Cache Coherence in CPU-GPU Heterogeneous Architectures using GEM5

Stephen Singh*, Andrew Femiano†, Austin Kee‡, Thomas Tymecki§
Department of Electrical and Computer Engineering
University of Florida
singhstephen@ufl.edu*, afemiano@ufl.edu†, austinjkee@ufl.edu‡, thomastymecki@ufl.edu§

***Abstract*—This report investigates cache coherence and memory sharing in CPU-GPU heterogeneous architectures using the GEM5 simulator. Specifically, the study evaluates the performance of Ruby, a modular memory system, in enabling efficient cache coherence between CPU and GPU components. It also explores the benefits and challenges of leveraging Heterogeneous System Architecture (HSA) for shared memory in such systems. Experimental results demonstrate the trade-offs between workload intensity, cache efficiency, and system coherence in different configurations. This document provides a comprehensive analysis of the methodologies, results, and conclusions derived from this project.**

I. Introduction

Heterogeneous architectures, which integrate CPUs and GPUs, have become a cornerstone of modern computing. These systems combine the high single-thread performance of CPUs with the massive parallelism of GPUs. Efficient memory sharing and cache coherence between these components are critical to unlocking their full potential.

Heterogeneous System Architecture (HSA) is an open standard aimed at simplifying memory sharing in such systems by providing a unified address space, enabling both the CPU and GPU to access data without explicit copying. In this project, GEM5 was used to simulate an APU architecture incorporating both CPUs and GPUs. Ruby, the modular memory model in GEM5, was employed to evaluate cache coherence protocols and their impact on system performance.

II. Methodology

A. Methods Attempted

The first method we attempted was **gem5-GPU**, a framework that integrates GEM5 with GPGPU-Sim to provide detailed simulations of heterogeneous CPU-GPU systems. This integration supports simulations of CUDA and OpenCL workloads, making it an excellent candidate for studying heterogeneous memory sharing and cache coherence. gem5-GPU's ability to model both CPU and GPU caches, combined with its flexible configuration options for coherence protocols, offers a robust platform for analyzing data consistency and memory latency in shared-memory architectures. Moreover, gem5-GPU allows researchers to study the interactions between the CPU and GPU in real-world workloads, which aligns well with our project's goals of understanding cache coherence mechanisms under concurrent memory accesses.

The **Full-System AMD GPU model** was the second method evaluated. This model, integrated natively within GEM5, represents AMD's Graphics Core Next (GCN3) architecture and supports the Heterogeneous System Architecture (HSA) for unified memory access. Its built-in support for the HSA packet processor enables efficient CPU-GPU communication, a critical feature for heterogeneous memory sharing. The GCN3 model also includes a detailed memory hierarchy, with support for private, shared, and global memory spaces, making it particularly suited for cache coherence studies. Additionally, its compatibility with modern workloads using the ROCm platform and its ability to simulate end-to-end execution in full-system mode provided a unique opportunity to analyze coherence protocols in realistic scenarios.

Finally, the **GPGPU-Sim standalone integration** was explored. Known for its high-fidelity simulation of GPU architectures, GPGPU-Sim offered an opportunity to examine the GPU cache hierarchy and memory access patterns in detail. Its detailed ISA-level simulation allows for a microarchitectural analysis of compute units and cache operations. While GPGPU-Sim lacks native CPU integration, its ability to simulate CUDA workloads at a granular level makes it a powerful tool for understanding GPU-specific cache behaviors, which could complement a broader study of CPU-GPU coherence.

B. Challenges and Failures

Despite the merits of each method, all three approaches encountered significant challenges that led to their failure in this project. **gem5-GPU** required older versions of dependencies, including GCC and Ubuntu, which were no longer readily available, making it infeasible to build and configure the simulator. The **Full-System AMD GPU model**, while promising, faced issues with outdated build scripts and compatibility with modern toolchains. Additionally, the limited documentation and ongoing development status meant that several features critical for our study were either incomplete or experimental. **GPGPU-Sim**, although excellent for GPU-specific studies, lacked support for heterogeneous systems and required legacy software environments, such as CUDA Toolkit 8.0 and Ubuntu 16.04, which were not practical to maintain or integrate with modern GEM5 builds. Collectively, these limitations prevented us from achieving a functional simulation environment capable of exploring cache coherence and memory sharing in heterogeneous architectures.

C. GCN3 GPU with RUBY

The **GCN3 (Graphics Core Next 3)** GPU model in GEM5 represents a significant advancement in simulating heterogeneous computing systems. It is tailored to emulate AMD's GCN3 architecture, emphasizing compute-intensive workloads rather than graphics-specific tasks. This architecture's hallmark is its support for *Heterogeneous System Architecture (HSA)*, which facilitates unified memory access between the CPU and GPU. This capability is crucial for studying **heterogeneous memory sharing** and **cache coherence**, as it allows both processing units to access a common memory space seamlessly, reducing data transfer overhead.

This RUBY model interacting with GCN3 through software can be seen in the 1 below.

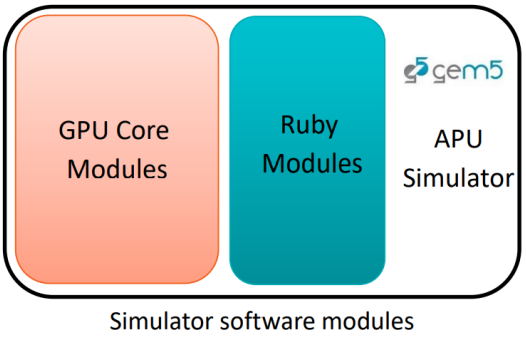

Fig. 1. RUBY GCN Software Architecture

To set up the GCN3 model, GEM5 integrates several key components that emulate real-world GPU behavior. The **HSA Packet Processor** is one of the central elements, enabling efficient communication between the CPU and GPU by handling command queues and dispatching workloads. Additionally, the model incorporates a detailed **cache hierarchy** for the GPU, including private L1 caches and shared memory, as well as the integration of the CPU's L1/L2 caches with the GPU through the memory controller. The use of **Ruby** as the memory system further enhances the ability to simulate advanced cache coherence protocols, such as `GPU_VIPER`, ensuring consistency between CPU and GPU memory accesses.

The `apu_se.py` script serves as the primary configuration file for the GCN3 model. It was customized in this project to enable **multiple compute units (CUs)**, reflecting the parallel processing capabilities of modern GPUs. Furthermore, the script was modified to configure Ruby as the memory controller and to establish a coherent cache hierarchy. These modifications allowed for realistic simulations of heterogeneous workloads where the CPU and GPU collaboratively process data with minimized latency and maximal throughput.

Finally, setting up GCN3 in GEM5 requires careful attention to dependencies and toolchains. The build process involves ensuring compatibility with modern ROCm libraries and HSA runtime environments. However, the evolving nature of the GCN3 model within GEM5 means that certain features may require additional configuration or debugging during setup. These steps, while complex, are critical to leveraging the full capabilities of the GCN3 GPU model for research in cache coherence and memory optimization.

D. Simulator Setup

The simulation environment was set up using GEM5, specifically the Dockerized GCN3 configuration from the official tutorial. The following steps detail the methodology:

The GEM5 APU model integrates the CPU and GPU with a shared memory system. Ruby was configured as the memory controller to manage the cache coherence between the CPU and GPU. The `apu_se.py` script was modified to enable:

- Multiple compute units (CUs) for GPU simulations.
- A coherent cache hierarchy using Ruby.
- Integration of the HSA packet processor for efficient CPU-GPU communication.

This GPU architecture for GCN3 can be seen in the 2 below.

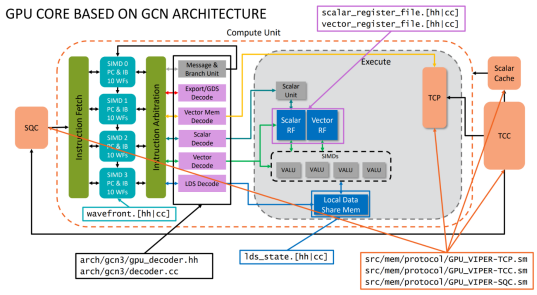

Fig. 2. GPU GCN Architechture

E. Ruby and Cache Coherence

Ruby is a highly customizable memory system in GEM5 that supports various coherence protocols. For this project, the **GPU_VIPER** protocol was used to enable coherence between the CPU's $L1/L2$ caches and the GPU's cache system. Ruby operates by maintaining a directory of cache states, ensuring that data consistency is preserved even when both the CPU and GPU access shared data. This can be seen in Figure 3 below.

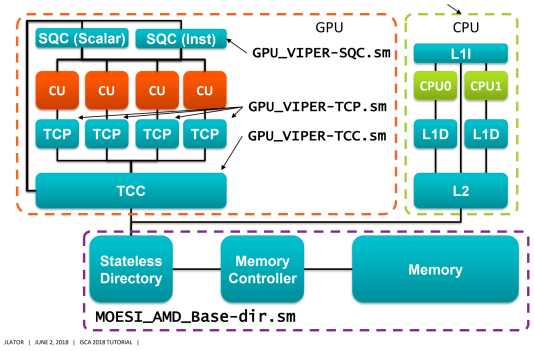

Fig. 3. GPU VIPER protocol

F. Heterogeneous System Architecture (HSA)

HSA unifies CPU and GPU memory access through a shared virtual memory space. In this setup:

- The CPU and GPU access the same physical memory addresses.
- The HSA runtime dispatches GPU tasks via the HSA packet processor.
- Data consistency is managed through Ruby's coherence protocol.

The HSA architecture that integrates with the GPU architecture for GCN3 can be seen in Figure 4 below.

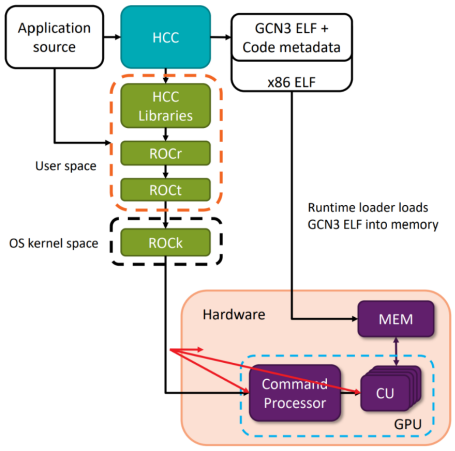

Fig. 4. GPU VIPER protocol

G. Workload and Metrics

The 'square' workload was used to simulate GPU compute tasks, with the CPU initializing data and the GPU performing computations. We altered the square file to run at a slightly less intensive workload to see how the GPU would handle it. Key metrics included:

- Cache hits, misses, and access rates for L1 and L2 caches.
- Memory read/write operations at the memory controller level.
- Task dispatch efficiency from the HSA packet processor.

III. Results and Analysis

The results compare the cache performance in high-intensity and low-intensity workloads. Key findings are summarized below.

A. L1 and L2 Cache Performance

The results reveal notable differences in cache performance between high-intensity and low-intensity workloads, underscoring the critical role of cache hierarchies in optimizing memory access and overall system performance.

Cache	Metric	High-Intensity	Low-Intensity
3*L1DCache	Data Array Reads	7.37M	16,571
	Data Array Writes	3.22M	9,150
	Hit Rate	95.8%	97.4%
3*L1ICache	Data Array Reads	50.99M	93,023
	Data Array Writes	483K	40
	Hit Rate	99.9%	99.3%
3*L2Cache	Data Array Reads	38.37M	9,191
	Data Array Writes	10.08M	1,234
	Hit Rate	86.2%	86.6%

TABLE I
L1 and L2 Cache Performance for High-Intensity and Low-Intensity Workloads.

a) L1 Cache Performance: For the **L1 Data Cache (L1DCache)**, the high-intensity workload exhibits significantly higher access rates, with approximately 7.37 million data array reads and 3.22 million writes. The **95.8% hit rate** suggests that the L1 data cache effectively handles the increased workload, minimizing the number of accesses that escalate to higher levels in the memory hierarchy. By contrast, the low-intensity workload shows drastically lower access rates, with only 16,571 reads and 9,150 writes. The hit rate for the low-intensity workload, at **97.4%**, is marginally higher than that of the high-intensity workload, indicating that the cache is better utilized under lighter loads, possibly because the working set fits entirely within the L1DCache.

The **L1 Instruction Cache (L1ICache)** demonstrates exceptionally high hit rates in both scenarios, with **99.9%** for high-intensity workloads and **99.3%** for low-intensity workloads. This near-perfect performance can be attributed to the predictable nature of instruction access patterns, which benefit from the prefetching mechanisms and spatial locality within the instruction stream. However, the access rates differ significantly between workloads, with 50.99 million reads in the high-intensity scenario compared to just 93,023 in the low-intensity case. This highlights the correlation between workload complexity and instruction fetch intensity.

b) L2 Cache Performance: The **L2 Cache** results reinforce the differences in workload intensity. For high-intensity workloads, the cache processes approximately 38.37 million reads and 10.08 million writes, achieving a hit rate of **86.2%**. While this hit rate is lower than that of the L1 cache, it remains robust given the increased volume of traffic from both the L1DCache and L1ICache. This result suggests that the L2 cache is effectively buffering access to the main memory, reducing memory latency for these workloads. For low-intensity workloads, the L2 cache handles only 9,191 reads and 1,234 writes, with a slightly higher hit rate of **86.6%**, likely due to the reduced contention for cache resources.

IV. Discussion

A. Ruby and Coherence Performance

Ruby effectively maintained coherence between the CPU and GPU caches. The directory-based protocol minimized data inconsistencies but introduced some latency under high workloads due to frequent invalidations and tag array accesses.

B. HSA and Shared Memory

HSA's unified memory model simplified memory management, allowing both the CPU and GPU to work seamlessly on shared data. However, the reliance on Ruby's coherence protocol highlighted the trade-offs between simplicity and performance under high-intensity tasks.

C. Insights and Recommendations

- **Low Workload:** The system achieves near-optimal hit rates, making it suitable for lightweight applications.
- **High Workload:** Performance bottlenecks emerge due to increased coherence overhead. Optimizations, such as prefetching or improved protocols, may help.

V. Limitations/Future Work

A. CPU timing models

According to the gem5 documentation, "The supported `cpu_types` (X86KvmCPU and AtomicSimpleCPU) as timing CPUs do not support the disjointed Ruby network required to simulate a discrete GPU". The implication here is that simulations currently cannot account for race conditions and competitions for cache access between a threaded CPU application and a discrete GPU, as the CPU is simulated without timing information. The lack of detailed CPU timing models in GPU simulations suggests that the results can only be interpreted about ideal access conditions.

B. Extension of HeteroGarnet (Garnet 3.0) with CPU-focused

With the current limitations of gem5 established, we propose additional development on gem5 to provide some support for CPU timing together with GPU modeling. As Unified Memory and zero-copy architectures become more common in the wild for heterogeneous computing (i.e.: ARM Mali, Apple/Imagination PowerVR, Apple M-series), there is greater interest in simulating such architectures. Since it is claimed that the Ruby network simulation for full GPU architectures like Vega (GCN 5.x) is too complex to include CPU behavior in the model, extending HeteroGarnet with a more generic GPU memory consumer should be explored.

C. Extension of GPU simulation parameters

Some parameters that modify GPU configuration only modify the value in gem5 and not the simulated device. For example: "The `dgpu_mem_size` parameter does not change the amount of memory seen by the device driver and is hardcoded to 16GB in C++". Since some parameters cannot be changed on simulated devices, optimizing the system's architecture on simulated devices are limited. If modifying parameters like `dgpu_mem_size` is supported, the effects heterogeneous models can be studied at various memory sizes.

D. Multi-GPU Simulation

The use of multiple GPUs in heterogeneous computing systems introduces additional opportunities and challenges for memory sharing and cache coherence. Multi-GPU setups are increasingly common in high-performance computing (HPC) and machine learning workloads, where they provide the computational throughput needed to handle large-scale data parallelism. By distributing tasks across multiple GPUs, systems can achieve significant performance improvements, particularly in scenarios involving matrix multiplications, deep learning, or simulations requiring high levels of concurrency.

From a memory sharing perspective, multi-GPU architectures require efficient interconnects, such as *NVLink* or *Infinity Fabric*, to facilitate fast and coherent communication between GPUs. These interconnects enable the sharing of memory pages and reduce the overhead of transferring data between GPUs. The inclusion of such features in GEM5's simulation environment could provide valuable insights into optimizing heterogeneous memory systems.

VI. Conclusion

This study demonstrates the critical role of cache coherence in ensuring efficient performance in CPU-GPU heterogeneous systems. The modular design of Ruby offers significant flexibility in configuring cache coherence protocols and memory hierarchies, making it a powerful tool for exploring diverse system architectures. However, the results indicate that this flexibility comes at the cost of increased overhead, particularly under high-intensity workloads where the demands on memory access and consistency mechanisms are significantly amplified.

The adoption of Heterogeneous System Architecture (HSA) simplifies programming and facilitates unified memory sharing between CPUs and GPUs, allowing developers to optimize applications for heterogeneous platforms. Nevertheless, the effectiveness of HSA depends heavily on robust coherence mechanisms to manage shared memory efficiently and minimize performance bottlenecks.

Future work should focus on evaluating alternative coherence protocols to identify designs that reduce overhead while maintaining data consistency. Additionally, the exploration of multi-GPU configurations is a promising direction for extending this study. Multi-GPU setups introduce new challenges, such as inter-GPU communication, memory sharing, and scalability, which require innovative approaches to cache coherence and memory hierarchy design. These investigations will be critical for optimizing next-generation heterogeneous computing systems and addressing the growing demand for performance and efficiency in parallel workloads.

Acknowledgments

The project was part of the Advanced Computer Architecture course. Thanks to the GEM5 community for providing extensive resources.

References

[1] N. Binkert et al., "The GEM5 Simulator," ACM SIGARCH Computer Architecture News, 2011.
[2] AMD, "Heterogeneous System Architecture," 2018.
[3] AMD Research, "AMD gem5 APU Simulator," presented at ISCA 2018. [Online]. Available: https://old.gem5.org/wiki/images/1/19/AMD_gem5_APU_simulator_isca_2018_gem5_wiki.pdf