the lower levels of the memory system. As GPU frequency climbs, memory request frequency coming from the GPU also increases which leads to higher contention between the CPU and GPU within the switching fabric and LLCs. Finally, I vary the latency of the switching fabric so that I can observe the role the switching fabric plays in overall system-wide latency and occupancy.

I start the study by varying GPU frequency from 1GHz to 4GHz and varying GPU to memory

controller bandwidth from 32GB/s to 76GB/s and then from 76GB/s to 288GB/s. The results are shown in Fig. 4.13. M2S-CGM's maximum simulated packet size is 72B, therefore GPU maximum internal bandwidth is 72GB/s at 1GHz and is 288GB/s at 4GHz. The results show that with a 1GHz GPU changing GPU to memory controller bandwidth from 32GB/s to 72GB/s has a small effect on overall collaborative CPU-GPU performance. I measure the average increase in CPU and GPU performance as 1.11x and 1.04x respectively with overall performance limited by the GPU. The results suggest that switching fabric occupancy and latency is more critical to performance in collaborative CPU-GPU executions than simply GPU to memory controller bandwidth alone. The leading cause is that in the collaborative CPU-GPU environment both CPU and GPU memory system requests enter the switching fabric, make their way to the LLCs, and then eventually back to the requesting CPU or GPU L2. The contention and latency incurred over these transactions is a significant driving factor determining collaborative CPU-GPU performance. The measured small increase in CPU and GPU performance is mainly due to the lowered contention and latency within the GPU/system agent switch node, system agent, and memory controller.

When changing the GPU from 1GHz to 4GHz (Fig. 4.13 right side) and changing GPU to memory controller bandwidth from 72GB/s to 288GB/s I immediately observe a significant jump in GPU average speedup to 1.92x and a decrease in average CPU speedup to 1.06x with overall performance now limited by the CPU. I found that one reason for the GPU's jump in performance is that the GPU's internal stall time decreases when increasing GPU frequency. However, I also observe that the increase in GPU frequency leads to a significant increase in switching fabric occupancy and contention. For example, in Nearest Neighbor the average CPU switching fabric node occupancy changes from 21% to 51%, nearly a 2.4x increase in occupancy. Additionally the GPU/system agent switch node has an extremely high 76% occupancy, even with the increased bandwidth of 288GB/s. The results suggest that the higher occupancy and contention in the switching fabric leads to longer memory system access latency and that the CPU's performance is very sensitive to

the additional switching fabric contention caused by the GPU. It is important to note that I now also observe significant differences between CPU and GPU execution time which means the execution has become unbalanced and that the settings leading to the optimal number of complementing CPU cores/threads and CPU workload percentage has shifted elsewhere.

Fig. 4.14 shows the effects of increasing GPU compute unit coalescer size from 32 to 96 in conjunction with changing the GPU from 1GHz to 4GHz and changing GPU to memory controller bandwidth to 72GB/s and 288GB/s. At a 1GHz GPU I observe average CPU and GPU speedups of 1.58x and 1.32x respectively with overall performance limited by the GPU. Interestingly, changing GPU coalescer size has a significant impact to CPU performance as well. I found that increasing GPU compute unit coalescer size leads to a significant reduction in the number of GPU memory system accesses which results in an average CPU switch node occupancy drop from 46% to 19% and GPU/system agent switch node occupancy drop from 62% to 19%. These results tell us that the GPU's coalescer must be adequately sized in heterogeneous CPU-GPU processors. At a 4GHz GPU frequency (Fig. 4.14 right side) the CPU remains impacted by the higher switching fabric occupancy and latency and only achieves an average speedup of 1.01x. However, the change in coalescer size results in an average speedup of 2.14x for the GPU with overall performance still limited by the CPU. These results also suggest that contention within the switching fabric is a major determinant of overall collaborative CPU-GPU performance.

The previous results suggested that contention and latency within the switching fabric are a significant factor in collaborative CPU-GPU performance. So, for the next study I directly evaluate the effect of changing switching fabric latency on overall performance. Again, I vary GPU frequency from 1GHz to 4GHz, but return the GPU to memory controller bandwidth to 32GB/s and set switching fabric latency to a negligible level. The results are shown in Fig. 4.15. With the GPU at 1GHz the lowered latency in the switching fabric resulted in an expected jump in CPU performance with a measured average speedup of 1.50x. However, overall performance is limited

by the GPU, with a measured average speedup of 1.07x. The GPU's lack of performance gain reconciles with my previous findings where I observe higher GPU internal stall time at lower GPU frequencies. With a 4GHz GPU (Fig. 4.15 right side) CPU performance remains approximately the same, however I observe a large jump in GPU performance with a measured average speedup of 2.66x and overall performance limited by the CPU. These results suggest that the switching fabric latency is more critical to the GPU than bandwidth internal to the GPU. Additionally increasing GPU frequency lowers the GPU's internal stall time and allows the GPU to make better use of system wide bandwidth.

Fig. 4.16 shows the effects of increasing GPU compute unit coalescer size from 32 to 96 in conjunction with changes to switching fabric latency. With a 1GHz GPU I observe that changing the coalescer size provides the same results for the GPU previously discussed above, with the exception of Block Matrix Multiply which underperformed due to the lowered bandwidth internal to the GPU. I also observe that the CPU exhibits only minor performance differences across all the benchmarks in Figs. 4.15 and 4.16. This means that my switching fabric changes have reduced the contention between the CPU and GPU to a nearly negligible point which is why there is no change to CPU performance when changing GPU coalescer size. With a 4GHz GPU (Fig. 4.16 right side) I observe another big jump in average GPU performance with a measured average speedup of 3.33x due to the combination of the physical reduction in GPU memory system accesses and the lowered switching fabric latency. However, overall performance remains limited by the CPU.

The third set of experiments successfully shows that GPU compute unit coalescing, GPU frequency, and switching fabric occupancy and latency are major determinants in overall collaborative CPU-GPU performance. The results show that collaborative CPU-GPU performance growth is possible in future heterogeneous CPU-GPU processors. In addition, further efforts can be made to re-balance the collaborative CPU-GPU executions shown in this section. The results show that future collaborative CPU-GPU executions could reach a theoretical average speedup of 3.33x

over the average speedups of my benchmark's best case collaborative CPU-GPU executions and therefore a theoretical average speedup of 6.3x over that of my benchmark's non-collaborative GPU-only executions.

4.6 Experimental Observations

Drawing on the experiences gained during the implementation of M2S-CGM, its benchmarks, and the conduct of the experiments presented in this dissertation I have gleamed several important insights related to the nature of heterogeneous CPU-GPU processor designs and applications.

- The CPU and GPU absolutely should be capable of sharing a single virtual address space and other system resources, like, the LLC. The results showed that by sharing a single virtual address space and other system resources underlying mechanisms like memory copies between the CPU and GPU are no longer required and higher levels of parallelism can be obtained through traditional synchronization and coherency mechanisms.

- Sharing the LLC between the CPU and GPU can result in a measurable boost in performance when the CPU and GPU coherently operate, however introducing coherency between the CPU and GPU is more critical.

- GPU address translation mechanisms need to be researched more. The current IOMMU approach incurs significant overhead as the system must trap back to the CPU to solve address translation issues. The GPU should be capable of resolving address translation issues on its own.

- In collaborative CPU-GPU applications using too little or too much CPU workload percentage and/or using too few or too many CPU cores/threads can lead to overall performance degradation.

- In collaborative CPU-GPU applications it is possible to have multiple configurations leading to balanced executions, however only one configuration will result in maximum performance.

- Optimal collaborative CPU-GPU application settings are *non-portable* between collaborative applications and heterogeneous CPU-GPU processor architectures.

- Developers should focus on increasing GPU compute unit coalesce size, GPU frequency, and lowering switching fabric latency in future heterogeneous CPU-GPU processors.

- New research regarding the development of collaborative CPU-GPU system profiling tools needs to be conducted. These new profiling tools can help to better predict an application's optimal configuration settings for a specific heterogeneous CPU-GPU processor architecture and can help developers and hypervisors attain maximum performance.

4.7 Summary and Conclusions

In this chapter I have presented an in-depth simulation backed architectural study of the trade space regarding the impacts of added coherency between the CPU and GPU with and without shared LLC, the optimization of the number of complementing CPU cores/threads and CPU workload percentage in collaborative CPU-GPU applications, and the impacts of key architectural features on collaborative CPU-GPU performance.

In the first study my benchmark executions show that added coherency between the CPU and GPU can provide significant performance gains. Results show that our modeled half coherent CPU-GPU heterogeneous system achieves speedups of 3.27x, 1.06x, 0.94x, 6.51x, 1.21x, and 1.15x and my fully coherent CPU-GPU heterogeneous system achieves speedups of 3.67x, 1.06x, 0.95x, 8.83x, 1.23x, and 1.16x for Backprop, LUD, Kmeans, Hotspot, Needleman, and BFS respectively over the

noncoherent equivalent. In the second study my benchmark execution results show that collaborative CPU-GPU benchmarks can achieve speedups as high as 2.23x over that of non-collaborative GPU-only benchmarks. The results also show that using too little or too much CPU workload percentage and/or using too few or too many CPU cores/threads can lead to performance degradation. Therefore, developers should endeavor to find the optimal configuration points for their collaborative CPU-GPU applications. Choosing the right optimal configuration point can result in significantly higher performance over arbitrarily complementing the GPU with a number of CPU cores/threads and CPU workload percentage. In the third study my benchmark execution results show how varying four key architectural parameters impacts collaborative CPU-GPU performance. I found that GPU compute unit coalesce size, GPU frequency, and switching fabric contention and latency are major determinants of overall collaborative CPU-GPU performance. The results show that future potential architectural changes to heterogeneous CPU-GPU processors can result in theoretical average speedups of 6.3x over non-collaborative GPU-only executions on today's processors. Developers should focus on improving these key architectural elements in future heterogeneous CPU-GPU processor designs.

CHAPTER 5: RELATED WORK

This chapter presents related work to the subject matter of this dissertation. Related work to my own can be broken down into the research areas of sequential and parallel discrete event-driven simulation methodologies, computer architectural simulation systems, heterogeneous CPU-GPU benchmarks, and heterogeneous CPU-GPU architectural studies which I present over the following sections.

5.1 Related Work in Sequential Discrete Event-Driven Simulation Methodologies

There is a long and diverse history of related work concerning discrete event-driven simulation covering a broad spectrum of methodologies and techniques. Information regarding many of these methods and techniques can be found over the course of a few comprehensive and relevant surveys [67, 68, 69]. In general, related work to my own falls into the category of discrete event-driven simulation methodologies intended for use in the development of computer architectural simulations. Thus for brevity and relevance, I limit the presentation of related work to the methodologies used in current mainstream computer architectural simulation systems.

The implementation methodology from which KnightSim, as presented in chapter 2, inherits its base functionality from is called The Threads Package and has been used in at least the FlashLite and M2S-CGM computer architectural simulation systems [13, 3]. However despite its prior usage, implementation details regarding The Threads Package itself have previously been little discussed. My implementation of KnightSim preserves the interfaces of The Threads Package making them functionally equivalent to each other. However, KnightSim incorporates a completely redesigned and optimized implementation that results in (1) fixes to functional issues that otherwise render

the methodology non-functional in modern Linux distributions, (2) a significant performance enhancement, and (3) a novel parallelized implementation that is further still capable of higher levels of performance.

Examples of directly related work regarding other sequential discrete event-driven simulation methodologies used in current mainstream computer architectural simulation systems can be found in GEM5 [21], Multi2Sim [15], Ruby [70], and their derivative computer architectural simulation systems, such as Gem5-GPU [71] and FusionSim [72]. Each of these computer architectural simulation systems employ a discrete event-driven simulation tool that utilizes a similar technique. In each, the discrete event-driven simulation engine works by scheduling and executing a predetermined event and callback function at a specified cycle. In essence, the discrete event-driven simulation engine's scheduler will call the function passed to it when the number of cycles provided by the developer transpires. Ruby employs a slightly different technique. In Ruby messages are enqueued in buffers linking modeled system elements together. The buffers impose variable latency and bandwidth on inserted events. Simulation execution proceeds by invoking a callback function for the next scheduled event in a given event buffer.

In comparison to the modeling methodologies incorporated in the other computer architectural simulation systems presented here, KnightSim utilizes a different approach to event execution by implementing events as independently executable x86 "KnightSim Contexts". As presented in Sec. 2.2.1, KnightSim Contexts encapsulate all of the functionality and interfaces associated with a single simulated system element in an executable package. KnightSim Contexts are treated as simulation objects that are scheduled for execution by an advance and await mechanism. In this approach occupancy and contention are then automatically modeled by KnightSim Contexts. In the other approaches discussed here, researchers must endeavor to carefully model the latency, occupancy, and contention incurred by the modeled resource. Since these simulation features are not an inherent part of the mechanism, such modeling must be implemented manually with a

collection of events, flags, and appropriate execution timings.

5.2 Related Work in Parallel Discrete Event-Driven Simulation Methodologies

Other parallel discrete event-driven simulation techniques have previously been researched. [73] presents a distributed simulation approach where each process in the physical system is simulated by a separate logical process. [74] presents the Wisconsin Wind Tunnel, a technique that runs a parallel shared-memory program on a parallel computer and uses execution driven, distributed, discrete-event simulation to evaluate the performance of cache coherent, shared-memory computers. Distributed discrete event simulation techniques presented in [75] and [76] utilize a concept called *lookahead*, which is a prediction on a processor's future behavior based on an analysis of the processor's simulation state. [77] presents Hornet, a cycle-level multicore simulator that utilizes timing approximations to provide support for a variety of memory hierarchies, interconnect routing and virtual channel buffer allocation algorithms, and accurate power and thermal modeling. [78] presents zSim, a simulator that utilizes an instruction-driven timing model that leverages dynamic binary translation to speed up the sequential simulation by performing the majority of the work in a core's timing model during program instrumentation.

In comparison to each of these, Parallel KnightSim parallelizes event execution by dividing KnightSim Contexts into multiple context batches for execution in parallel over a user specified number of threads each simulated cycle, see Sec. 2.5. From the developer's perspective, the developer groups contexts with specific threads and during execution the workload is automatically parallelized.

5.3 Related Work in Computer Architectural Simulation Systems and Heterogeneous CPU-GPU benchmarks

There is a significant amount of related work concerning computer architectural simulation methodologies and heterogeneous CPU-GPU benchmarks dating all the way back to the beginning of the GPGPU era, circa 2005. However, older work is quickly becoming less relevant due to the recent rapid advance of heterogeneous processor design. So, for brevity and relevance I present related work that focuses on the recent advent of computer architectural simulation systems supporting CPU-GPU execution in cache coherent environments utilizing shared virtual memory features and the heterogeneous CPU-GPU benchmarks that make use of these new heterogeneous CPU-GPU architectural features.

Related work in heterogeneous CPU-GPU benchmark applications can be found in the Rodinia-SVM, Hetero-Mark, and Chai benchmark suites [4, 7, 11, 12, 79]. The Rodinia-SVM benchmark suite ports each of the previously established, and widely used, Rodinia OpenCL 1.2 version non-collaborative benchmarks [16] to new OpenCL 2.0 collaborative implementations. Both the Hetero-Mark and Chai benchmark suites provide a set of collaborative benchmarks focused on expressing various CPU-GPU collaborative patterns. The Rodinia-SVM benchmark suite was profiled on a physical Intel Skylake system and both the Hetero-Mark and Chai benchmark suites were profiled on a physical AMD A10 APU system.

Related work in heterogeneous computer architectural Simulation systems can be found in Multi2-Sim, Gem5-GPU, and an Intel CPU-iGPU simulator [80, 15, 81]. Multi2Sim is an execution-driven computer architectural simulator that was recently extended to support the HSA/HSAIL runtime and is capable of running the Hetero-Mark benchmarks. Gem5-GPU is an execution-driven computer architectural simulation system that was also recently extended to support the OpenCL 2.0 runtime environment [82]. The Intel CPU-iGPU simulator is a trace-based computer architectural

simulation system that is capable of running CPU and GPU traces taken from physical system executions. Each of the simulators presented here provide configurable CPU, GPU, and memory system simulation components to varying levels of fidelity and support shared LLC and shared virtual memory between the CPU and GPU. Multi2Sim supports ISA level simulation for both recent NVIDIA and AMD GPUs, Gem5-GPU predominantly supports ISA level simulation of NVIDIA GPUs, and the Intel CPU-iGPU simulator supports Intel HD GPU configurations.

In comparison to the related work presented here, I have chosen to implement my own computer architectural simulation system called M2S-CGM [3] and five of my own benchmarks comprising non-collaborative GPU-only and collaborative CPU-GPU implementations. M2S-CGM is similar to the other computer architectural simulators presented here, however in comparison it has a more detailed system wide occupancy and contention model and provides modeled system elements that have not been presented in the related work, such as, a GPU hub/IOMMU and system agent. For my set of benchmarks I port the Rodinia OpenCL Backpropagation and Nearest Neighbor benchmarks and create three additional benchmarks of my own. My benchmark set is designed to specifically support the nature of my experiments, utilizes new heterogeneous CPU-GPU architectural features, and expresses different memory system access patterns between the CPU and GPU. M2S-CGM and my benchmarks enable execution-driven simulation-based research within the collaborative CPU-GPU design space where previously not possible.

5.4 Related Work in Heterogeneous CPU-GPU Architectural Studies

Related work in heterogeneous CPU-GPU architectural studies is diverse and covers many aspects over the breadth of both the software stack and hardware layer. Recent software-based approaches include techniques that manage system coherence such as hypervisor layers and modified programming models [1, 83, 84]. Higher level software-based approaches that utilize the current under-

lying hardware and software may provide a boost in speedup and efficacy of programming, however significant system changes proposed require a more robust software and hardware co-design where the programming model and underlying hardware are both changed to evoke a substantive improvement in processing performance.

Examples of recent work related to software-based approaches include the Heterogeneous System Architecture Intermediate Language (HSAIL) [1], OpenACC [83], and OpenCL [25]. HSAIL provides an intermediate layer that abstracts separate ISAs into a single instruction type. A HSAIL virtual machine manages the execution of the application and automatically constructs executables for the target ISAs and executes them on the target hardware in a cohesive environment. OpenCL and OpenACC are similar where highly parallelizable regions of the application are directly placed on the GPU for execution. Both examples build upon HSAIL and provide higher levels of programmability and reduces the need for developers to manually manage system resources. However, despite the higher levels of flexibility, efficiency, and programmability offered by HSAIL, OpenACC, and OpenCL the architecture of the underlying hardware remains critical to overall application performance. Effective exploitation of the system's hardware in a heterogeneous manner requires that the underlying computational architecture better supports a shared processing environment. This touches many system-level design areas including elements such as the memory system, OS, compiler, runtime, drivers, and to an extent the co-processor itself. In my heterogeneous CPU-GPU system simulation experiments I configure the simulated software stack so that both the CPU and GPU operate in a single coherent shared virtual memory environment.

Examples of recent work related to hardware-based approaches includes a timestamp-based protocol called Temporal Coherence [85]. Here the authors propose a low overhead coherence mechanism for memory systems internal to the GPU that utilizes a timestamp to determine if a cache line is dirty between CPU compute units. A directory-based region coherence protocol called Heterogeneous System Coherence [8]. The authors introduce a set of region buffers to the CPU and GPU

L2 caches. In this context the region buffer acts as a course-grain filter for memory system message traffic. Thus, if the CPU or GPU controls a region of memory it then does not need to consult with other processing elements before preforming reads or writes to main memory. A bus based CPU-GPU integration is proposed in [45, 46]. The authors propose an approach that coherently integrates the CPU and GPU by bus and shows that coherence integration at the hardware level can result in significant performance gains over software-based CPU-GPU coherence. Studies of the effects of LLC sharing during collaborative execution have been made in [86, 87, 88]. The authors utilize simulation or physical systems and execute a set of benchmarks with and without a shared LLC. However, these studies do not propose an integration with the directory in each LLC. An analysis regarding the effects of interference between the CPU and GPU at the LLC is reported in [81]. The authors utilize the Intel CPU-iGPU simulator and a single micro benchmark to study CPU-GPU contention within the LLC while varying CPU and GPU workload parameters and sizes. A study of CPU-GPU collaborative patterns is performed in [10]. The authors utilize the Chai Canny Edge Detection and Random Sample Consensus benchmarks and study the effects of varying collaborative data and task partitioning on an AMD A10 APU. An analysis of OpenCL memory management methods was performed in [9]. The authors port a set of the Rodinia OpenCL benchmarks to make use of OpenCL 2.0's shared virtual memory features and compare the performance of the `sharedalloc` memory management feature to the older OpenCL 1.2 `managed` memory management feature on an AMD A10 APU.

In comparison to the work presented here I performed three architectural studies with the goal of studying the collaborative CPU-GPU processor and application trade space. In my work I study (1) the architectural effects of shared LLC and CPU-GPU coherence on the overall performance of non-collaborative GPU-only applications, (2) the optimization of the number of complementing CPU cores/threads and CPU workload percentage in collaborative CPU-GPU applications, and (3) the impact of future looking architectural changes to GPU compute unit coalescer size, GPU to

memory controller bandwidth, GPU frequency, and switching fabric latency on collaborative CPU-GPU performance. My work provides a deep understanding of collaborative CPU-GPU application performance and architectural interaction characteristics that could not be attained without simulation. I believe that the research presented in this dissertation helps to inform researchers on ways to best optimize collaborative CPU-GPU executions and provides new directions for future research regarding heterogeneous CPU-GPU processor design and profiling tools meant to aid in predicting optimal collaborative CPU-GPU configurations. To the best of My knowledge I believe that my work provides the first in-depth simulation backed study of the collaborative CPU-GPU trade space including optimizing operating points and analyzing architectural parameter impacts.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

In this dissertation I explore the heterogeneous CPU-GPU processor and application design space and answer several open research questions. I study and present several architectural design trade-offs in heterogeneous CPU-GPU processors and have discovered ways to go about maximizing heterogeneous CPU-GPU processor and application performance. Over the course of chapters 2 and 3 I implement two novel and related computer architectural simulation tools, called KnightSim and M2S-CGM. KnightSim advances the state-of-the-art in discrete event-driven simulation methodologies and computer architectural modeling by (1) introducing a sequential event-driven simulation capability that is faster than other equivalent sequential event-driven simulation methodologies in use today and (2) introduces a novel parallel event-driven simulation methodology that is capable of scaling with computer architectural simulation system size. These advancements impact the computer architectural simulation research community by providing increases to computer architectural simulation speed and therefore simulated problem sizes. M2S-CGM advances the state-of-the-art in computer architectural simulation systems by introducing the capability to simulate heterogeneous CPU-GPU processors with coherency between the CPU and GPU, shared LLC, and shared virtual memory address spaces. This advancement impacts the computer architectural simulation research community by enabling execution-driven simulation-based research within the collaborative CPU-GPU design space where previously not possible and subsequently enables the architectural studies presented in this dissertation.

In chapter 4 I present the results of three heterogeneous CPU-GPU architectural studies. In the first study my benchmark executions show that added coherency between the CPU and GPU can provide significant performance gains over equivalent non-coherent implementations. In the second study my benchmark execution results show that, indeed, collaborative CPU-GPU benchmarks can achieve speedups over that of non-collaborative GPU-only benchmarks and provides approaches

to achieve maximum performance in collaborative CPU-GPU applications. In the third study my benchmark execution results show how varying four key architectural parameters impacts collaborative CPU-GPU performance. These three studies together help to advance the state-of-the-art in heterogeneous CPU-GPU processor design and application performance. To the best of my knowledge I have presented the first simulation backed architectural study of the heterogeneous CPU-GPU design space and have showed how heterogeneous CPU-GPU application performance changes with simulated system architecture. This advancement impacts the computer architectural simulation research community by helping to inform researchers on ways to best optimize collaborative CPU-GPU executions and by providing new directions for future research regarding heterogeneous CPU-GPU tools and processor design.

Ready-made and fully working implementations of KnightSim, Parallel KnightSim, M2S-CGM, and the benchmarks presented in this dissertation are made available as free software and can be found on GitHub. By making all software freely and readily available researchers can easily download my work for use in supporting future research efforts.

6.1 Future Work

In this section I summarize my thoughts on possible future work which is intended to build upon the work I have presented in this dissertation.

- Utilize Parallel KnightSim and implement parallel CPU, GPU, and memory system computer architectural models. This will result in new performance impacts in computer architectural modeling and could provide measurable performance gains in moderate to large computer architectural models. Future work should focus on determining what the maximum attainable parallel performance is and how to achieve it.

- Explore new uses for KnightSim and Parallel KnightSim. For example, I believe that Parallel KnightSim could be of use in machine learning applications and could be used to implement large neural networks and provide speedups in training. Parallel KnightSim could provide a means to automatically parallelize propagation in neural networks.

- Extend M2S-CGM to include newly emergent heterogeneous CPU-GPU architectural features such as the Heterogeneous System Architecture Intermediate Language (HSAIL), system wide synchronization mechanisms, and to support simulation of the AMD ROCm platform. These extensions will enable further research into the heterogeneous CPU-GPU design space and will keep M2S-CGM's modeling and simulation capabilities current with industry-driven changes in heterogeneous CPU-GPU processors.

- Extend the CGM memory system model and create a standalone portable library with an intuitive API that provides a means to easily integrate CGM with other architectural models. Also provide easier means to configure memory system element models, like, cache hierarchies and switching fabric topologies. With this new tool researchers could easily connect their architectural models to CGM and make use of its extensive memory system simulation capabilities.

- Utilize M2S-CGM and the benchmarks presented in this dissertation to conduct new research regarding the creation of profiling tools intended to help developers and new hypervisors better predict a given heterogeneous CPU-GPU application's optimized parameters and to perform new architectural studies aimed at optimizing switching fabric latency in heterogeneous CPU-GPU processors.

APPENDIX : SETJMP AND LONGJMP ASSEMBLY ROUTINES

The following algorithms are for the 32bit Setjmp and Longjmp assembly routines:

Algorithm Appx.1 Setjmp i386

```
1:  procedure SETJMP(jmp_buf buf)
2:      .section .text
3:      .globl setjmp
4:      .type setjmp, @function
5:      setjmp :
6:      xor   %eax,%eax
7:      mov   0x4(%esp),%edx        ▷ Store callee registers
8:      mov   %ebx,(%edx)
9:      mov   %esi,0x4(%edx)
10:     mov   %edi,0x8(%edx)
11:     lea   0x4(%esp),%ecx        ▷ Get stack pointer
12:     mov   %ecx,0x10(%edx)       ▷ Sore stack pointer
13:     mov   (%esp),%ecx           ▷ Get inst pointer
14:     mov   %ecx,0x14(%edx)       ▷ Store inst pointer
15:     mov   %ebp,0xc(%edx)
16:     mov   %eax,0x18(%edx)
17:     ret
18: end procedure
```

Algorithm Appx.2 Longjmp i386

```
1:  procedure LONGJMP(jmp_buf buf, int val)
2:      .section .text
3:      .globl longjmp
4:      .type longjmp, @function
5:      longjmp :
6:      mov   0x4(%esp),%eax        ▷ Restore callee registers
7:      mov   0x14(%eax),%edx       ▷ Restore inst pointer
8:      mov   0x10(%eax),%ecx       ▷ Restore stack pointer
9:      mov   (%eax),%ebx
10:     mov   0x4(%eax),%esi
11:     mov   0x8(%eax),%edi
12:     mov   0xc(%eax),%ebp
13:     mov   0x8(%esp),%eax        ▷ Set return val
14:     mov   %ecx,%esp
15:     jmp   *%edx
16: end procedure
```

The following algorithms are for the 64bit Setjmp and Longjmp assembly routines:

Algorithm Appx.3 Setjmp x86_64

```
1:  procedure SETJMP(jmp_buf buf)
2:      .section .text
3:      .globl setjmp
4:      .type setjmp, @function
5:      setjmp :
6:      mov   %rbx,(%rdi)              ▷ Store callee registers
7:      mov   %rbp,%rax
8:      mov   %rax,0x8(%rdi)
9:      mov   %r12,0x10(%rdi)
10:     mov   %r13,0x18(%rdi)
11:     mov   %r14,0x20(%rdi)
12:     mov   %r15,0x28(%rdi)
13:     lea   0x8(%rsp),%rdx           ▷ Get stack pointer
14:     mov   %rdx,0x30(%rdi)          ▷ Sore stack pointer
15:     mov   (%rsp),%rax              ▷ Get inst pointer
16:     mov   %rax,0x38(%rdi)          ▷ Store inst pointer
17:     mov   %rax,%rax
18:     ret
19: end procedure
```

Algorithm Appx.4 Longjmp x86_64

```
1:  procedure LONGJMP(jmp_buf buf, int val)
2:      .section .text
3:      .globl longjmp
4:      .type longjmp, @function
5:      longjmp :
6:      mov   0x8(%rdi),%r9            ▷ Restore callee registers
7:      mov   0x10(%rdi),%r12
8:      mov   0x18(%rdi),%r13
9:      mov   0x20(%rdi),%r14
10:     mov   0x28(%rdi),%r15
11:     mov   0x30(%rdi),%r8           ▷ Restore stack pointer
12:     mov   0x38(%rdi),%rdx          ▷ Restore inst pointer
13:     mov   (%rdi),%rbx
14:     mov   %esi,%eax                ▷ Set return val
15:     mov   %r8,%rsp
16:     mov   %r9,%rbp
17:     jmpq  *%rdx
18: end procedure
```

LIST OF REFERENCES

[1] Phil Rogers. Heterogeneous system architecture overview. Symposium on High Performance Chips, 2013.

[2] V. Mekkat, A. Holey, P. Yew, and A. Zhai. Managing shared last-level cache in a heterogeneous multicore processor. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 225–234, Sep. 2013.

[3] C. E. Giles and M. A. Heinrich. M2s-cgm: A detailed architectural simulator for coherent cpu-gpu systems. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 477–484, Nov 2017.

[4] M. Damschen, F. Mueller, and J. Henkel. Co-scheduling on fused cpu-gpu architectures with shared last level caches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2337–2347, Nov 2018.

[5] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 161–171, April 2016.

[6] Shoumeng Yan, Xiaocheng Zhou, Ying Gao, Hu Chen, Gansha Wu, Sai Luo, and Bratin Saha. Optimizing a shared virtual memory system for a heterogeneous cpu-accelerator platform. *SIGOPS Oper. Syst. Rev.*, 45(1):92–100, February 2011.

[7] S. Mukherjee, Y. Sun, P. Blinzer, A. K. Ziabari, and D. Kaeli. A comprehensive performance analysis of hsa and opencl 2.0. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 183–193, April 2016.

[8] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous system coherence for integrated cpu-gpu systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 457–467, New York, NY, USA, 2013. ACM.

[9] Mohammad Dashti and Alexandra Fedorova. Analyzing memory management methods on integrated cpu-gpu systems. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 59–69, 2017.

[10] Li-Wen Chang, Juan Gómez-Luna, Izzat El Hajj, Sitao Huang, Deming Chen, and Wen-mei Hwu. Collaborative computing for heterogeneous integrated systems. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE)*, pages 385–388, 2017.

[11] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli. Hetero-mark, a benchmark suite for cpu-gpu collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, Sep. 2016.

[12] J. Gómez-Luna, I. E. Hajj, L. Chang, V. García-Floreszx, S. G. de Gonzalo, T. B. Jablin, A. J. Peña, and W. Hwu. Chai: Collaborative heterogeneous applications for integrated-architectures. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 43–54, April 2017.

[13] M. Heinrich, D. Ofelt, M. A. Horowitz, and J. Hennessy. Hardware/Software Co-Design Of The Stanford FLASH Multiprocessor. *Proceedings of the IEEE*, 85(3):455–466, Mar 1997.

[14] M. Chaudhuri, M. Heinrich, C. Holt, J. P. Singh, E. Rothberg, and J. Hennessy. Latency, occupancy, and bandwidth in dsm multiprocessors: a performance evaluation. *IEEE Transactions on Computers*, 52(7):862–880, Jul 2003.

[15] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: A simulation framework for cpu-gpu computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 335–344, New York, NY, USA, 2012. ACM.

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct 2009.

[17] C. Giles, C. Peterson, and M. Heinrich. A fast discrete event driven simulation methodology for computer architectural simulation. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; (HPCC)*, pages 510–517, June 2018.

[18] C. Giles, C. Peterson, and M. Heinrich. Knightsim: A fast discrete event-driven simulation methodology for computer architectural simulation. *IEEE Transactions on Computers*, 2019. early access.

[19] C. E. Giles and M. A. Heinrich. A simulation backed architectural study of the collaborative cpu-gpu trade space. Technical report, University of Central Florida, October 2019.

[20] David P. Reed and Rajendra K. Kanodia. Synchronization with eventcounts and sequencers. *Commun. ACM*, 22(2):115–123, February 1979.

[21] 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.

[22] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct 2009.

[23] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob. Dramsim: A memory system simulator. *SIGARCH Comput. Archit. News*, 33(4):100–107, November 2005.

[24] Openmp, the openmp api specification for parallel programming. http://www.openmp.org/. Accessed: Jan-31-2017.

[25] Kronos Group. Opencl: The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/. Accessed: Jan-13-2017.

[26] J. Nickolls and W. J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, March 2010.

[27] Chris McClanahan. History and evolution of gpu architecture a paper survey. Technical report, Georgia Tech, 2011.

[28] D. B. Skillicorn. A taxonomy for computer architectures. *Computer*, 21(11):46–57, Nov 1988.

[29] Intel. Intel streaming simd extensions technology. https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html. Accessed: Oct-13-2018.

[30] Intel. Pci express architecture. https://www.intel.com/content/www/us/en/io/pci-express/pci-express-architecture-general.html. Accessed: Oct-8-2018.

[31] AMD. Amd 7th gen a-series desktop apus. Accessed: Sept-21-2019.

[32] Intel. The compute architecture of intel processor graphics gen9. Technical report, Intel, August 2015.

[33] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, New York, NY, USA, 2010. ACM.

[34] Intel. Intel core i7-4790k processor. Accessed: Oct-8-2018.

[35] Intel. Intel 64 and ia-32 architectures optimization reference manual, 04 2018.

[36] Advanced Micro Devices. Southern islands series instruction set architecture. Technical report, Advanced Micro Devices, Feburary 2014.

[37] J P Turley. Introduction to intel architecture, the basics. Technical report, Intel, Janurary 2014.

[38] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[39] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, June 1990.

[40] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA '88, pages 280–298, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[41] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, June 1990.

[42] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proceedings*

of the 17th Annual International Symposium on Computer Architecture, ISCA '90, pages 148–159, New York, NY, USA, 1990. ACM.

[43] Mark Andrew Heinrich. *The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols*. PhD thesis, Stanford University, 1998.

[44] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *SIGARCH Comput. Archit. News*, 12(3):348–354, January 1984.

[45] T. Suh, H. . S. Lee, and D. M. Blough. Integrating cache coherence protocols for heterogeneous multiprocessor systems. part 1. *IEEE Micro*, 24(4):33–41, July 2004.

[46] T. Suh, H. . S. Lee, and D. M. Blough. Integrating cache coherence protocols for heterogeneous multiprocessor system. part 2. *IEEE Micro*, 24(5):70–78, Sep. 2004.

[47] M. Macedonia. The gpu enters computing's mainstream. *Computer*, 36(10):106–108, Oct 2003.

[48] Mark Harris. Gpgpu: General-purpose computation on gpus. *SIGGRAPH 2005 GPGPU COURSE*, pages 1–51, 2005.

[49] Z. Zheng, Z. Wang, and M. Lipasti. Adaptive cache and concurrency allocation on gpgpus. *IEEE Computer Architecture Letters*, 14(2):90–93, July 2015.

[50] A. Majumdar, G. Wu, K. Dev, J. L. Greathouse, I. Paul, W. Huang, A. Venugopal, L. Piga, C. Freitag, and S. Puthoor. A taxonomy of gpgpu performance scaling. In *2015 IEEE International Symposium on Workload Characterization*, pages 118–119, Oct 2015.

[51] Hyunjun Jang, Jinchun Kim, Paul Gratz, Ki Hwan Yum, and Eun Jung Kim. Bandwidth-efficient on-chip interconnect designs for gpgpus. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 9:1–9:6. ACM, 2015.

[52] Hyesoon Kim, Richard Vuduc, Sara Baghsorkhi, Jee Choi, and Wen-mei Hwu. Performance analysis and tuning for general purpose graphics processing units (gpgpu). *Synthesis Lectures on Computer Architecture*, 7(2):1–96, 2012.

[53] J. Zhu, G. Chen, and B. Wu. Gpgpu memory estimation and optimization targeting opencl architecture. In *2012 IEEE International Conference on Cluster Computing*, pages 449–458, Sep. 2012.

[54] S. Dublish, V. Nagarajan, and N. Topham. Characterizing memory bottlenecks in gpgpu workloads. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–2, Sep. 2016.

[55] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):105–118, Jan 2011.

[56] M. Mantor. Amd radeon^{TM} hd 7970 with graphics core next (gcn) architecture. In *2012 IEEE Hot Chips 24 Symposium (HCS)*, pages 1–35, Aug 2012.

[57] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 48–59, New York, NY, USA, 2010. ACM.

[58] Andy Kegel, PAUL Blinzer, ARKA Basu, and Maggie Chan. Virtualizing io through io memory management unit (iommu). *ASPLOS-XXI Tutorials*, 2016.

[59] Hongil Yoon, Jason Lowe-Power, and Gurindar S. Sohi. Reducing gpu address translation overhead with virtual caching. Technical report, University of Wisconsin–Madison, December 2016.

[60] Jason Power, Mark D. Hill, and David A. Wood. Supporting x86-64 address translation for 100s of gpu lanes. *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 568–578, 2014.

[61] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. Performance gaps between openmp and opencl for multi-core cpus. In *2012 41st International Conference on Parallel Processing Workshops*, pages 116–125, Sep. 2012.

[62] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, IISWC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[63] Jie Shen and Ana Lucia Varbanescu. A detailed performance analysis of the openmp rodinia benchmark. *Proceedings of Technical Report PDS-2011-011, Delft University of Technology, Delft*, 2011.

[64] Dana Schaa and Rafael Ubal. Multi-architecture isa-level simulation of opencl. International Workshop on OpenCL, 2013.

[65] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: A simulation framework for cpu-gpu computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 335–344, New York, NY, USA, 2012. ACM.

[66] F. Candel, S. Petit, J. Sahuquillo, and J. Duato. Accurately modeling the gpu memory subsystem. In *2015 International Conference on High Performance Computing Simulation (HPCS)*, pages 179–186, July 2015.

[67] S. Robinson. Discrete-event simulation: from the pioneers to the present, what next? *Journal of the Operational Research Society*, 56(6):619–629, Jun 2005.

[68] Voon yee Vee and Wen jing Hsu. Parallel discrete event simulation: A survey. Technical report, Nanyang Technological University, 1999.

[69] Fred J. Kaudel. A literature survey on distributed discrete event simulation. *SIGSIM Simul. Dig.*, 18(2):11–21, June 1987.

[70] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005.

[71] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood. Gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, Jan 2015.

[72] V. Zakharenko, T. Aamodt, and A. Moshovos. Characterizing the performance benefits of fused cpu/gpu systems using fusionsim. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 685–688, March 2013.

[73] K. Mani Chandy and Jayadev Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on software engineering*, SE-5(5):440–452, 1979.

[74] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The wisconsin wind tunnel: Virtual prototyping of parallel computers. *SIGMETRICS Perform. Eval. Rev.*, 21(1):48–60, June 1993.

[75] David R Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.

[76] David M. Nicol. Parallel discrete-event simulation of fcfs stochastic queueing networks. *SIGPLAN Not.*, 23(9):124–137, January 1988.

[77] Pengju Ren, Mieszko Lis, Myong Hyon Cho, Keun Sup Shim, Christopher W Fletcher, Omer Khan, Nanning Zheng, and Srinivas Devadas. Hornet: A cycle-level multicore simulator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(6):890–903, 2012.

[78] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 475–486, New York, NY, USA, 2013. ACM.

[79] Y. Sun, S. Mukherjee, T. Baruah, S. Dong, J. Gutierrez, P. Mohan, and D. Kaeli. Evaluating performance tradeoffs on the radeon open compute platform. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 209–218, April 2018.

[80] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood. Gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, Jan 2015.

[81] P. Gera, H. Kim, H. Kim, S. Hong, V. George, and C. C. Luk. Performance characterisation and simulation of intel's integrated gpu architecture. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 139–148, April 2018.

[82] L. Wang, R. Tsai, S. Wang, K. Chen, P. Wang, H. Cheng, Y. Lee, S. Shu, C. Yang, M. Hsu, L. Kan, C. Lee, T. Yu, R. Peng, C. Yang, Y. Hwang, J. Lee, S. Tsao, and M. Ouhyoung. Analyzing opencl 2.0 workloads using a heterogeneous cpu-gpu simulator. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 127–128, April 2017.

[83] OpenACC-standard.org. Openacc. Accessed: May-3-2017.

[84] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 347–358, New York, NY, USA, 2010. ACM.

[85] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. Cache coherence for gpu architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 578–590, Feb 2013.

[86] V. García, J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade, and A. J. Pena. Evaluating the effect of last-level cache sharing on integrated gpu-cpu systems with heterogeneous applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, Sep. 2016.

[87] L. Yu, T. Chen, M. Wu, and L. Liu. Buffer on last level cache for cpu and gpgpu data sharing. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, pages 417–420, Aug 2014.

[88] J. Ma, L. Yu, T. Chen, and M. Wu. Analyzing memory access on cpu-gpgpu shared llc architecture. In *2015 14th International Symposium on Parallel and Distributed Computing*, pages 90–99, June 2015.