Enabling Collaborative Heterogeneous Computing

A Dissertation Presented

by

Yifan Sun

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in

Computer Engineering

Northeastern University

Boston, Massachusetts

August 2020

Contents

List of Figures

List of Tables

Acknowledgments

First, I would like to thank my Ph.D. advisor Dr. David Kaeli for his selfless support. He guided me to the path of researching on CPU-GPU Collaborative Computing. He supported my attempt to start a new GPU simulator infrastructure, which eventually becomes MGPUSim. Because of this work, I can establish my Multi-GPU collaborative execution research. He helped me built connections to other researchers and gave me important support on every aspect of my research and career.

I would also like to thank my Ph.D. committee, Dr. Gunar Schirner and Dr. Rafael Ubal, for the constructive feedback for my proposal and dissertation.

I cannot complete this dissertation without the help and feedback from my colleagues at Northeastern University Computer Architecture Research (NUCAR) Group, including, Nicolas Bohm Agostini, Yuhui Bao, Trinayan Baruah, Zhongliang Chen, Shi Dong, Xun Gong, Julian Gutierrez, Charu Kalra, Elmira Karimi, Griffin Knipe, Xiangyu Li, Amir Momeni, Saoni Mukherjee, Fanny Nina-Paravecino, Fritz Gerald Previlon, Derek Rodriguez, Kaustubh Shivdikar, Yash Ukidave, Leiming Yu, and Amir Kavyan Ziabari. I also thank the undergraduate and high school students who have been working with me over the past few years. It is my privilege to work with them.

I have also been collaborating with researchers outside the NUCAR group. I appreciate the opportunity to work with faculties, post-docs, and Ph.D. students from the Department of Mechanical and Industrial Engineering, the Khoury College of Computer Sciences, and the College of Arts, Media, and Design, including Rana Azghandi, Rozhin Doroudi, Ozlem Ergun, Min Gong, Jacqueline Griffin, Casper Harteveld, Omid Mohaddesi, and Pedro Sequeira. The collaboration with a team outside Northeastern University is also extremely helpful. This team includes José L. Abellán, Leila Delshadtehrani, Ajay Joshi, John Kim, Yenai Ma, and Saiful A Mojumder. This team has contributed to most of my work on multi-GPU collaborating execution.

I have had two great internship experiences during my Ph.D. years, at Dell EMC and AMD. The work I have done in these companies has significantly broaden my horizon and has thrust my academic research. I want to thank my advisors and colleagues at Dell EMC, including Jack Harwood, John Cardente, Rob Lincourt, Junping Zhao, Layne Peng, Jie Bao, and Kun Wang. I also want to thank Gray Christeen, who is my advisor at AMD.

Finally, I would like to express my deepest gratitude to my family. I cannot finish my dissertation without the support from my parents. My wife, Yixuan Zhang, has been supporting me over the years in all the aspects of my life and career. I also want to express my gratitude to our daughter Elianna Sun, who has brought us a lot of joy in these years.

Abstract of the Dissertation

Enabling Collaborative Heterogeneous Computing

by

Yifan Sun

Doctor of Philosophy in Computer Engineering

Northeastern University, August 2020

Dr. David Kaeli, Advisor

GPUs have been accelerating a wide range of algorithms and applications with their massively parallel computing capabilities. Today, as GPU programs work under the close supervision of a CPU, computing platforms are usually equipped with both CPUs and GPUs. Most existing GPU programs underutilize the computing capabilities of the CPU since the CPU is frequently idle while GPUs are executing. Additionally, existing work primarily focuses on using a single GPU, thus failing to utilize the computing power of multi-GPU systems.

In this dissertation, we propose a new heterogeneous computing paradigm – Collaborative Heterogeneous Computing. Collaborative Heterogeneous Computing leverages fine-grained CPU-GPU communication mechanisms in computation with the use of CPUs and multiple GPUs simultaneously. Collaborative Heterogeneous Computing can be divided into CPU-GPU collaborative computing and Multi-GPU collaborative computing.

For CPU-GPU collaborative computing, we first categorize 7 CPU-GPU collaborative execution patterns. With the guidance of the summarized CPU-GPU execution patterns, we implement Hetero-Mark, a benchmark suite that enables users to explore CPU-GPU collaborative execution patterns. We also design and develop Multi2Sim-HSA, an HSA system emulator that emulates behaviors of CPUs and GPUs in CPU-GPU collaborative execution applications. In addition, we observe that CPU-GPU communication over the PCIe network causes congestion and can become a major performance bottleneck. To address this issue, we propose a priority-based PCIe scheduling algorithm that can improve device utilization and satisfy Quality of Service (QoS) requirements.

For multi-GPU collaborative computing, we develop MGPUMark, a benchmark suite that explores multi-GPU collaborative execution and fine-grained inter-GPU communication. Benchmarks in MGPUMark enable exploration of inter-GPU communication patterns. We also design and develop MGPUSim, a high-performance, high-flexibility, and high-accuracy GPU simulator. MGPUSim can

faithfully model behaviors of GPUs in multi-GPU collaborative computing. We identify that the main performance bottlenecks occur due to long-latency, low-bandwidth, inter-GPU interconnects. Thus, reducing inter-GPU communication traffic can improve multi-GPU system performance. We develop the Locality APIs, a set of GPU programming APIs that enables programmers to place the computing threads and associated data on a multi-GPU system to reduce inter-GPU traffic. We also introduce Progressive Page-Splitting Migration (PASI), a hardware-based solution that allows GPUs to adjust data placement to avoid inter-GPU traffic.

Chapter 1

Introduction

Computing has become an essential resource that drives the development of science and technology. High-performance computing and low-power edge computing have enabled critical technologies such as remote sensing [1], medical imaging [2, 3, 4], gene sequencing [5], computational materials science [6, 7], and artificial intelligence [8, 9]. In the past decade, CPU-GPU heterogeneous computing has transformed the landscape of computing as GPUs can speed up data-parallel workloads by 10X - 1000X over CPU implementations [10]. Today, more than 26% of the top 500 most powerful supercomputers use GPUs [11]. GPUs have also contributed to more than half of the total computing power of the newly-built (November 2017 - July 2018) Top-500 supercomputers.

The traditional heterogeneous computing paradigm struggles to fully utilize the computing power of computing platforms that equip both CPUs and GPUs [12, 13, 14]. Most existing applications underutilize the CPU's computing capability since the CPU is frequently idle during GPU execution [15]. Existing applications also tend to use only a single GPU [16, 17] and cannot exploit the computing power of multi-GPU systems.

To enable applications to utilize modern multi-CPU, multi-GPU systems, computing devices (i.e., CPUs and GPUs) need to work closely together. To this end, we propose a new heterogeneous computing paradigm, *Collaborative Heterogeneous Computing*. Collaborative Heterogeneous Computing leverages fine-grained CPU-GPU communication mechanisms to calculate with both CPUs and GPUs simultaneously. In this dissertation, we analyze existing heterogeneous-computing programs to identify potential collaborative computing patterns. Then, we develop benchmark suites, as well as performance modeling tools, to understand the behavior of collaborative computing workloads. Finally, we propose new software and hardware solutions to improve both programmability and the performance of Collaborative Heterogeneous Computing.

We divide Collaborative Heterogeneous Computing into two parts: 1) CPU-GPU Collaborative Computing, and 2) Multi-GPU Collaborative Computing. We will discuss the contributions of this thesis in terms of each of these parts.

1.1 CPU-GPU Collaborative Computing

The traditional CPU-GPU heterogeneous computing paradigm underutilizes the computing capacity of CPUs. Many CPU-GPU computing applications assign a majority of the computing workload to the GPU and keep the CPU idle throughout the program execution. A few other applications leverage both the CPU and the GPU, but the CPU and the GPU rarely calculate simultaneously. An application typically needs to copy data to the GPU's memory before the GPU starts processing the data. After the GPU completes data processing, the application copies the results back to the CPU. The CPU waits for the results during this process, wasting its computing power. Overlapping CPU execution with GPU execution is necessary to improve resource utilization.

Moreover, CPUs and GPUs typically need to communicate through the PCIe network. Although the bandwidth of the PCIe network is increasing in each PCIe version (32GB/s for a X16 PCIe 3.0 link and 64GB/s for a X16 PCIe v4.0 link), it is still dwarfed by the local GPU memory bandwidth (e.g., 1TB/s, as in AMD's Radeon VII GPU [18], and 1.5TB/s, as in NVIDIA's A100 GPU [19]). The CPU-GPU communication channel has been observed to be a major performance bottleneck in modern big-data applications [20]. Therefore, properly scheduling the communication over the PCIe link is crucial to improve the device utilization of future CPUs and GPUs in high-performance computing systems.

In recent years, GPU vendors have added new features such as the Unified Memory [21] and System-Level Atomics to enable fine-grained CPU-GPU communication. However, it is still challenging to implement CPU-GPU collaborative computing applications. The differences in the memory model of CPUs and GPUs complicate CPU-GPU synchronization and hinder writing correct and deadlock-free programs. A common dilemma for researchers is to choose the best synchronization primitives. CPU and GPU synchronization mainly rely on locks and barriers, respectively. They require a developer to be highly-experienced in leveraging CPU and GPU synchronization primitives to write fully-functional and high-performance CPU-GPU collaborative computing applications. Due to these difficulties in CPU-GPU synchronization, we see a need to generate "reusable" solutions to reduce development effort. To provide these reusable solutions, we

categorize 11 commonly-used patterns in existing CPU-GPU workloads. We also provide reusable templates to allow developers to create and adapt to their applications with less effort.

The challenges of implementing CPU-GPU collaborative computing applications are rooted in their profound performance implications. Developers commonly encounter pitfalls that may make CPU-GPU collaborative computing slower than only using GPUs alone. Previously, developers could estimate the execution time of a heterogeneous-computing application by summing up the CPU execution time, the GPU execution time, and the memory copy time. New communication features have introduced complex interplay between devices and have complicated performance estimation. Before we can enjoy performance improvements offer by CPU-GPU collaborative computing, we need to understand the performance better. Benchmark suites and emulators are the essential tools to facilitate performance evaluation.

Despite a large number of GPU benchmark suites, most of them focus on the computing throughput of GPUs, rather than evaluating the overall CPU-GPU system performance. There are a general lack of workloads for assessment of new CPU-GPU computing features, hindering the development of new CPU-GPU computing systems. We therefore develop Hetero-Mark, a benchmark suite that has a wide range of workloads to explore the CPU-GPU collaborative computing patterns. We also use Hetero-Mark to evaluate the Radeon Open Compute (ROCm) platform [22], a state-of-the-art, high-performance, heterogeneous-computing platform developed by AMD.

A system emulator can reveal the behavior of a system and evaluate the performance of a proposed system design without manufacturing an actual device. Existing emulators can emulate CPU-GPU systems, but do not support new heterogeneous computing frameworks such as the Heterogeneous Systems Architeture (HSA) (a new heterogeneous computing framework that emphasizes CPU-GPU collaboration). In this dissertation, we introduce Multi2Sim-HSA, an HSA system emulator that emulates both the HSA runtime library and HSA intermediate language (HSAIL) instructions. We build Multi2Sim-HSA based on Multi2Sim [23, 24]. Multi2Sim is a natural selection for emulating CPU-GPU collaborative computing since it can emulate both CPU and GPU architectures.

To improve the utilization of computing resources in CPU-GPU systems, we need a systematic solution that can schedule CPU-GPU communication. To this end, we design and evaluate a priority-based PCIe scheduling mechanism. By carefully scheduling CPU-GPU communication packets, we can enable the GPU to overlap computing with the communication. We also explore a novel scheduling mechanism that can meet the Quality of Service (QoS) requirements of demanding applications. This design is tailored for cloud-based CPU-GPU heterogeneous computing, where

multiple tenants may share physical hardware resources, resulting in congestion in the PCIe network.

1.2 Multi-GPU Collaborative Execution

Modern GPUs can provide much higher computing capabilities than CPUs. However, due to the scaling challenges of CMOS technology, it has become increasingly impractical to add more compute resources to a single GPU [25]. As a result, these single GPU systems cannot support the processing needs of future big-data applications [26, 27, 28].

One attractive path to sustain GPU performance scaling growth is to integrate multiple GPUs into a single platform. NVIDIA offers multi-GPU DGX platforms [12, 13], focusing on accelerating Deep Neural Network (DNN) training. AMD integrates four MI25 GPUs in its TS4 servers [29] to accelerate deep learning applications. Prior work has also proposed designing Multi-Chip-Module (MCM) GPUs to integrate multiple GPU chips in a single package [25]. However, recent studies suggest that GPU-GPU synchronization and multi-GPU memory management overhead constrain the performance of multi-GPU systems [17, 20, 30]. Designing a high-performance low-overhead inter-GPU communication fabric and memory management system is necessary to unlock the full potential of future multi-GPU platforms.

Designing solutions for multi-GPU systems requires a better understanding of multi-GPU system performance. In the traditional multi-GPU programming model, a GPU works independently and does not communicate with other GPUs during kernel execution. Thus, it is easy to predict the execution time of an application. If the data to be processed is large and can be evenly distributed into N GPUs, the execution time is $1/N$ of the single-GPU execution time. With multi-GPU collaborative execution, GPUs communicate, synchronize, and compete for shared resources such as the inter-device network. The profound performance implications of multi-GPU collaborative execution demand more thorough investigation. Therefore, similar to the CPU-GPU collaborative computing study, a benchmark suite and a simulator are needed.

Prior GPU benchmark suites mainly evaluate the performance of a single GPU. In this dissertation, we present MGPUMark, a multi-GPU computing benchmark suite. We first identify multi-GPU communication patterns. Then, we provide a wide range of workloads that aim to help explore the multi-GPU collaborative execution patterns. The workloads in the MGPUMark suite overlap with Hetero-Mark, but MGPUMark adapts these workloads for multi-GPU execution. We also select other workloads from popular benchmark suites such as the AMDAPPSDK [31] and the Rodinia [32] suites, introducing new extensions for multi-GPU computing.

In additional to a relevant benchmark suite, GPU architects need an architectural simulator to study multi-GPU collaborative execution behavior and evaluate multi-GPU system designs. Existing publicly-available GPU simulators, such as GPGPU-Sim [33] and Multi2Sim [23], were developed for single-GPU platforms and do not support state-of-the-art multi-GPU platforms. This is because: 1) Existing GPU simulators simulate out-dated GPU architectures. Newer GPUs add special features such as System-Level Atomics and GPUDirect [34] are required to facilitate collaborative execution across multiple GPUs. 2) Existing simulators lack modularity, hindering multi-GPU system modeling. 3) Existing simulators are inefficient in terms of simulation speed. A few seconds of execution on an actual GPU may take a few days to simulate. The increased number of components in multi-GPU systems can further exacerbate performance issues. These limitations of current simulation tools impede the design of future multi-GPU system solutions.

To better analyze multi-GPU system performance, we present MGPUSim, a GPU simulator tailored for multi-GPU platform simulation. MGPUSim faithfully simulates the AMD GCN3 ISA [35], a state-of-art, and widely adopted GPU ISA. MGPUSim features high flexibility, high extensibility, and parallel simulation. High configurability allows users to model platforms with different properties, such as the instruction scheduling algorithm, the memory hierarchy, and the number of GPUs in the system. High extensibility enables developers to add new simulator features without extensive modification. Parallel simulation can accelerate the simulation without losing accuracy.

Equipped with the right set of tools for performance evaluation, we design solutions that improve multi-GPU computing performance. Today, multi-GPU systems mainly embrace a discrete multi-GPU model, one which that requires the application developer to control each GPU. To provide a simpler programming model to utilize the computing power of multi-GPU systems, programmers need to rewrite the application. Recent work [16, 17] has proposed moving to a unified multi-GPU model, which allows a single-GPU application to run directly on a multi-GPU platform. The unified multi-GPU model provides developers with a single GPU interface that hides the complexity of the multiple GPUs in the system. In unified multi-GPU systems, the driver and the hardware is responsible for controlling the GPUs. However, the driver and the hardware lack locality information and cannot place a thread and its required data in the same GPU, causing excessive inter-GPU communication. To better leverage the programmer's knowledge to reduce inter-GPU communication, we introduce a new multi-GPU computing API, named the Locality API, which combines the advantages of a discrete multi-GPU model and a unified multi-GPU model. The Locality API enables single-GPU applications to run on a multi-GPU platform by introducing a few

Table 1.1: Contributions of this Thesis.

	CPU-GPU Collborative Computing	**Multi-GPU Collaborative Computing**
Design Patterns	3 non-overlapping patterns, 8 overlapping patterns	6 patterns
Workloads	Hetero-Mark	MGPUMark
Simulators	Multi2Sim-HSA	MGPUSim
System Designs	Priority-based PCIe Scheduling	Locality API, PASI

new API calls and does not require modification to the GPU code.

To further reduce inter-GPU communication, we also propose Progressive Page-Splitting Migration (PASI). PASI can reduce inter-GPU communication with a fully programmer-transparent mechanisms. PASI utilizes page migration to move data to the GPU that uses the data, as page migration can better utilize data locality. To avoid extensive page migrations, PASI employs a memory coherency protocol to allow read-sharing pages to be duplicated on multiple GPUs. PASI also allows dynamic page sizes and page splitting. When false sharing occurs, GPUs can split pages in half and only migrate the half accessed by the remote GPU, reducing inter-GPU traffic and the likelihood of future false sharing. By progressively splitting large pages into smaller ones and migrating the smaller page to the desired GPUs, the GPU hardware can gradually align the memory layout with the computing thread placement, reducing data movement and improving performance.

1.3 Contributions

We summarize the contributions of this dissertation in Table 1.1. For both CPU-GPU Collaborative Computing and Multi-GPU Collaborative Computing, we first analyze execution patterns and implement a set of standard workloads. We also develop system and performance modeling tools, including Multi2Sim-HSA and MGPUSim. Finally, we propose software-based and hardware-based solutions to simplify programming and to improve system performance. To be specific, the contributions of this Ph.D. dissertation include:

- We introduce Collaborative Heterogeneous Computing. Unlike the traditional heterogeneous computing paradigms, Collaborative Heterogeneous Computing allows multiple devices to

work concurrently by leveraging fine-grained communication and synchronization mechanisms.

- We characterize CPU-GPU collaborative execution patterns that can guide both software and hardware design. We also introduce Hetero-Mark, a CPU-GPU collaborative execution benchmark suite that supports exploration of CPU-GPU collaborative execution patterns. We provide at least one workload for each CPU-GPU collaborative execution pattern.
- We introduce Multi2Sim-HSA, an HSA instructor emulator that emulates the HSA runtime API and HSA intermediate language (HSAIL) instructions.
- We introduce a priority-based PCIe scheduling mechanism that can improve the computing resource utilization in cloud-based, multi-tenant, CPU-GPU heterogeneous computing systems.
- We categorize multi-GPU collaborative execution patterns. We also implement MGPUMark, a benchmark suite that evaluates multi-GPU platform performance.
- We design and develop MGPUSim, a GPU simulator that features high-performance, high-flexibility, multi-GPU system simulation.
- We introduce the Locality API, an API extension to existing GPU-programming APIs. The Locality API allows programmers to run a single-GPU application on multi-GPU systems with the minimum modifications to the host code and no modification to the GPU code. The GPU program modified with the Locality API can achieve the performance close to hand-tuned multi-GPU applications.
- We also introduce Progressive Page-Splitting Migration (PASI), a programmer transparent mechanisms that allows the hardware to split pages during the page-migration process.

Chapter 2

Background

In this chapter, we introduce the background of the thesis. We begin by introducing GPU programming frameworks including OpenCL, CUDA, C++ AMP, HC++, and HIP. We then introduce HSA—an intermediate layer that delivers interoperability between high-level programming frameworks and low-level GPU drivers and GPU devices. Next, we introduce Radeon Open Compute Platform (ROCm), an AMD GPU programming platform that uses HSA to support several GPU programming frameworks. We also introduce GPU architectures and discuss how GPU hardware architectures support software GPU programming models. Finally, we discuss the technologies that enable CPU-GPU communication and inter-GPU communication.

2.1 GPU Programming Frameworks

A GPU application usually starts with a host program running on a CPU. A host program invokes a set of APIs provided by a runtime library to control the GPUs. Besides the host program, a programmer would also need to write a special program in a programming language designed for GPUs (e.g., OpenCL C) so that the GPUs can perform predefined operations. A set of GPU-controlling APIs, combined with a GPU programming language, is called a GPU programming framework. CUDA and OpenCL represent traditional GPU programming frameworks, while more recent GPU programming frameworks with unique features (e.g., HC++ and HIP) have been proposed. In the rest of this section, we introduce OpenCL in detail and then we discuss differences between other programming frameworks and OpenCL.

2.1.1 OpenCL

OpenCL is a standard for implementing data-parallel algorithms. OpenCL is popular mainly because it is an open standard and a large number of vendors support it. Once an OpenCL program is written, the program can run on CPUs, AMD GPUs, and NVIDIA GPUs. Moreover, thanks to its generality, other types of devices such as DSPs and FPGAs can also run OpenCL programs.

OpenCL [36] was initially developed by Apple when OpenCL 1.0 was released in 2009. As Apple deprecated OpenCL in favor of it in-house GPU programming framework Metal, Khronos Group continued with the standard development. The OpenCL standard matured in the release of OpenCL 1.2 in 2011, adding a large number of convenient features to the OpenCL 1.0 standard. OpenCL 1.2 is still widely used nowadays, regardless of the availability of updated release versions of OpenCL. We mainly discuss the OpenCL 1.2 programming model before we introduce new features of the later versions of OpenCL that can simplify CPU-GPU and multi-GPU collaborative execution.

2.1.1.1 OpenCL Host API

An OpenCL host program usually starts by discovering devices available on the platform. Typically, a device can be a CPU or a GPU that is capable of executing compiled OpenCL code.

OpenCL code is loaded from a source file. The source file is compiled into the device ISA at runtime. This just-in-time (JIT) compilation process guarantees that the OpenCL code can be executed on any supported device. Alternatively, a programmer can load a GPU program from a pre-compiled binary file. Using a pre-compiled binary file saves compilation time but compromises portability, as the application can only run on the devices that match the ISA that binary uses.

OpenCL uses command queues to manage tasks such as memory copies and compute kernels. A command queue maintains a list of tasks in a queue and the tasks from one queue are always executed in-order without overlap. Tasks from different queues can execute in parallel. For memory copy tasks, OpenCL provides a `clEnqueueCopyBuffer` API to add a memory copy command into a command queue. For compute tasks, a programmer can use the `clEnqueueNDRangeKernel` API.

A kernel launch forms a 1-D, 2-D, or 3-D NDRange, as shown in Figure 2.1. An NDRange is composed of a certain number of work-items. Programmers need to specify the number of work-items of an NDRange on each dimension in the host program. A work-item is similar to a CPU thread and maintains its own set of registers. All the work-items of an NDRANGE can access the

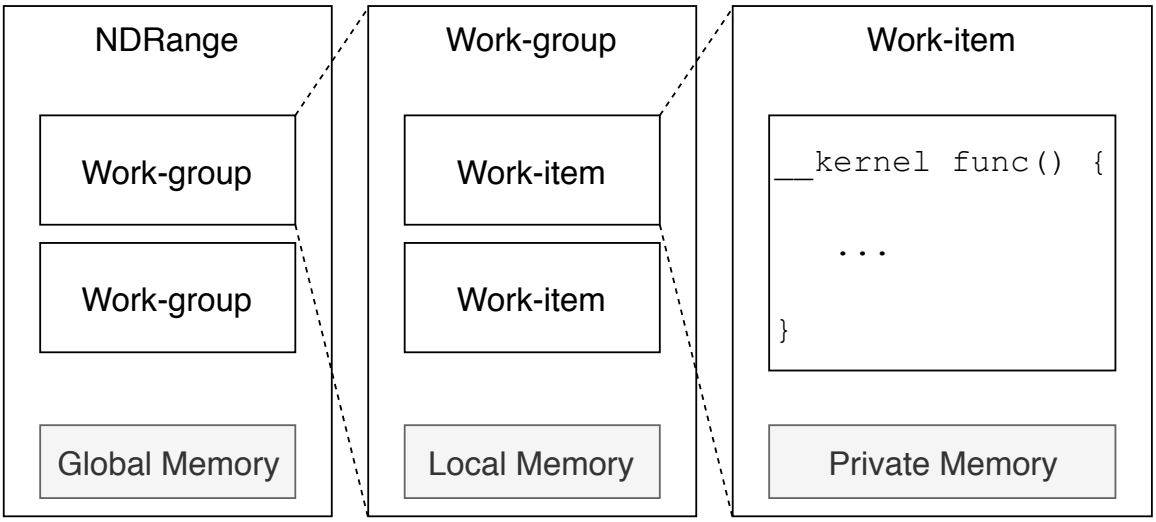


Figure 2.1: OpenCL Kernel Execution Model

Global Memory, while each work-item can also access a memory space called *Private Memory*. A certain number of the work-items form a work-group. All the work-items in a work-group share a *Local Memory* space. Also, the work-items in a work-group can synchronize with the help of barriers. On the contrary, there is no way to synchronize all the work-items in a kernel. In the case that kernel-level synchronization is needed, the only solution is to stop the kernel and start a new kernel after the synchronization.

2.1.1.2 OpenCL C

OpenCL provides OpenCL C, a C-like programming language for implementing GPU kernels. It inherits most of the features such as variable declarations, arithmetic computation, branching, and function calls, from C, while adding unique features for GPU programming.

Listing 2.1: Sample OpenCL Kernel.

```
__kernel void vecAdd(
    __global double *A,
    __global double *B,
    __global double *C,
    const unsigned int N
) {
    int index = get_global_id(0);
    if (index < N) {
        C[index] = A[index] + B[index];
    }
}
```

Listing 2.1 shows a sample kernel that can add two vectors elementwise. The kernel is defined as a function with the extra __kernel keyword. The kernel function is written to represent the behavior of each individual work-item in the kernel. In the function argument list, we see that the function takes three pointer arguments named as A, B, and C. The type of the pointer arguments are prefixed with the __global keyword, denoting that the memory that the pointers point to is located in global memory. The kernel also takes a fourth argument N for the vector length. In the kernel body, a work-item that executes the kernel first retrieves the ID of the current work-item, so that the work-item knows which element to access. Then, similar to a standard C program, the work-item reads data from vectors A and B and store the sum in vector C. The sum and assign operation is protected in an if clause to prevent out-of-range memory accesses, as out-of-range access can produce undefined behavior or even corrupt the kernel.

2.1.1.3 New Features Since OpenCL 2.0

The OpenCL version released after OpenCL 1.2 is OpenCL 2.0, providing non-traditional features that improve programmability. Some of the features aim at bringing CPUs and GPUs closer and giving GPUs more autonomy. Among those features, we discuss Shared Virtual Memory and Nested Parallelism.

Shared Virtual Memory: Shared Virtual Memory (SVM) allows programmers to allocate a memory space without specifying where the memory is located. After allocation, both the CPU and the GPU can access the memory. Using SVM relieves the programmer from manually copy memory from one device to another and shifts the memory management responsibility to device drivers and hardware.

OpenCL provides two types of SVM: Coarse-grained SVM and Fine-grained SVM. Coarse-grained SVM shares memory at the granularity of whole buffers. Programmers would need to call `clEnqueueSVMMap` before a CPU starts to access the shared data and call `clEnqueueSVMUnmap` after the CPU is done with accessing the shared data. Under the hood, memory copies are performed by the device driver when the map and the unmap functions are invoked. Fine-grained SVM further relieves the requirement of explicit memory mapping. Data is shared at the granularity of each load and store operation. Fine-grained access allows CPUs and GPUs to access the same buffer at the same time.

Nested Parallelism: In OpenCL 1.2, only CPUs can launch kernels to GPUs, while a GPU cannot recursively launch a kernel. However, in many cases such as recursion algorithms, a GPU may have the information needed to start another data-parallel workload. Transferring the data back to a CPU and let the CPU to launch another kernel introduces an extra communication round trip and hence, is not efficient. To address this problem, OpenCL 2.0 introduces Nested Parallelism, empowering a GPU to launch kernels to itself. A new kernel is considered as part of the parent kernel that starts the child kernel and the parent kernel continues execution after the child kernel returns.

2.1.2 CUDA

CUDA is a proprietary GPU programming framework that is developed by NVIDIA for NVIDIA GPUs. The CUDA programming model is very similar to OpenCL, but the terminologies are different. Work-groups and work-items are called blocks and threads in CUDA, respectively.

Listing 2.2: Example code of launching a kernel with CUDA.

```
vector_add<<<dimGrid, dimBlock>>>(output, in1, in2);
```

CUDA allows programmers to write code for CPUs and GPUs in the same file. It also features a convenient way of launching a kernel with the <<< ... >>> syntax. This feature allows programmers to write less verbose code.

2.1.3 C++ AMP and HC++

Microsoft introduces its own GPU programming framework named C++ AMP [37]. C++ AMP does not introduce a new GPU programming language but relies on lambda expressions that are defined by the standard C++ to program GPUs. HC++ is an AMD programming framework that follows the Microsoft C++ AMP standard. ROCm includes `hcc`, an open-source compiler based on the Clang/LLVM compiler infrastructure, to compile HC++ code.

An interesting feature of HC++ and C++ AMP is `array_view`. It creates a viewport for a native CPU buffer and allows both the CPU and GPU to access the data. Device-side memory allocations, memory transfers, and synchronization are all performed under-the-hood by runtime libraries and drivers, relieving the developer from dealing with explicit memory management. In this thesis, we evaluate the performance of `array_view` to study if GPU drivers can handle memory transfers efficiently.

2.1.4 HIP

HIP is a GPU programming framework that emphasizes portability between the AMD ROCm platform and the NVIDIA CUDA platform. Programs that are written in HIP can run on both AMD and NVIDIA devices. With `hipify`, CUDA-based application source code can be automatically converted to HIP-based source code.

2.2 HSA

Heterogeneous System Architecture (HSA) is a standard proposed by the HSA foundation. The HSA standard defines the HSA Intermediate Language HSAIL, a low-level runtime API, and a set of system specifications for the hardware that follows the HSA standard. HSA provides game-changing features that allow CPUs and GPUs to work seamlessly together and give GPUs a higher-level of autonomy.

2.2.1 HSAIL and BRIG

HSAIL is an intermediate language that lies between high-level GPU programming languages and low-level machine ISAs. A GPU programmer would first convert the GPU kernel written in a high-level programming language such as OpenCL to HSAIL at compile-time. Then the HSA runtime API will use a vendor-provided finalizer to convert HSAIL code to the machine ISA at runtime. As HSAIL is closer to the device ISA, finalizing HSAIL is much faster than compiling a kernel written in a high-level GPU programming language. Besides, as HSAIL is device independent, HSAIL code can be finalized into different GPU ISAs, guaranteeing code portability. HSAIL is defined in an assembly-style pure-text format. To ease the implementation of finalizers, HSA also defines BRIG, the binary format representation of HSAIL.

2.2.2 HSA Signals

HSA defines a new feature called HSA Signals, enabling a CPU and a GPU to wake up each other and send messages. Values of HSA signals can be both changed from the CPU side or the GPU side, and both the CPU and the GPU can wait for the signal value to change. On the CPU side, signals are accessible with HSA APIs, and on the GPU side, signals can be accessed with special HSAIL instructions. It provides a new way for CPUs to dispatch tasks to GPUs. It also allows GPUs to send early results to a CPU before the whole kernel completes execution.

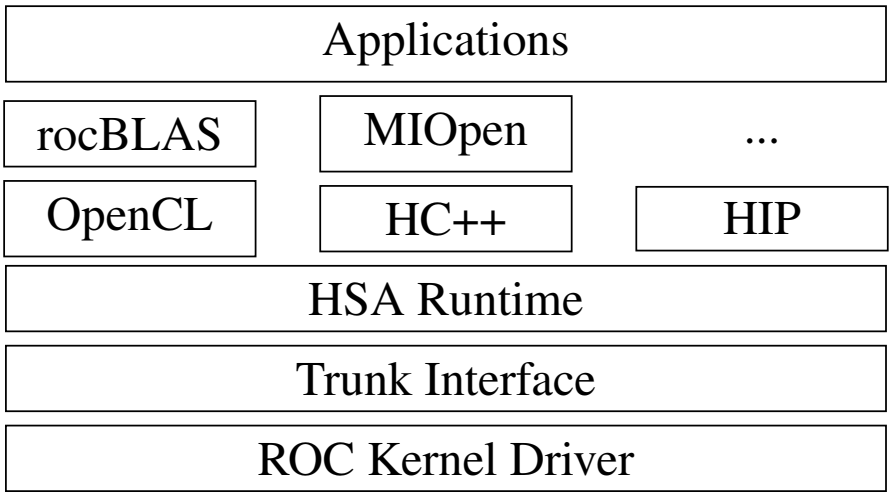


Figure 2.2: Organization of the ROCm platform

2.2.3 AQL Queues

Traditionally, command queues are managed by the hardware directly. The hardware defines a few types of commands it supports. The host program can only use API calls to send pre-defined types of commands to command queues. However, it is difficult to introduce a new customized type of command as both the hardware and the API are hard to change.

HSA solves the extensibility problem by replacing hardware-controlled command queues with memory-based AQL queues. As the queue is in memory, programmers can use easy-to-change software to parse the content of the queued commands. A CPU can use AQL queues to launch kernels and send customized types of commands to GPUs. Device vendors can define customized packet formats to support new types of tasks. With Shared Virtual Memory, a GPU can also write into the memory of AQL queues, allowing the GPU to dispatch tasks to itself, to another GPU, or even to a CPU.

2.3 ROCm

AMD introduces Radeon Open Compute Platform (ROCm) to provide solutions for high performance and ultra-scale computing on AMD GPUs. The 1.0 version of ROCm was released in April 2016. At the time of this writing, the most recent version of the ROCm platform is 2.7. Note that ROCm is not a GPU programming framework since it does not specify a GPU programming language, nor a set of GPU-controlling APIs. ROCm provides the underlying functionalities for

high-level GPU programming frameworks so that programmers can leverage ROCm in a framework-neutral way. GPU programming frameworks can be implemented on top of the ROCm platform. The GPU programming frameworks that are supported on the ROCm platform serve as good examples of how frameworks can be implemented with the HSA runtime and on the ROC platform.

We summarize the organization of ROCm in Figure 2.2. At the top of the figure is user applications, which are programmed using the services provided by ROCm. Programmers can use the ROC libraries, such as rocBLAS and MIOpen, to embedded well-defined algorithms in their program. The programmmer can also go one level lower and use high-level programming frameworks to write application-specific GPU kernels. Currently, ROCm mainly supports three GPU programming frameworks: i) OpenCL, ii) HC++, and iii) HIP.

The OpenCL, HC++, and HIP frameworks running on ROCm are implemented with the HSA runtime APIs. The ROC platform enables interoperability across these three frameworks. For example, GPU memory that is allocated by one framework can be used in kernels written in another framework. In addition, programmers can choose to use the closer-to-hardware HSA runtime API directly in their program, especially if precise hardware control is required. At the bottom of the software stack is the ROC kernel driver and a thin wrapper for the kernel driver called the ROC Trunk Interface.

2.4 GPU Architecture

Next, we provide an overview of GPU architectures, emphasizing how hardware handles the aforementioned software features. We use the details of the AMD R9 Nano GPU [38] as an example, especially since our MGPUSim models the AMD R9 Nano GPU by default.

2.4.1 Kernel Dispatching

When a GPU program needs to launch a kernel on a GPU, the program incorporates the driver to send a command to the GPU. The command is sent by writing a special memory region in an AQL Queue and ringing a special HSA Signal called doorbell signal. The GPU employs a Command Processor (CP) to process all the incoming commands from CPUs.

Once a CP receives the doorbell signal, the CP allocates an Asynchronous Compute Engine (ACE) to dispatch the kernel to the Compute Units (CU) of the GPU. The CU is where

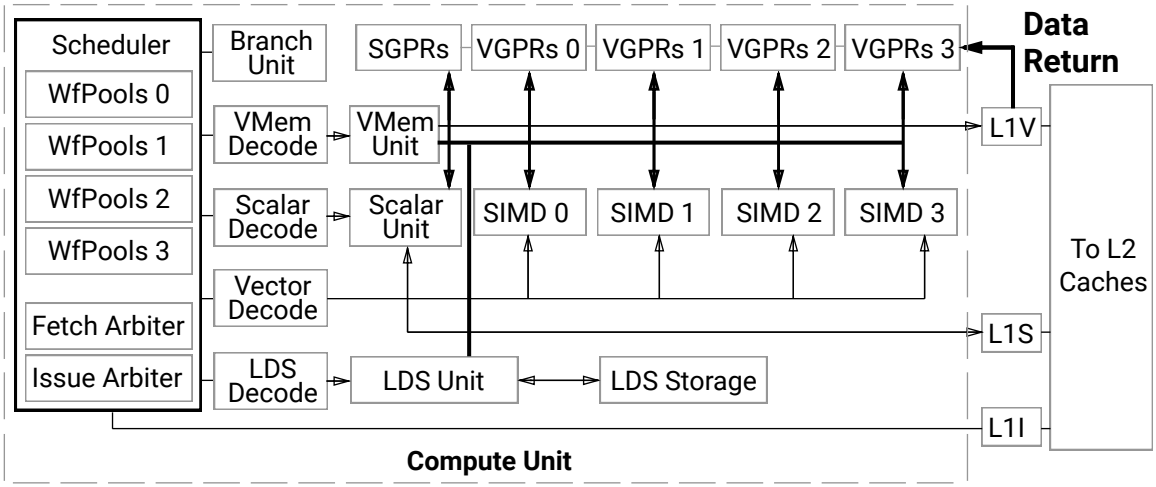


Figure 2.3: Organization of a GCN3 Compute Unit.

work-item instructions are executed. There are multiple ACEs in one GPU, allowing a GPU to run multiple kernels at the same time.

When an ACE starts to dispatch a kernel, it forms an NDRange of the kernel. The ACE assigns the work-groups of the NDRange to the CUs of the GPU. All the work-items from one work-group must run on the same CU, while a CU can run work-items from multiple work-groups at the same time. As a work-group is composed of a certain number of wavefronts (a group of 64 work-items), the ACE can dispatch one wavefront per cycle. In general, the number of work-groups in an NDRange is more than the CUs in a GPU can handle at the same time. In this case, the ACE fills all the CUs to their maximum capacity first and pause dispatching. After any CU finishes the execution of a work-group, the ACE dispatches another work-group to that CU. Since the work-groups from an NDRange are not executed at the same time, a GPU cannot perform synchronization for all the work-items in an NDRange.

2.4.2 Compute Unit

The Compute Units (CUs) are where the instructions are executed on a GPU. An AMD R9 Nano GPU has 64 CUs. The number of CUs in lower-end GPUs or APUs (integrated CPU-GPU chips) tends to be lower than 64. As shown in Figure 2.3, a CU is equipped with a scheduler, a set of instruction decoders, a set of register files, a set of ALUs (4 SIMD units and a scalar unit), and memory interfaces. The scheduler is responsible for deciding which wavefront can fetch instructions from and which wavefront can issue instructions. The decoders decode instructions and feed the

decoded instructions to the execution units.

There are three types of execution units in a CU. SIMD units are the most important execution units in a CU as they can execute vector instructions. Vector instructions follow a SIMD execution fashion and work-items from the same wavefront execute the same instruction at the same time. Another execution unit is the scalar unit, which can execute a single instruction for the whole wavefront at one time. The last type of execution unit is the Branch Unit and it is responsible for executing branch instructions.

There are also a few memory interfaces in a CU. The LDS unit executes Local Data Share instructions and interacts with the local memory space. The vector memory unit can load and store 64 data elements by executing one instruction. Finally, the scalar unit also duplexes as a scalar memory unit, which loads a single piece of data for the whole wavefront by executing a scalar memory instruction. A scalar unit is especially useful for loading and calculating addresses as all the work-items in a wavefront usually share the same buffers.

A CU receives requests for wavefront dispatching from ACEs. There are 4 limiting factors that determine if a CU can accept a wavefront or not. First, each wavefront needs to be associated with a SIMD unit and each SIMD unit can only handle at most 10 wavefronts at a time. Second, the CU needs to allocate register space in the vector register files. The CU can accept the wave only if there is enough space to hold the vector registers required by the wavefront. Third, as the wavefront needs scalar registers as well, the CU needs to guarantee that there is enough space to hold the scalar registers in the scalar register file. Finally, the CU also needs to guarantee there is enough Local Data Share space to hold the local memory needed to run the wavefront.

Assuming we have a few wavefronts dispatched to a CU and are waiting to be executed, the scheduler needs to decide which wavefront can fetch instructions and which wavefront can issue instructions for execution. The fetch arbiter selects one SIMD unit in a cycle in a round-robin fashion and within the wavefronts that are associated with the SIMD unit, the fetch arbiter selects the wavefront that has not fetched any instruction for the longest time. The issue arbiter also selects one SIMD per cycle in a round-robin fashion. Different from the fetch arbiter where only one wavefront can be selected, the issue arbiter can select as many wavefronts as possible, as long as the selected wavefronts follow the folloing criteria: 1. No two selected wavefronts can occupy the same next-level unit (branch unit or the decoders). 2. If one wavefront has an instruction executing in the instruction pipeline, the wavefront cannot be selected. 3. If the wavefront has special instructions such as barriers to execute, the instruction can be evaluated in the scheduler and does not need to be issued.

Each SIMD unit has 16 single-precision ALUs (i.e., lanes), meaning that it takes 4 cycles

to execute a wavefront-level instruction (one wavefront is composed of 64 work-items). The number of double precision units is usually only a fraction of the number of single-precision units. In an AMD R9 Nano GPU, there are 2 double-precision units per SIMD unit. Therefore, the execution of a double-precision instruction from a 64-work-item wavefront takes 32 cycles. Given that an AMD R9 Nano GPU has 64 CUs and each CU has four 16-lane SIMD units, an R9 Nano GPU can execute $64 \times 4 \times 16 = 4096$ instructions per cycle. Considering that R9 Nano GPUs run at 1 GHz frequencies, one GPU can run $4096 \times 1G = 4T$ instructions per second. In addition, as R9 Nano GPUs support fused multiply-add instructions that combine one multiplication operation and one addition operation in one instruction, an R9 Nano GPU can deliver a $4T \times 2 = 8TFlops$ theoretic computing capability.

2.4.3 Memory Hierarchy

GPUs typically employ a 2-level cache design. Each L1 vector cache serves a CU. L1 vector caches use write-through polices and each L1 cache sends a write request to an L2 cache when it receives a write request from the CU. Every four CUs also share one L1 instruction cache and one L1 scalar cache. Both the instruction cache and the scalar cache are constant caches and cannot handle write requests.

GPU L2 caches are usually memory-side caches. Each L2 cache serves a memory bank of a certain address range. In total, there are 4 to 8 L2 caches that cover the whole address range of the GPU memory. L1 and L2 caches are connected through a crossbar interconnect. As each L1 cache covers the whole memory address space, one L1 cache needs to connect to all the L2 caches to fetch data. L2 caches usually use a write-back policy. As L1 cache write backs update the L2 cache immediately and one cacheline can only reside in one L2 cache, L2 caches always have the most recent data. Given that GPUs use a relaxed memory consistency model, a cache coherency protocol is not required.

2.5 Multi-GPU System Solutions

With the growing amount of data to be processed by GPUs, a single GPU can no longer meet users' requirements. GPU vendors start to support multi-GPU computing and multi-GPU communication. In the traditional sense, each GPU cannot access the data that is located on another

GPU during kernel execution. Programmers have to stop running kernels and use peer-to-peer memory copy to explicitly copy the memory from one GPU to another.

To avoid explicit memory copying, both NVIDIA and AMD allow their GPUs to use Unified CPU-GPU Memory spaces (i.e., Unified Memory). Unified Memory does not only simplify the CPU-GPU memory management, but it also simplifies data management on multi-GPU systems. A kernel can access the data located on another GPU without explicit data movement. The hardware performs the operation under the hood, using two main approaches: Demand Paging and Direct Cache Access.

When a GPU accesses a memory page (4KB to 2MB consecutive memory space), it needs to translate the virtual address to the physical address first. The GPU may not have this page present on its own memory and can trigger a page fault to be handled by the operating system and the GPU driver. The GPU driver requests the page to be migrated from one device to the GPU that needs the page. Once the page is physically present in the GPU, the GPU can continue accessing the page. The page migration process introduces long latencies caused by device synchronization and large data movement. In addition, when more than one device needs the page, the page needs to be migrated back-and-forth across the devices, producing a large amount of inter-device traffic.

To avoid unnecessary page migrations, both NVIDIA and AMD GPUs support Direct Cache Access. One GPU can access a piece of data that is located on another GPU directly without migrating the page. This approach moves the data using a much finer granularity (usually a 64-byte or 128-byte cacheline). Also, the inter-GPU data is usually not cached to prevent coherency issues. This approach is more suitable for occasional inter-GPU accesses as it introduces less GPU traffic. However, Direct Cache Access cannot utilize spatial locality and when the data is frequently accessed remotely, it may have lower performance than Demand Paging.

Other than memory management approaches, GPU vendors are also investing in high-bandwidth inter-GPU fabrics. To facilitate inter-GPU communication, NVIDIA introduces NVLink [39], a high-bandwidth point-to-point interconnect fabric. In the first version of NVLink, each link provides 20 GB/s for each direction of communication. Each NVIDIA P100 GPU [40] has 4 NVLink slots, allowing up to an 80GB/s out-going and 160GB/s bidirectional communication rate. The second version of NVLink upgrades the bandwidth of each link to 25GB/s. The NVIDIA V100 GPU [41], released together with NVLink-V2, has 6 NVLink slots, rendering an up-to 150GB/s out-going and 300GB/s bidirectional communication rate. At the same time, AMD relies on both the PCIe-v4 interconnect and the high bandwidth xGMI interconnect [42] for inter-GPU communication. The inter-GPU xGMI fabric is to be introduced in AMD next-generation GPUs.

Chapter 3

Related Work

In this chapter, we review prior work related to the research areas covered in this dissertation. We first introduce existing CPU-GPU heterogeneous computing benchmark suites and GPU performance modeling tools. Then we discuss related work on designing efficient CPU-GPU collaborative computing systems and multi-GPU collaborative computing systems.

3.1 Heterogeneous Computing Benchmark Suites

Benchmark suites are essential tools used in computer architecture research. Benchmark suites provide standard workloads, allowing researchers to compare different system design solutions. Commonly used CPU benchmarks include Linpack [43], SPEC [44] and PARSEC [45, 46]. With the increasing popularity of general-purpose GPU computing, GPU benchmarks suites have also become widely available.

Rodinia [32, 47] is one of the most popular GPU benchmark suites. The suite includes a wide range of workloads, according to the Berkeley Dwarfs design patterns [48]. Rodinia initially supported CUDA and OpenMP implementations of the workloads. The developers of the suite later developed OpenCL implementations.

The **Parboil** [49] suite takes a similar approach by providing both OpenCL and CUDA implementations for commonly used algorithms. Parboil workloads cover broad domains, including astronomy, biomolecular simulation, fluid dynamics, image processing, and dense and sparse linear algebra. The goal of Parboil workloads is to evaluate computing systems in terms of computing throughput, as well as many other aspects such as memory bandwidth, floating-point throughput, latency tolerance, and cache effectiveness.

SHOC [50] is a collection of benchmark programs for testing the performance and stability of a system containing GPUs and multi-core processors. Rather than only focusing on the performance of a single GPU computing node, SHOC evaluates the performance of GPU clusters. SHOC organizes benchmarks into two levels. Level 0 includes microbenchmarks that can test system capabilities, such as peak FLOPs and inter-node communications bandwidth, while Level 1 includes benchmarks that implement well-known algorithms, such as matrix multiplication and array sorting. Similar to Rodinia and Parboil, the SHOC suite also provides both OpenCL and CUDA implementations.

Polybench [51] is another widely used benchmark suite for GPU research. Initially, Polybench only has C implementations for testing CPU parallel computing performance. Polybench developers later extended the suite with GPU implementation. Polybench provides a wide range of linear algebra workloads that can extensively stress the computer capability and the memory bandwidth of GPU systems. Polybench is implemented with each workload in a single file and short source codes, which simplifies compiler optimizations.

ViennaCL [52] is an accelerated linear algebra library that abstracts the underlying programming framework and platform. Programs that use ViennaCL can run on CPUs, AMD GPUs, and NVIDIA GPUs. ViennaCL also doubles as a linear algebra benchmark suite. The goal of the ViennaCL benchmark suite is to help programmers identify the best programming framework and platform to use for a selected class of applications or algorithms.

FinanceBench [53] mainly focuses on financial-related workloads. It evaluates the performance of GPUs on performing the BlackScholes, Monte-Carlo, Bonds, and Repo (price calculation for securities repurchase agreement) algorithms. Other than standard OpenCL and CUDA implementations, FinanceBench also includes implementations in directive-based GPU programming schemes, including OpenACC [54] and HMPP [55].

LoneStar [56] explores the performance of parallel computing devices when processing graphs. LoneStar provides both CPU and GPU implementations. The benchmarks from the LoneStar benchmark suite reveal that parallel computing devices can exploit parallelism present in graph-processing workloads. The level of parallelism increases with the increase of the problem sizes.

Recently, GPU vendors have added unique features (e.g., Unified Memory and System-Level Atomics) that simplify CPU-GPU and multi-GPU collaborative computing. Recent benchmark suites have also started to explore new features and evaluate CPU-GPU interaction.

Valar [57] is a benchmark suite that mainly quantifies the performance of features supporting host-device interaction. Valar categorizes GPU workloads into one of three groups: 1)

Computation Pipeline Implementations, 2) Multi-Device Decoupled Implementations, and 3) Multi-Device Coupled Implementations. The three groups summarized by Valar map well to the three traditional CPU-GPU interaction patterns summarized in Hetero-Mark. Valar also includes five benchmarks that test different types of host-device interaction behavior.

NUPAR [58] is a benchmark suite that evaluates new GPU features, including nested parallelism [59], concurrent kernel execution [60], and shared host-device memory [61]. It also evaluates new instructions for precise computation and data movement. NUPAR includes eight benchmarks. Five of the benchmarks are implemented in CUDA, while the three remaining benchmarks are implemented in OpenCL.

Chai [62] is another benchmark suite that was developed in parallel with Hetero-Mark. Similar to Hetero-Mark, the main goal of Chai is also explore CPU-GPU Collaborative Computing. Chai categorizes CPU-GPU collaborative computing using three main patterns: i) Data Partitioning, ii) Fine-grain Task Partitioning and iii) Coarse-grain Task Partitioning. Chai includes 14 benchmarks. Eight benchmarks possess the Data Partitioning Pattern. Chai also provides three benchmarks each the Fine-grain Task Partitioning and the Coarse-grain Task Partitioning patterns.

Tartan [63, 64] is a benchmark suite for multi-GPU computing. Tartan includes microbenchmarks, intra-node scale-up applications, and inter-node scale-out applications. The main target of the Tartan benchmark suite is to evaluate new inter-GPU communication fabric designs.

Traditional GPU benchmark suites mainly focus on GPU performance, ignoring the performance of the CPU and CPU-GPU communication. Recent GPU benchmarks start to explore CPU-GPU interaction and multi-GPU interaction. However, researchers have limited choice when selecting CPU-GPU and multi-GPU collaborative computing workloads. Moreover, few benchmarks evaluate fine-grained CPU-GPU and inter-GPU communication (e.g., GPUDirect [34]), making designing architectural solutions that accelerate fine-grained communication challenging to devise. In this dissertation, we present Hetero-Mark and MGPUMark, two workloads that primarily target evaluation of CPU-GPU and multi-GPU fine-grained communication, respectively.

3.2 GPU Simulators

In addition to benchmark suites, high-quality simulators are essential tools used extensively in computer architecture research. Since producing new hardware is very expensive, researchers typically evaluate design tradeoffs using a simulator pre-silicon before integrating the design feature

in the final hardware. Popular GPU simulators include GPGPUSim [33], Multi2Sim [23, 24], and the gem5 AMD GPU model [65].

GPGPUSim [33] is a GPU simulator that models the NVIDIA Fermi architecture and supports the CUDA 4.0 runtime API. GPGPUSim runs PTX, an intermediate language between the CUDA programming language and the real ISA. Power et al. [66] connect GPGPUsim to the gem5 CPU simulator to form a heterogeneous CPU-GPU simulator. MAFIA [67] extends GPGPUSim to support concurrent workloads running on one GPU. MOSAIC [68] further extends the MAFIA simulator to support virtual-to-physical address translation and a unified CPU-GPU address space.

Multi2Sim [23] is a simulator infrastructure that can simulate a series of CPU and GPU architectures. It supports detailed timing simulation of CPUs that run the X86 instruction set. For GPUs, Multi2Sim models the AMD Evergreen and Southern Islands series GPUs. Recently, Multi2Sim started to support modeling the NVIDIA Kepler GPUs [24]. The versatility of Multi2Sim fits our requirements of building an HSA emulator that integrates both CPUs and GPUs in one system.

More recently, Gutierrez et al. [65] published the **AMD gem5 GPU model** that is capable of simulating GCN3-based GPUs. This simulator is based on the gem5 simulator framework. The main target of the AMD gem5 GPU model is APU (CPU and GPU on the same chip) devices [69]. It is highly accurate since it can generate error-free dynamic instruction counts and SIMD utilization statistics.

Multi-GPU systems are widely used in many fields to deliver higher computing power. There is a growing need to design architectural solutions for multi-GPU systems. However, existing GPU simulators mainly only consider single-GPU systems. Extending existing simulators to model multi-GPU systems is tedious due to a lack of modularity and low simulation performance. In this dissertation we develop MGPUSim, a highly accurate, highly flexible, high-performance multi-GPU simulator that enables multi-GPU system design evaluation.

3.3 Multi-GPU System Design

Building multi-GPU systems is a promising solution to increase the computing capabilities of a computer node. However, the adoption of multi-GPU solutions is constrained by both the required programming model and the inherent inefficient memory management. Recently, researchers have developed several solutions to simplify multi-GPU programming and improve multi-GPU system efficiency.

Kim et al. [16] proposed Scalable Kernel Execution (SKE), allowing a single kernel to execute on multiple GPUs as if there is only one GPU on the platform. This unified-GPU design can significantly reduce the burden of programmers. A GPU program that was written for a single-GPU platform can directly run on a multi-GPU platform without modification. SKE relies on the GPU driver and the GPU hardware to distribute memory and compute workloads to multiple GPUs. GPU-GPU remote direct memory access (RDMA) enables each GPU to access the combined memory space across multiple GPUs.

Ziabari et al. [17] proposed the Unified Memory Hierarchy (UMH). UMH enables a unified CPU-GPU memory address space and eliminates the need for memory copies across devices. UMH uses the large GPU DRAMs as the caches of the system memory. This work also introduced the NMOESI protocol to maintain cache coherency between the CPU and the GPU.

Multi-Chip-Module GPUs (MCM-GPU) [25] is a promising solution that scales GPU performance beyond die-size limitations. MCM-GPU encapsulates multiple GPUs in the same package. Similar to SKE, MCM-GPU integrates multiple actual GPUs under a single-GPU interface. A recent study [70] suggests that energy consumption is a significant constraint in MCM-GPU system implementations, and inter-chiplet communication consumes a large amount of energy. Arunkumar et al. [25] introduced an L1.5 cache to store the data that belongs to another GPU. They also combined the *first-touch based page mapping* policy with an improved Cooperative-Thread Array (CTA) scheduling policy to avoid remote memory access.

Milic et al. [71] proposed a NUMA-aware multi-GPU solution. The NUMA-Aware GPU solution redesigns the GPU runtime, the inter-GPU interconnect, and the GPU cache management solution to reduce inter-GPU traffic. Young et al. [72] points out that the GPU LLC is too small to cache remote data. They developed a hardware-based mechanism, Caching Remote Data in Video Memory (CARVE), to store the remote data in local memory.

CODA [73] is another solution that can match the compute placement with the data placement. CODA estimates the amount of data that needs to be accessed by each thread block. With the estimated data-access information, CODA places the associated data on the same GPU where the thread-block is executed. In addition, CODA also exploits an affinity-based thread-block scheduler to dispatch the thread-block to the GPU that owns the data required by the thread-block. By intelligently colocating both the thread-blocks and the data, CODA can significantly reduce inter-GPU traffic and improve performance.

Pure RDMA-based multi-GPU solutions suffer from high performance penalties caused by massive inter-GPU traffic. Recent solutions define coherency protocols to allow data duplication

and increase the percentage of local memory requests. However, existing solutions either require significant programmer involvement or underutilize memory. In this dissertation, we propose the *Locality-API*, which requires a minimal amount of modification to the system while still allowing programmers to improve performance. We also plan to explore *Progressive Page-Splitting Migration (PASI)* in this thesis, with the goal of reducing memory wastage due to statically partitioned DRAM caching.

Chapter 4

CPU-GPU Collaborative Computing

4.1 Multi2Sim-HSA

The goal of Multi2Sim-HSA is to emulate the execution of HSA applications faithfully. HSA defines both runtime APIs and a GPU kernel execution model (with HSAIL). We designed Multi2Sim-HSA to emulate both the CPU for the HSA runtime library and the GPUs for compute kernels. The host program can be either provided as an HSAIL kernel (see Section 4.1.2) or as an x86 executable file (see Section 4.1.3).

Multi2Sim-HSA embraces a vendor-neutral execution model. This model follows the HSA vendor-neutral framework design. Any device from any vendor should produce the same results for the same application. To guarantee vendor neutrality, Multi2Sim-HSA emulates the execution of the intermediate language, HSAIL, rather than a device ISA. Intermediate-language-level simulation compromises timing simulation accuracy [65, 24]. Since the primary goal of Multi2Sim-HSA is behavior emulation rather than detailed timing simulation, timing accuracy is not our first consideration.

In the rest of this section, we first give a brief introduction about how to use Multi2Sim-HSA. Then we discuss the emulator design.

4.1.1 Mult2Sim-HSA Execution

We start with a simple *HelloWorld* program sample. The HelloWorld HSA program, as listed below, has one function. The function adds the number 1 and the number 2 together, and prints the result to standard output.

```
function &m2s_print_u32 () (arg_u32 %integer) {};

kernel &main ()
{
    mov_u32 $s0, 1;
    mov_u32 $s1, 2;
    add_u32 $s2, $s0, $s1;

    {
        arg_u32 %num;
        st_arg_u32 $s2, [%num];
        call &m2s_print_u32 () (%num);
    }
};
```

The code is in the HSAIL format. To emulate it with Multi2Sim-HSA, a user needs to convert the HSAIL code to a binary format (i.e., BRIG) with the *HSAILTools* [74]. Assuming that the converted BRIG file is `helloworld.brig`, the user can run the HSA program on Multi2Sim-HSA with the following command:

```
m2s helloworld.brig
```

We deliver Multi2Sim-HSA as a Linux command line application. It follows the standard Linux command line format. The first token is `m2s`, the name of the simulator executable file. The second token is the name of the HSA program we want to simulate. Multi2Sim-HSA also accepts arguments to alter default options. Users can add arguments to Multi2Sim-HSA between the `m2s` and the HSA program name. The arguments for the HSA program follow the HSA program executable name in the command. For example, the command

```
m2s --hsa-debug-isa isa.debug helloworld.brig abc
```

passes two arguments to Multi2Sim-HSA, allowing Multi2Sim-HSA to dump instruction-by-instruction execution details into a file named `isa.debug` (see Section 4.1.4.4). It also passes the argument `"abc"` to the HSA program.

The HSA program listed above is produced for emulation purposes and cannot run on a real GPU. There are two main differences between native HSA programs and Multi2Sim-HSA-supported HSA programs. First, the simulated application does not have a host program to launch the kernel. Multi2Sim-HSA can directly run an HSA kernel, reducing the hassle of writing the host program. This allows users to test their HSAIL code faster and with less effort. Second, we add

simple predefined input/output functions to allow users to read and print data easily. With these I/O functions, users can check the output of the program as the emulation runs.

4.1.2 HSAIL-Hosted HSA Emulation

Multi2Sim can launch HSA kernels directly from an HSAIL function. This removes the necessity of writing host programs and reduces the programmer's work before running the kernel for the first time. Multi2Sim-HSA launches the main kernel at the beginning of the simulation. The main kernel serves as the host program. If necessary, the main kernel can start another parallel-executing kernel. Since the main kernel is written in HSAIL format, we name this emulation mode as HSAIL-Hosted HSA Emulation.

4.1.2.1 Program Entry

The program entry point of an emulated HSA program is always the main kernel, defined as follows:

```
kernel &main (kernarg_u32 %argc, kernarg_u64 %argv)
```

The main kernel has two arguments, `%argc` and `%argv`. They follow the standard C-style main-program interface. Specifically, the `%argc` argument is an unsigned 32-bit long integer that stores the number of the arguments. The `%argv` argument is the address that points to an array of argument strings. These kernel arguments can be omitted if a programmer decides not to use command-line arguments.

Multi2Sim-HSA starts the main kernel automatically when an simulation is launched. During simulation, Multi2Sim-HSA first creates an Architected Query Language (AQL) queue for the hosting CPU device. It automatically injects an AQL packet in the queue to launch the `&main` kernel. This kernel launch forms a grid on a CPU, with only a single work-item in the grid. The execution of the main function is single-threaded. By using runtime functions, users can easily launch other parallel kernels from the main kernel.

4.1.2.2 HSAIL Runtime

Multi2Sim-HSA defines a set of runtime functions that can be called by emulated HSAIL kernels. The main kernel and other kernels can call these runtime functions to control the emulated device, performing predefined actions such as memory allocation, AQL queue creation and kernel dispatching kernels.

Multi2Sim-HSA-defined HSAIL runtime functions have a one-to-one mapping to the official HSA runtime function. For example, if the official runtime function is:

```
uint64_t hsa_queue_add_write_index_relaxed
    (hsa_queue_t * queue, uint64_t value);
```

, the corresponding Multi2Sim-HSA HSAIL function will be:

```
function &hsa_queue_add_write_index_relaxed
    (arg_u64 %ret) (arg_u64 %queue, arg_u64 %value) {};
```

Specifically, the rules that map from the official HSA runtime functions to Multi2Sim-HSA HSAIL runtime functions are as follow:

- The names of the functions are identical, except for an extra `&` in front of the HSAIL function name. The HSAIL standard requires all function names to start with an `&`.
- The types of input and output arguments remain the same. The name of an argument can by any valid HSAIL variable name.
- Pointers are 64-bit integer addresses, regardless of the machine type (32-bit or 64-bit). HSAIL programmers have to perform a typecast if the HSAIL program uses 32-bit integers for addresses.
- Users do not need to implement the runtime functions. Multi2Sim-HSA ignores any user implementation to these functions and only performs the simulator-defined behavior.

When any predefined runtime function is invoked, Multi2Sim-HSA intercepts the function calls and converts them into application binary interface (ABI) calls to the HSA virtual device driver. The driver is part of the simulator. The driver performs predefined actions according to the information of the ABI call and returns results via memory. When the runtime function needs to call a callback function, the driver builds a stack frame for the callback function and hands over control to the emulation environment. After the callback function returns, the virtual driver takes over execution to finish the rest of the runtime function calls.

4.1.3 X86-Hosted HSA Emulation

Although it is very convenient to launch kernels from an HSAIL program, users may also want to simulate unmodified HSA programs with x86 host programs and HSAIL kernels.

Multi2Sim-HSA supports simulating the execution of host programs by the x86 component of Multi2Sim.

Multi2Sim-HSA ships with an HSA runtime implementation. A user needs first to compile the Multi2Sim-provided HSA runtime implementation and links the compiled runtime library with the application to emulate. The x86 emulator of Multi2Sim treats the Multi2Sim-provided HSA runtime library as part of the HSA application and emulates the instructions of the runtime library functions. The library relies on IOCTL system calls to communicate with the Multi2Sim-HSA virtual driver. Upon receiving the IOCTL system call from the runtime library, the driver performs specific actions, such as device capability queries, queue creation, kernel launching, and atomic signal operations.

4.1.4 HSA Instruction Emulator Design

4.1.4.1 Emulator Hierarchy

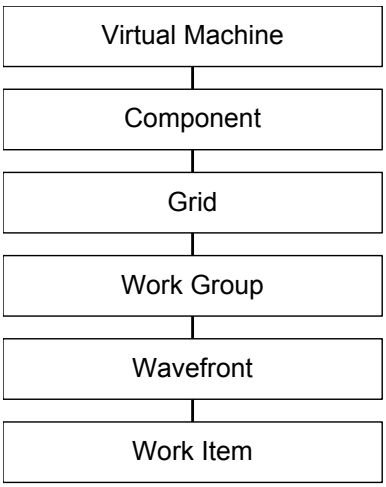


Figure 4.1: Hierarchical design of the Multi2Sim-HSA simulator.

The Multi2Sim-HSA HSA instruction emulator maintains a hierarchical structure, as shown in Figure 4.1. The emulator is composed of several HSA kernel agents (a device that supports the HSA specification and runs HSA kernels). When a kernel is dispatched, it forms a one-, two-, or three-dimensional grid. A grid can be divided into work-groups and can be further divided into wavefronts and work-items. A work-item is the lowest-level unit of computational work. All computation occurs in work-items. For simplicity of the emulator design, we do not emulate lock-stepped execution. Work-item execution is in a round-robin fashion. If a work-group has more

than one wavefront, the second wavefront will start executing after the first one completes. When executing a wavefront, Multi2Sim-HSA emulates one instruction from a work-item and moves to the next work-item.

Each work-item holds a standalone function call stack. The HSA specification requires registers to be associated with functions and function calls cannot use registers to pass argument values. Also, variables (i.e., memory pointers) can belong to different memory segments. Multi2Sim-HSA needs to use both the memory segment name and the address to locate the memory that holds the variable value. This unique design of the HSA specification requires Multi2Sim-HSA to opt for a nontraditional memory-based stack design, and a stack frame needs to store rich information. In Mult2Sim-HSA, each stack frame captures a snapshot of the current emulation state. The stack frame holds the current status of registers, arguments and variables.

While instantiating a work-item, Multi2Sim-HSA creates a base-stack frame for the entry function. Multi2Sim-HSA sets the program counter to the first entry in the code section of the BRIG file. According to the BRIG specification, the instructions and the directives are mixed in the code section, and so the program counter can be either pointing at a directive or an instruction. Multi2Sim-HSA treats directives (e.g., pragma and variable declaration) the same as instructions. Once the execution of an instruction or directive is finished, the program counter is updated to point to the next entry. By repeating this action, Multi2Sim-HSA can execute all the instructions and directives until all the work-items finish execution.

4.1.4.2 Memory Systems

Multi2Sim-HSA memory system design, as shown in Figure 4.2, follows the memory hierarchy defined in the HSA specification [44]. All the data is stored in a flat address space. Multi2Sim-HSA employs a memory manager to handle memory allocation and deallocations. The memory manager runs a best-fit memory allocation algorithm.

The HSA's segmented memory space requires an address translation between the intra-segment address and the flat address. We create a segment memory manager for each segment to handle operations involving non-global segments. Allocating memory on any memory segmentation is managed by the corresponding memory segment manager. For example, when a kernel is launched, the AQL packet explicitly requests the memory to hold the group segment and the private segment. Starting the execution of a work-item triggers memory allocation for the variables of the work-item with the help off the segment memory managers. Multi2Sim-HSA assigns each variable with an

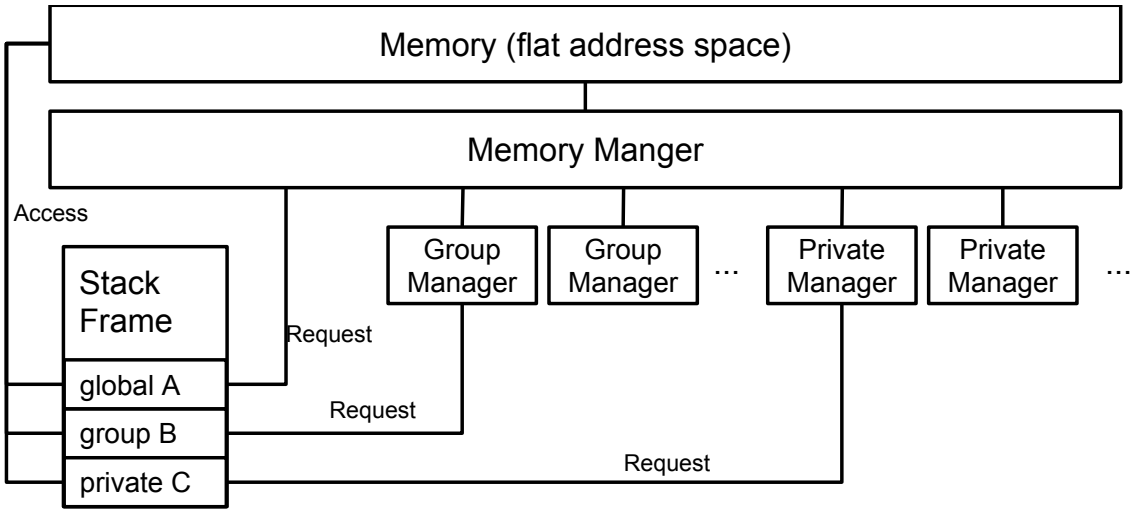


Figure 4.2: The Segment Management System.

address relative to the beginning of the belonging segment. When accessing those variables, the segment manager would first translate the inner-segment address to a flat address before issuing a load or a store to the memory directly.

4.1.4.3 Basic I/O Support

Since Multi2Sim-HSA supports running standalone HSA programs, providing functionalities for basic I/O can help users understand the HSA program execution. Simulators commonly handle I/O commands using system calls, as implemented in the SPIM MIPS simulator [75]. However, a system call interface is not presently available in the HSAIL specification. Therefore, Multi2Sim-HSA supports a customized set of library-like functions. The general format is as follows:

```
function &m2s_action_TypeLength
    (arg_TypeLength %input)
    (arg_TypeLength %output)
```

Here, the action can be either *print* or *read*. The type and length can be an integer, unsigned integer, bit string, or floating-point types are supported by HSAIL. The argument type and length must match the type and length in the function name.

4.1.4.4 Instruction-Level Execution Tracing

In addition to supporting HSAIL I/O functions, Multi2Sim-HSA provides a rich set of logging tools to capture the internal state of work-item execution. For example, by issuing:

```
m2s --hsa-debug-isa isa.debug hello_world.brig
```

a user can record detailed information about the execution of each instruction in the log file `isa.debug`.

The following log excerpt corresponds to the execution of one line of the our helloworld sample.

```
Executing: st_arg_u32 $s2, [%num];
***** Stack frame *****
  Function: &main,
  Program counter (offset in code section): 0xe4,
    call &m2s_print_u32 () (%num);
  ***** Registers *****
    $s0: 1, 1, 0.000000, 0x00000001
    $s1: 2, 2, 0.000000, 0x00000002
    $s2: 3, 3, 0.000000, 0x00000003
  ***** ********* *****
  ***** Function arguments *****
  ***** ******** ********* *****
  ***** Argument scope *****
    u32 %num(0x4) = 3 ( 0x00000003 )
  ***** ******** ***** *****
  ***** Variables *****
  ***** ********* *****
  ***** Backtrace *****
    #1 &main ()
  ***** ********* *****
***** ***** ***** *****
```

This piece of the log file records the work-item state right after executing the `st_arg_u32` instruction. The information captured includes the function associated with the current context, the program counter value, register values of interest, function parameters and their values, local variables and their values, and finally, a stack backtrace. When debugging an HSA program, a programmer can easily trace back to the instruction where the error first starts and fix the problem accordingly.

Besides ISA logging, Multi2Sim-HSA provides additional logging options. For example, *–hsa-debug-aql* can trace the AQL queue creation and removal, AQL packet reads and writes, and kernel launching. Option *–hsa-debug-driver* records details on how driver functions were invoked. The command `m2s -help` can dump all the possible logging options.

4.2 Hetero-Mark

Next, we introduce our work in characterising CPU-GPU collaborative execution. We first identify CPU-GPU Collaborative Computing patterns, then design Hetero-Mark, a benchmark suite that evaluates the CPU-GPU Collabortive Computing patterns. Finally, we evaluate the performance of CPU-GPU Collaborative Computing on an APU device.

4.2.1 CPU-GPU Collaborative Computing Patterns

We start our exploration of the CPU-GPU Collaborative Computing by identifying patterns in inter-device data flow of heterogeneous computing programs, analyzing the characteristics of each pattern, and their resulting impact on performance. The design patterns that are identified here are not meant to be mutually exclusive, nor to be fully comprehensive in the sense that they include all possible patterns. We identify patterns that can help us best utilize the available resources on both CPU and GPU. Note that a single program can include multiple design patterns.

4.2.1.1 Traditional Patterns

In traditional patterns, the CPU and the GPU work serially and not concurrently. Traditional patterns are important because most modern workloads are still written assuming these patterns. Also, not all the algorithms can be adapted to perform concurrent CPU-GPU execution due to limitations imposed by the algorithm, or constraints imposed by the targeted heterogeneous programming platform. In such cases, applications must be written assuming one of the traditional patterns. In traditional patterns, CPU-GPU communication features such as kernel launching, memory sharing, and synchronization may have a significant impact on the overall performance. Therefore, benchmarking traditional models is an important part of benchmarking CPU-GPU collaborative system performance. We consider three traditional patterns in this work: 1. *CPU to GPU*, 2. *CPU to GPU Iteration*, and 3. *CPU and GPU Iteration*.

CPU to GPU (C2G): Since the inception of general-purpose computing on GPUs, these devices have been used as accelerators, but act as a slave device for a CPU master. The CPU copies the input data to the GPU memory and the GPU only processes the data that is transferred to its memory. When the GPU kernel finishes execution, the CPU copies back the result from the GPU's memory to its own memory. When using this pattern, we consider the results retrieved from the GPU as the final result. The results can be used in other parts of the CPU program, but will not be

directly copied back to the GPU. Even though this is the simplest heterogeneous design patterns, the CPU-GPU pattern is still very powerful and frequently used in many high performance codes.

CPU to GPU Iteration (C2GI): As an extension of the C2G pattern, the GPU may need to use an iterative algorithm in order to produce the final result. Multiple iterations can be implemented within the kernel, or can rely on the host program to repeatedly invoke the kernel. The former solution is usually preferred since it avoids extra memory copies and kernel launch overhead. However, due to the lack of global synchronization capabilities in traditional GPU programming frameworks, data dependencies between iterations limit our ability to use only a single kernel launch. It may be necessary for the host program to repeatedly launch the GPU kernel, relying on the end of each GPU kernel execution to perform global synchronization.

When using the C2GI pattern, the data is initially copied from the CPU to the GPU. After kernel execution completes, the data does not need to be copied back, even if program control returns back to the CPU. Most modern heterogeneous computing frameworks are capable of supporting this feature.

CPU and GPU Iteration (C&GI): If we further extend the previous pattern and assume that the CPU needs to take part in the computation, the programming pattern becomes a *CPU and GPU iteration*. This is a common pattern found in applications that require the CPU to perform some computation before the GPU kernel can be relaunched. This pattern requires data to be copied back and forth between the CPU and the GPU device. Therefore, C&GI requires efficient memory management.

4.2.1.2 Collaborative Models

Forcing the CPU to wait for the GPU kernel to complete underutilizes the computing power of the CPU device on the targeted platform. Since a CPU is well-designed to process serialized workload, manage user input, and perform I/O to disk or a network, the application can use the CPU to perform such tasks during kernel execution. However, programming the CPU-GPU concurrent execution is still not a common practice, and can be a tedious process due to the limited communication and synchronization mechanisms provided by some current heterogeneous frameworks. Here, we summarize four design patterns that can help ease the development process, and that can be easily applied to many common algorithms.

Workload Partition (WP): Since the CPU can perform the same calculations as a GPU device (though at a different rate), engaging the CPU to do some portion of the work will reduce the

burden on the GPU and potentially reduce execution time.

Selecting the best *Workload Partition* requires an algorithm to be highly parallel and contain no dependencies between the data points across partition boundaries. Considering the nature of GPU programs, a large number of programs satisfy this requirement. Based on the Berkeley Dwarf taxonomy [48], both Dense Linear Algebra and the Structured Grid are suitable targets to use the *Workload Partition* pattern, since there is no dependency between data points in these two classes of workloads. If the overhead of memory copying and synchronization between the CPU and GPU can be kept relatively low, the N-body Pattern and Graph Traversal models can also be adapted to work with the *Workload Partition* approach.

When using *Workload Partition*, the programmer needs to balance the load between the CPU device and the GPU device. This pattern is particularly attractive to compute-intensive workloads. In memory-intensive workloads, the CPU may be too slow and eventually slow down the process. In this case, it may be more efficient to move all the workload to the GPU and leave the CPU to do some other tasks.

CPU producer GPU consumer (CPGC): In a large number of applications, the GPU serves as a service provider. For example, in video decoding applications, the GPU will run a video decoding kernel and the CPU will call that kernel for each frame during the processing of the video clip. In this scenario, the CPU produces the data while the GPU consumes the data and returns the desired result. In this pattern, the CPU needs to launch the GPU kernel repeatedly. If there exist cross-frame dependencies (most video decoding algorithms have some cross-frame dependence), flushing the data from registers and caches to GPU memory (or even CPU memory) will introduce significant overhead. Therefore, a proper hardware/software solution is needed.

One solution to support the CPU producer GPU consumer pattern is to use *persistent kernels*. Persistent kernels are defined as kernels that remain on the GPU and wait for the new input from the CPU [76]. The advantage of using persistent kernels is that kernel launch overhead and kernel synchronization can be avoided. If the data is preserved in the kernel registers and local data share (LDS) memory units, then data is immediately available rather than being refetched from memory.

Persistent kernels require the GPU device to support CPU-GPU bidirectional communication to allow the CPU to dispatch tasks, and to support kernel preemption. Kernel preemption allows graphics tasks and compute kernels to share the GPU for their execution. If those features are not available, CPU-GPU concurrent execution is still possible by asynchronously launching kernels from the command queue of the GPU. However, we lose the benefits of storing data close to the compute

units of the GPU.

GPU producer CPU consumer (GPCC): Alternatively to CPGC, the GPU can produce data during kernel execution and feed the results to the CPU. If the CPU program only requires some of the data in order to proceed, part of the data from the GPU can be transferred immediately to the CPU when it becomes available, and then the CPU can start to process the partial data. For example, a video encoding kernel can pass each frame to the CPU one-by-one during the video encoding process. Whenever a frame is processed, the encoded frame can be written to disk or offloaded to a network by the CPU.

This pattern can be implemented using different approaches. If CPU-GPU bidirectional communication is available on the platform, a CPU thread can wait for the ready signal from the GPU. When the data is ready, the CPU can start processing the data. If global atomic operations are available, the signal can also be sent via global memory, and the CPU can keep watching for further changes in the signal memory. An alternative approach is to use cross-device task enqueuing. The GPU device can launch a serial task on the CPU. If neither of these mechanisms is available, we can fall back to using the traditional CPU-GPU pattern where the GPU stores the data in an array and CPU processes the elements in that array.

CPU-GPU Pipeline (CGP): The most generic pattern for CPU-GPU collaborative programming is the CPU-GPU pipeline pattern. In this pattern, an algorithm is divided into stages. The CPU can work on some stages while the GPU works on the other stages. The data also needs to be divided into blocks to be fed to the CPU-GPU pipeline. A Convolutional Neural Network (CNN) [9] is a very good example of a classic CPU-GPU pipeline pattern. In a CNN, the data traverses through many different neural network layers and finally reaches the final stage. If the CNN has many layers, the GPU can process some of the layers, while the CPU processes a few fully-connected layers.

The advantage of using this pattern is that we do not need any advanced features on our target platform in order to support execution. Any traditional CPU-GPU heterogeneous computing framework should be able to support this feature as long as it supports asynchronous kernel execution. The main drawback of this pattern is that the program flow may need to be changed to better match this pattern, which can result in redundant data copies between the CPU and the GPU.

4.2.2 Benchmark Suite Design

Hetero-Mark is designed to explore different memory patterns between the CPU and GPU devices. We provide at least one workload for each of the seven patterns categorized in Section

4.2.1. Our workload selection follows the model suggested in the Berkley Dwarf Patterns [48]. The benchmark applications are listed in Table 4.1, alongside their Dwarf taxonomy, the CPU-GPU collaboration pattern, and target application domain. The KMeans benchmark in Hetero-Mark is a modified version of the KMeans program included in the Rodinia benchmark suite. The remainder of the benchmarks are written from the scratch.

All of our benchmarks are implemented as if they are an accelerated library that works as a component in a larger program. Unlike the typical implementation of a benchmark where the buffer type of the heterogeneous computing framework is used throughout the program, Hetero-Mark applications accept input from ordinary pointers and variables and output the results using ordinary pointers and variables. This allows the user to benchmark the most realistic use cases of a GPU device. We take into account that many applications consist of a main function which is written in a standard object-oriented programming language such as C++ (this code does not use any heterogeneous computing features), and only the performance-critical part of the application is equipped to use GPU acceleration. Additionally, we made sure to include applications in Hetero-Mark which, similar to traditional GPU applications, transfer a data buffer from the CPU to the GPU, as input to the GPU kernel.

4.2.3 Workloads

Table 4.1: Hetero-Mark Workloads

Abbv.	Workload	DWARF	Pattern	Domain
AES	Advanced Encryption Standard	Dense Linear Algebra	C2G	Cryptography
BE	Background Extraction	Structured Grid	CPGC	Image Processing
BS	Black-Scholes	Dense Linear Algebra	WP	Finance
CH	Color Histogram	Map Reduce	C2G	Image Processing
EP	Evlotionary Programming	Map Reduce	CGP	Machine Learning
FIR	Finite-Impluse Filter	Dense Linear Algebra	C2G	Signal Processing
GA	Gene Alignment	Dynamic Programming	GPCC	Bioinformatics
KM	KMeans	Dense Linear Algebra	C&GI	Data Mining
PR	PageRank	Sparse Linear Algebra	C2GI	Data Mining

Advanced Encryption Standard (AES) The AES-256 algorithm [77] uses a 256-bit key for both encrypting plaintext into ciphertext, and decrypting the ciphertext back to plaintext. A series

of operations such as *byte-substitution*, *row-shifting*, *column-mixing*, and *bitwise xor* are applied to each 128 bits of the plaintext. In our implementation, the entire plaintext data is transferred to the GPU. The GPU then scans the plaintext and encrypts the data without exiting the kernel. Finally, the GPU writes the encrypted ciphertext to an output buffer. Since AES follows a traditional CPU to GPU pattern, we use it to benchmark the impact of memory management in a compute-intensive application.

Background Extraction (BE) Background extraction is a very useful algorithm in video and image processing. In a surveillance video, the background information is not very useful, and only the foreground contains the information of interest. Background extraction algorithms usually create a background model based on static components of the frame using a Running Gaussian Average [78] or a Mixture of Gaussians (MoG) [79] method. We use the former in our implementation.

The Background Extraction follows a standard CPU Producer GPU Consumer (CPGC) pattern. The CPU decodes a video frame and feeds this frame to the GPU kernel. The GPU kernel uses the frame to update its background model and then subtracts the background from the frame. Then the GPU sends the resulting image, which only contains the foreground, back to the CPU. The background extraction is performed frame-by-frame, so if a persistent kernel is used, the kernel launch and synchronization overhead will be much lower. However, due to limitations imposed by the current HC C++ API on the programmer, we cannot directly access the HSA signals. This makes it difficult to leverage the benefits of a persistent kernel. Therefore, we use asynchronous kernel launches for this benchmark.

Black-Scholes (BS) Black-Scholes is a mathematical model which provides a partial differential equation to evaluate the price of the European-style financial market *options*. The BS benchmark calculates the *samples* of call and put prices of an option based on a given sample of stock price, strike price, remaining time to expiration, and volatility. The BS is massively parallel (options are completely independent from each other). Therefore, we use both CPU and GPU devices in order to compute the call and put price for the options. This application is categorized under the WP pattern.

HC C++ AMP features a *completion future* object which allows concurrent execution of CPU and GPU code. The application launches a large portion of the array of options on the GPU, but immediately return to the CPU code. The CPU continues running multiple instances of BS on other smaller portions of the array until the GPU finishes its execution (marking the completion feature object as ready). Then another large portion of the array is calculated on the GPU, and this routine is repeated until all portions are calculated for the whole array.

Color Histogramming (CH) Color Histogramming is a popular method in image processing to divide the color space into groups, and counts the number of pixels in a picture that fall into each group. Our benchmark adopts a widely used parallel implementation of the CH application, also used in statistical analysis of any type of data [80].

The implementation of Color Histogramming is divided into two phases. In the first phase the GPU kernel scans the whole image and each GPU thread covers a small portion of the image. Each thread stores the histogram information of the pixels it has scanned in a region of the private memory that is dedicated to that thread. In the second phase, each GPU thread takes the histogram produced in the first phase and adds it to an output histogram using atomic operations. The `atomic_add` instruction used in the second phase is a global atomic operation, and the result is visible to both CPU and GPU. We include Color Histogramming to evaluate the performance of global atomic operations, as one of the key CPU-GPU collaborative features.

Evolutionary Programming (EP) Evolutionary Programming solves optimization problems using an approach that mimics the natural selection process. EP is good at providing solutions for problems that lack a straight-forward mathematical formulation, as well as many non-convex global optimization problems. The optimization process is divided into five consecutive stages, *reproduction*, *evaluation*, *selection*, *crossover*, and *mutation*. A large population (a group of entities) is generated in the reproduction stage, and a *fitness score* is assigned to each entity in the evaluation stage, using a fitness function. The least fit entities are eliminated during the selection stage, leaving the best fit entities to pass their *good* properties to the next generation of the entities in the next crossover stage. The mutation stage adds some randomness to some entities to help them jump out of possible local optimum regions. The stages execute recursively until arriving at the optimal result or satisfying selected conditions.

In our benchmark implementation, we use Evolutionary Programming to solve a non-convex optimization problem. Since the evaluation of the population and the mutation on the population have no dependency on individual entities, this computation maps nicely to a GPU. To utilize CPU and GPU resources at the same time, we utilize a CPU-GPU Pipeline (CGP) pattern. We divide the whole population into islands, where an island population evolves by itself, but some entities occasionally migrate from one island to another. When the CPU is done with one stage, we feed its results to the GPU and let the CPU process the population from another island.

Finite-Impulse Response (FIR) Filter The FIR filter is a signal processing algorithm that produces a finite response if the input signal is finite. The filter applies a window to the signal data using convolution. The GPU kernel reads in adjacent data points, multiplies the data points within

the filter window in an element-wise fashion, sums the result, and writes the result back to global memory. FIR is considered a highly parallel, memory-intensive application. The application has a large memory footprint, so the copying of data between the CPU and the GPU can consume a large portion of the execution time. We include FIR in Hetero-Mark to test how well a platform can handle the memory transfer between the CPU and the GPU, and how well a GPU device can process a memory-intensive workload.

Gene Alignment (GA) Gene Alignment algorithms are used to answer questions about specific gene sequences (e.g., "CATGCATG") that occur in the human gene sequence. It is an essential bioinformatics application studying how genes change from generation to generation, or when identifying how a gene sequence links to a certain disease. Usually, the query sequence (e.g., the "CATGCATG") is much smaller than the target sequence (e.g., the human gene sequence). The target sequence can scale up to tens of billions of elements. A simple substring search algorithm cannot solve the problem since the gene alignment needs to consider gaps. For example, a string "CATGTG" fits the sequence "CAATGATG" very well by opening up two gaps in the first sequence, making it "C-ATG-TG".

Our implementation uses a modified version of the Basic Local Alignment Search Tool (BLAST) [81]. The algorithm is divided into two phases. In the first phase, a *coarse-grained search* is performed on the GPU, and the whole target sequence is scanned for the substrings in the query sequence. To speed up the coarse-grained search process, we do not perform gaped matching. The results of the coarse-grained search are called *seeds*, and are used as input for the second phase, *fine-grained match*. The fine-grained match uses the Smith-Waterman [82] algorithm to perform *gapped matches* only on the seed points. We utilize the CPU to perform the fine-grained match since the fine-grained match is highly serialized as compared to the coarse-grained search, and can be executed in parallel with a GPU kernel.

Gene alignment is a workload that follows the GPU Producer CPU Consumer (GPCC) pattern. The kernel executes for a relatively long time and generates some seeds during kernel execution. These seeds are independent from each other and need to be processed by the CPU. In order to start the fine-grained match process on the CPU as soon as a seed is found, we allocate a shared buffer for the CPU and GPU to communicate. Whenever a seed is found, the GPU writes into the shared buffer. The CPU watches the buffer and starts a thread to perform the fine-grained match upon receiving a signal from any thread on the GPU.

KMeans (KM) KMeans is a popular clustering algorithm, frequently used in unsupervised machine learning and data mining applications. The KMeans algorithm locates k *centroids* to

minimize the sum of the distances from the data points to the nearest cluster centers. During each iteration, the GPU updates the tag of each data point, indicating which centroid is the closest. The CPU uses the result of the GPU calculation to update the position of the centroids. When a centroid location does not change between iterations, the optimal clustering has been achieved.

We include KMeans in Hetero-Mark since its workload follows the CPU-GPU iteration pattern. The tag data and the centroid data need to be copied back and forth between the CPU and the GPU. The KMeans benchmark can be used to evaluate the efficiency of memory management of a heterogeneous device and the ability of the device driver to avoid redundant data copies.

PageRank (PR) The PageRank algorithm was first used by Google in their search engine to evaluate the importance of web pages. For each page, the PageRank computes the number of outgoing links for each page and the quality of each link, in order to estimate its importance. The algorithm runs recursively until a convergence point is reached.

In our implementation, we read the link graph in the form of a sparse matrix. Two vectors, A and B, are used to store the importance values in pages. The first iteration uses A as the input and uses B as the output, and the second iteration uses the opposite order. Since the CPU does not need the data before completion of the whole process, the data can remain on the kernel side. We include the PageRank workload because it represents the CPU to GPU Iteration (C2GI) pattern. We can use the workload to evaluate how well a platform handles this pattern, as well as the memory copyies between the CPU and GPU.

4.2.4 Evaluation Methodology

Execution platform: We used an AMD A10-7850K APU as the primary device to demonstrate the capabilities of Hetero-Mark. Based on Kaveri APU model, the AMD A10-7850K is the first processor that supports the HSA 1.0 specification. The HSA software stack is configured using the Radeon Open Compute Platform (ROCm) [22] 1.1, while the OpenCL software stack consists of AMD APP SDK 3.0 and Fglrx driver (version 1912.5). All evaluations are performed on 64-bit Ubuntu 14.04.4.

Benchmark Setup: We provide an OpenCL 1.2 implementation of the applications that are categorized under the traditional patterns. These implementations use OpenCL API calls such as `clEnqueueReadBuffer`, `clEnqueueCopyBuffer`, etc., to manually manage memory transfers between the CPU and the GPU. The HSA version of applications with traditional patterns use a Unified Memory Space, so instead of using buffers, native C pointers are used as arguments to

Table 4.2: Hetero-Mark Workloads Input

Abbv.	Workload	Input
AES	Advanced Encryption Standard	A plaintext of 1048576 bytes
BE	Background Extraction	One second (24 frames) 1080p (1920 × 1080 pixels) B&W video
BS	Black-Scholes	65536 stock price samples
CH	Color Histogram	A 3000 × 2000 pixels B&W picture
EP	Evoluationary Programming	3 generations, each generation has 10000 population
FIR	Finite-Impulse Filter	100 groups, 32768 samples per group
GA	Gene Alignment	Search an 1024 element sequence against an 1M element sequence
KM	KMeans	10000 34 dimension data points
PR	PageRank	4096 Nodes

the kernel.

Benchmarks with collaborative patterns are implemented in HC C++ API. We implement two versions of the same application: overlapped execution and non-overlapped execution. The overlapped version of the benchmarks with collaborative patterns benefit from both the CPU and the GPU in parallel for computation. The non-overlapped implementation of the workloads serializes the CPU and GPU operations, but still leverages both devices in the computation. The only exception is in the *BlackScholes* application which does not require the CPU to carry out computations in the non-overlapped version (it uses the CPU in overlapped version only to boost its performance). The workload setup for each benchmark is shown in Table 4.2. Input parameters are chosen to capture typical use cases of each workload. However, users of Hetero-Mark are able to change the input size of each benchmark easily.

4.2.5 Evaluation

4.2.5.1 Traditional Pattern Benchmarks

Figure 4.3 presents the normalized execution time of the Hetero-Mark applications that are implemented using the traditional patterns. Execution time is divided into two parts: 1) *k*ernel time

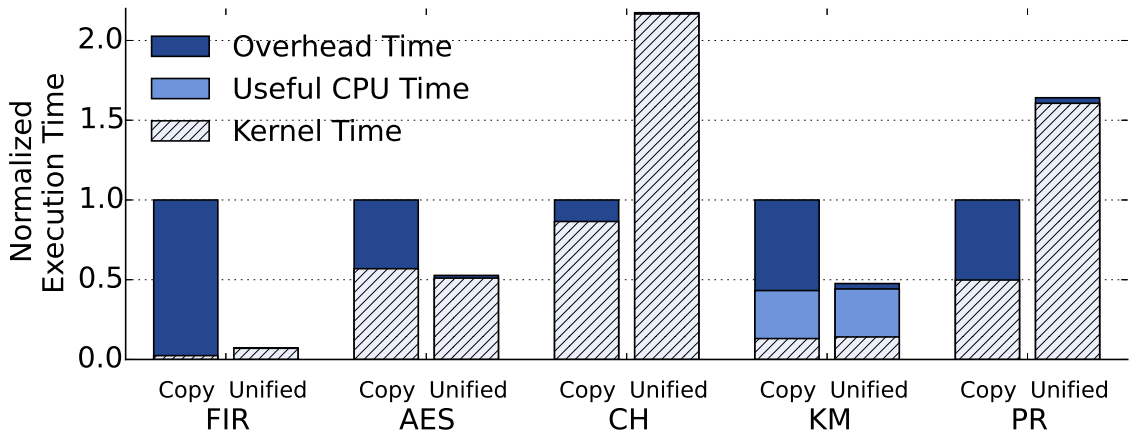


Figure 4.3: Execution time breakdown for workloads that use traditional patterns. "Copy" and "Unified" represent the *Memory Copy* approach in OpenCL 1.2, and the *Unified Memory* approach in HSA, respectively.

and 2) *n*on-kernel time. The kernel time only includes the time spent on the GPU for computation. Any time that is not spent by the GPU on the kernel execution is categorized as non-kernel time. This includes the time it takes to launch a kernel, the time spent copying data back and forth between the CPU and the GPU, and the time spent on the CPU (the host program). For the *KMeans* application, the non-kernel time also includes the CPU calculation time. Since there is no overlapped execution on the CPU and the GPU, the total execution time of a program is equal to the sum of the kernel time and the non-kernel time.

An application that is implemented using HSA benefits from using unified memory (CPU and GPU share the location of the data via a pointer), while the OpenCL 1.2 implementations have data copied between the CPU and the GPU. As shown in the Figure 4.3, the non-kernel time in the unified memory approach is more than 2$\times$ shorter than the non-kernel time for the memory copy implementation for all of the benchmarks that use a traditional pattern. FIR enjoys the most benefit from unified memory, reducing the CPU execution time from 0.19 second to 0.6 ms; This is because the application spends 97.64% of its execution time copying data back and forth between the CPU and the GPU. The percentage of overall execution time due to data copy dropped to 4.17% when unified memory is used.

However, we observed some drawbacks with unified memory solution in terms of increased kernel execution time, mainly caused by compiler and intermediate language inefficiencies due to increased programming flexibility (HSA is in its infancy, and we expect this overhead to be minimized

in future releases). The applications that are most impacted in terms of increased kernel time are CH and PR. Analyzing the HSAIL code of CH shows that all the atomic operations (which is used to increase programming flexibility) are compiled to cross-device atomic operations (intermediate language inefficiency), resulting in a large number of redundant communications between the CPU and the GPU. For benchmarks such as AES and KMeans, the HSA framework has a more positive impact on kernel execution performance, providing a speedup of 1.12× and 0.93× respectively. But we see a slowdown of approximately 0.65% in the kernel time of FIR, KMeans, and PageRank. The performance impact in kernel execution is compensated by the significantly lower runtime overhead for FIR and Kmeans, resulting in a 13.7× and 2.1× overall speedup, respectively, but leads to a 1.64× slowdown for PageRank. The observed slowdown indicates that Hetero-Mark is capable of finding out potential design defects in heterogeneous systems.

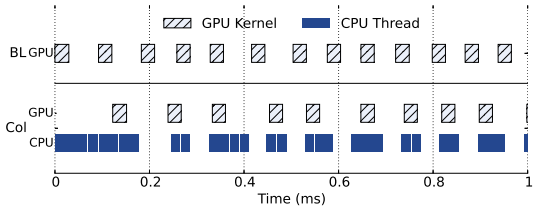
4.2.5.2 CPU-GPU Collaborative Execution

Figure 4.4 shows a detailed breakdown of execution time of all four benchmarks that use CPU-GPU collaborative execution patterns. We have observed speedups in all four benchmarks as compared to their baseline implementations.

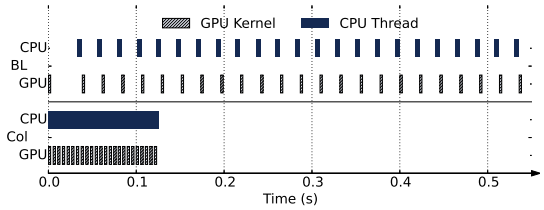
The *Block-Scholes* (Figure 4.4a) application applies the Workload Partition pattern to better utilized CPU computing resources. After launching a GPU kernel, the CPU processes a small block of data and checks if the GPU kernel has completed. If not, the CPU processes the next small block, otherwise, the CPU launches the next GPU kernel. In our experiment, 21% of the workload is processed by the CPU, achieving a 1.31× speedup for BS.

The *Background Extraction* (Figure 4.4b) application uses the CPU Producer and GPU Consumer pattern to allow the GPU kernel to provide service to the CPU. In our baseline implementation, each kernel is launched when the CPU knows that the previous kernel has finished execution. Due to long synchronization and memory copy times, the GPU is highly underutilized. On the contrary, when the kernel is launched asynchronously, the GPU kernel launching overhead becomes much smaller (the synchronization only happens at the beginning and the end of the application) and the GPU is fully utilized during the execution period, resulting in a 2.78× speedup.

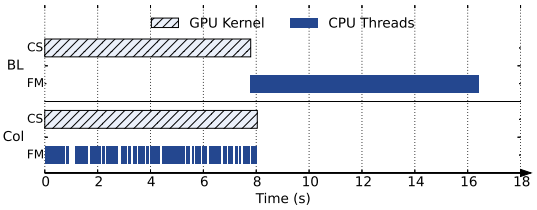
The *Gene Alignment* (Figure 4.4c) workload follows the GPU Producer CPU Consumer pattern, allowing the CPU to start to process the gapped gene sequence match while the GPU kernel is continues forward, looking for the next point of interest. As can be seen in the figure, collaborative concurrent CPU and GPU execution provides nearly a 2× speedup for the GA application.



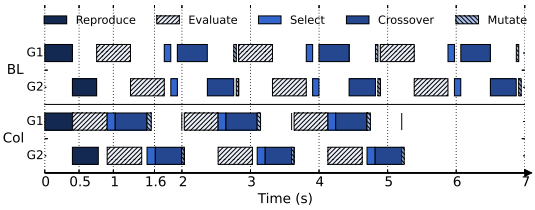
(a) Blacksholes. Collaborative model finishes execution at 1.15 ms, comparing to 1.52 ms of the baseline model.



(b) Background Extraction



(c) Gene alignment. "CS" and "FM" stands for *Coarse-grained Search* and *Fine-grained Match*. Each CPU block in this figure maps to multiple threads execution.



(d) Evolutionary Programming. Shaded regions are GPU stages. "G1" and "G2" resprents the processing of data group 1 and data group 2.

Figure 4.4: Execution Time-line of Collaborative Executing Workloads. "BL" and "Col" are short for *Baseline Implementation* (CPU GPU execution is not overlapped) and *CPU-GPU Collaborative Execution Implementation*.

The *Evolutionary Programming* (Figure 4.4d) workload is implemented as staged execution, so we use the CPU-GPU Pipelined collaborative pattern. As stated earlier we have an overlapped and a non-overlapped implementations of this application. In the non-overlapped implementation, one computing device (CPU or GPU) can be active and can only work on only one group. For instance, as shown in the figure, at time $0.5s$ the CPU is only allowed to work on group 2 (G2), and GPU is not allowed to work on group 1 (G1), even though the groups are independent from each other, and the GPU device is idle. GPU has to postpone the computation of the G1 (to $0.8s$) until CPU finishes the execution of the G2. On the other hand, in the overlapped version, the two devices can compute different stages of different groups at the same time.

Figure 4.4d also reveals a classic limitation of pipelines. Similar to any pipelined operation, the duration of all the stages of the pipeline are equal to the time spent in the longest stage of the pipeline. For instance, at $1.6s$ we observe that the pipeline in not fully utilize the resources, since the

Crossover stage (a CPU stage) takes much larger time to complete in comparison to the Mutation stage (a GPU stage).

4.2.6 Reliability Analysis

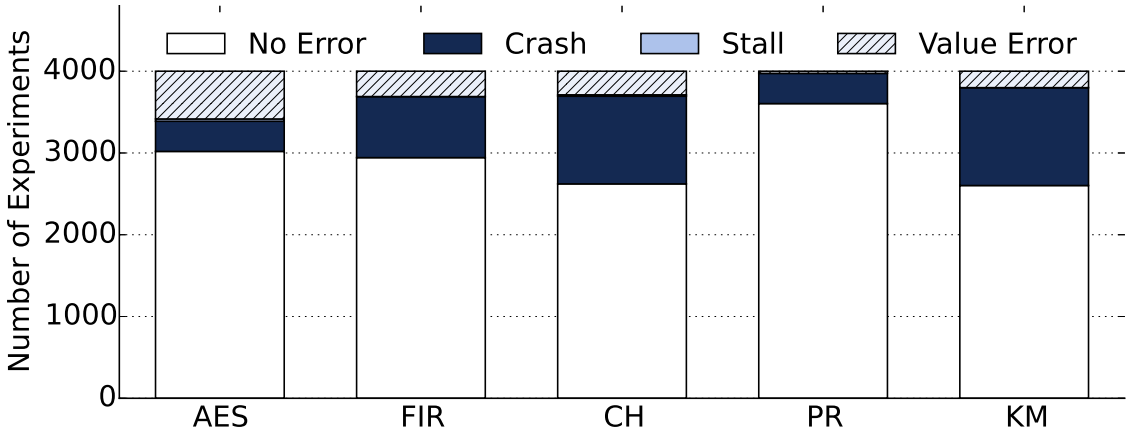


Figure 4.5: Number of Errors of Each Workload When Injected With 1 Bit Flip in the GPU Register File.

In terms of reliability of collaborative CPU-GPU environment, the GPU is considered to be more vulnerable to faults than the CPU due to its large register file and the lack of ECC protection [83]. For CPU and GPU collaborative computing applications, faults can no longer be discounted on the GPU side, since they may propagate to the CPU side through unified memory. Therefore, it is attractive to use a benchmark suite such as Hetero-Mark to test reliability of CPU-GPU collaborative execution platforms. We extended the Multi2Sim [23] simulator to provide an HSAIL level simulator, in order to analyze the fault tolerance properties of Hetero-Mark. We utilize the Multi2Sim X86 model to run the host program and redirect the HSA API calls to the simulator's virtual driver. Our simulator fully supports a unified memory space where the same pointers can be used both on GPU and the CPU. By simulating both the CPU and the GPU execution, we can evaluate the flow of information via the share memory, and assess how errors can be passed between the GPU to the CPU.

In our study, we performed 4000 bit flip fault injection experiments on each benchmark. In each experiment, one bit flip is injected into a random register in GPU register file at a random time. All the injections are performed on live registers (used at least once in the program). The registers considered here include the general purpose registers (GPRs), control registers (1-bit registers defined

in the HSA standard for conditional branches), and the program counter (PC) register. Since the simulator only supports a native HSA runtime environment, we only present the results of applications that implement traditional patterns.

Figure 4.5 illustrates the diversity of the Hetero-Mark workloads under fault injection. Although all injected errors modify live registers, on average 74% of the injections do not cause any error in the benchmarks. This is expected since the fault may have been injected into registers that do not impact program's visible outputs. We observe that application such as *KMeans* and *Color Histogramming* are more vulnerable to fault injections. This leads to total 34% and 35% error rate in CH and KM for 4000 single bit-flip fault injection, respectively.

4.3 Improving CPU-GPU Communication with Priority-Based PCIe Scheduling

4.3.1 Introduction

Recently, as GPUs have quickly become a standard computing platform present in datacenter systems, cloud computing vendors are starting to deploy multi-GPU systems in the cloud and deliver GPUs "as a service" [84, 85]. In cloud-based multi-GPU systems, a typical configuration is to allocate each user (tenant) a set number of dedicated GPUs, while virtualizing the CPU and the PCIe connection, so that they can be shared by multiple users [86, 87, 88]. Typical GPU workloads follow the *"copy-then-execute"* model. The execution on the GPU side cannot start until data is fully copied from the CPU to the GPU. As shown in Figure 4.6, the PCIe bandwidth between a CPU and a GPU remains the same when from moving a single GPU system to a multi-GPU architecture, leading to serious bandwidth concerns when multiple GPUs are communicating with the CPU, delaying the start of GPU execution.

By default, the communication traffic from the CPU to the attached GPUs is scheduled in a *Round-Robin* (RR) manner. RR scheduling attempts to guarantee fairness among all GPUs, but introduces delay of key memory packets, impacting the throughput of the corresponding GPU execution. As observed on production multi-GPU systems, this bandwidth competition causes severe performance degradation, especially for memory-bound workloads. Moreover, multi-tenant system users cannot manage their own data movement as each user is treated agnostically. Therefore, it is necessary to *schedule* the PCIe traffic associated with different GPUs to improve the overall system throughput.

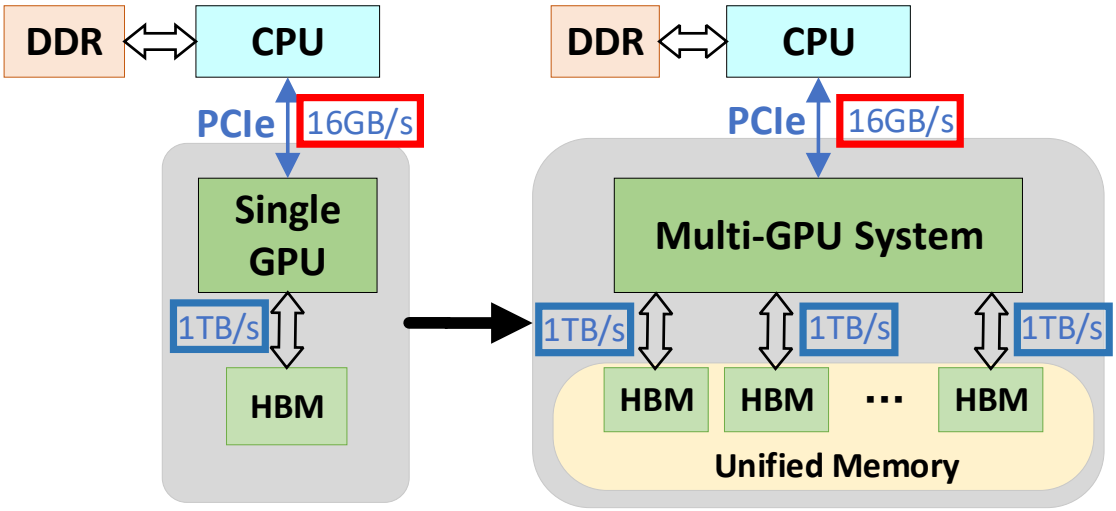


Figure 4.6: PCIe Bandwidth Bottleneck in CPU-GPU Heterogeneous Computing Systems.

In this section, we exploit a priority-based PCIe scheduling policy and describe semi-QoS application management for CPU-GPU communications to improve overall system computing throughput. Memory transfer commands with smaller data sizes are prioritized at runtime to achieve higher throughput. If a task has a specified QoS goal, but we predict that the goal will not be met, the task's priority level is escalated to meet the requirement. Experimental results show that system throughput is improved by 7.6% on average with priority-based PCIe scheduling as compared with the Round-Robin-based PCIe scheduling. The semi-QoS management can also meet defined QoS goals, achieving a 5.3% performance improvement as compared with RR PCIe scheduling.

4.3.2 Motivation

Current commercial cloud-based CPU-GPU systems consist of 1-2 CPUs and 4-8 GPUs interconnected with a PCIe or NVLink fabric. Traditionally, 4 GPUs share a host CPU and a PCIe bus. Every two GPUs are connected to a PCIe switch, which also connects to the root complex of the PCIe bus, as shown in Figure 4.7. Constrained by the inter-GPU or CPU-GPU communication bandwidth, a multi-GPU system cannot scale performance linearly as with an increasing number of GPUs. The low bandwidth and high latency associated with the current inter-GPU/CPU-GPU fabrics can be a bottleneck for the system performance.

Figure 4.8 shows the performance degradation due to the PCIe bandwidth contention between CPUs and GPUs. We run two deep neural network models *ResNet50* [89] and *DeepIn-*

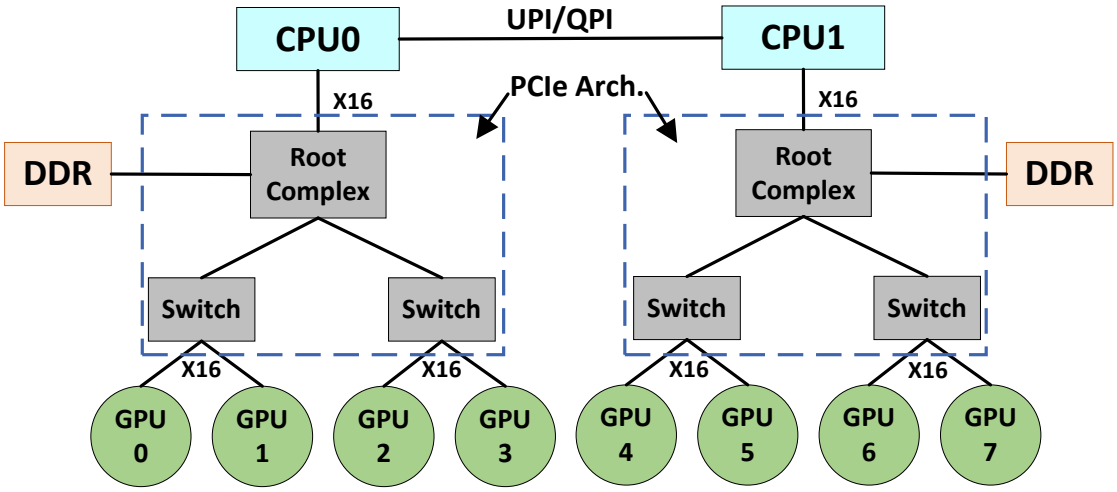


Figure 4.7: Multi-GPU Topology

terest [89] and multiple instances of each (tasks), totaling eight tasks in all possible combinations. Each task executes either the training of *ResNet50* or *DeepInterest* on one GPU. Different tasks are bound to different CPU cores to avoid interference. We have two key observations from Figure 4.8. First, since tasks are independent of each other, and we intentionally allocate them on different CPU cores and GPUs, so most of the slowdown will be due to contention of CPU-GPU PCIe bandwidth, resulting in performance degradation of both *ResNet50* and *DeepInterest* training, as each execution needs to wait for the data transfer. Second, workloads have different sensitivities to PCIe contention. *DeepInterest* is more sensitive, while *ResNet50* is not. The slowdown of *ResNet50* is around 5% over an isolated execution, even when seven instances are executing concurrently, while *DeepInterest* sees more than a 20% degradation with one instance competing with seven *ResNet50* instances. On average, we observe a 18.1% slowdown for *DeepInterest*. We conclude that the contention on the PCIe connection significantly degrades the performance of multi-tenant multi-GPU systems, especially for bandwidth-sensitive workloads. Delivering an improved PCIe scheduling policy is mandated to reduce such contention.

Existing PCIe connections in CPU-multi-GPU systems adopt a RR scheduling policy. For each data transfer request, the DMA engine breaks down data transfers into smaller packets and buffers them in the network interface. The PCIe arbiter selects a packet from a different application to transfer during each time slot. Assuming two applications have a similar amount of data to transfer to their GPU, an RR-based PCIe connection would complete the data transfer in approximately the same time, which is about twice the time of a single data transfer without contention. Consequently,

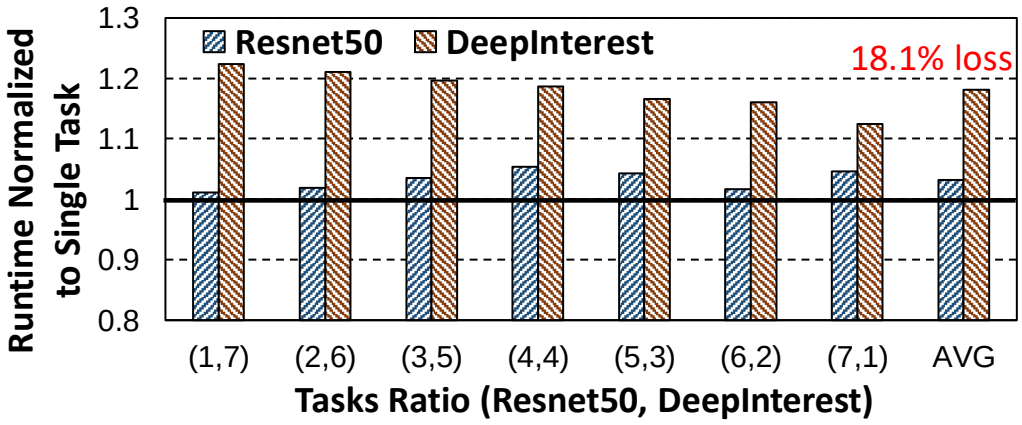


Figure 4.8: Slowdown due to PCIe bandwidth contention. (Collected from a multi-GPU system with 8 NVIDIA Volta GPUs.)

the GPUs start executing kernels at approximately the same time, leaving their computing pipelines idle during the data transfer. Instead, if we only transfer the data from one application first, the GPU can start executing the kernel for this application earlier. As the GPU is executing the application, the PCIe connection can transfer the data for the other applications, overlapping transfers and execution. Ideally, this approach can effectively hide the data-transfer latency across multiple GPUs and improve resource utilization over the execution of multiple iterations.

4.3.3 Design

4.3.3.1 Baseline Round-Robin Scheduling Design

The baseline CPU-GPU system in this paper consists of four GPUs connected to one host CPU, which is among the most widely adopted architecture in commercial CPU-multi-GPU servers. The data transfer between CPU and GPU is managed by the DMA engine. A memory copy command from CPU to GPU can be divided into multiple data packets. Each packet on the PCIe link is labeled with the destination GPU (ID_D) and the packet type (ID_T). To transfer data from the CPU to GPUs via the DMA engine and PCIe switches, packets are first mapped to a particular virtual channel (VC) according to its traffic class (TC) as shown in Figure 4.9. TC can be calculated according to

$$TC = N_T ID_D + ID_T, \tag{4.1}$$

where N_T is the total number of packet types. TCs are directly mapped to different VCs by

$$VC = TC \bmod N_{VC} \tag{4.2}$$

The RR policy is set as the VC arbitration policy in the PCIe switch. All VCs have the same priority so that packets from different VCs are fetched and handled one-by-one in the RR order. Although the TC could be defined by users, we assign the TC for each memory transfer command using the driver, in order to support transparent multi-tenant management.

In this case, if there are packets in VCs that are queued, the arbiter forwards one following the VC arbitration policy to the routing logic in each cycle. The routing logic then directs packets to their destination GPUs. Since packets being sent to different GPUs are assigned different VCs, each VC is scheduled in RR. Fairness across multiple GPU traffic streams can be achieved. However, naive scheduling will lead to contention on the PCIe interconnect, resulting in long GPU stalls.

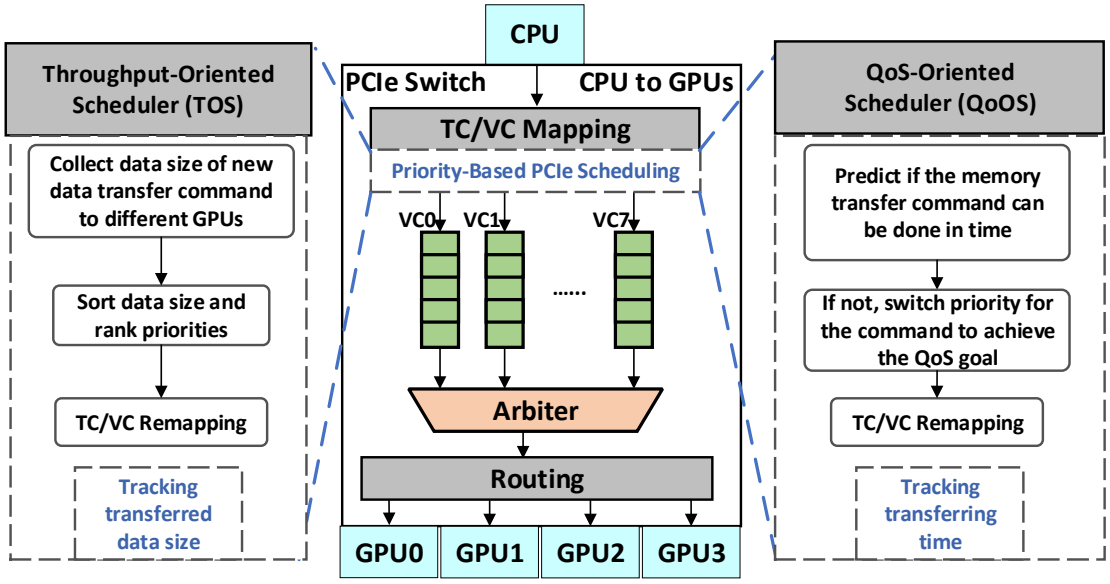


Figure 4.9: High-level Overview of a PCIe Switch (The Process of a Packet Sending from the CPU Side to GPU Side).

4.3.3.2 Priority-Based Scheduling for Throughput

To mitigate bandwidth contention, we propose a dynamic priority-based PCIe scheduling policy. The key idea is to hide the memory transfer latency by overlapping it with kernel execution of different GPUs. To enable this capability, we increase the priority of some memory transfers. These memory transfers can be completed without interruption, allowing the associated GPU kernel can start executing as soon as the transfer is complete. Low-priority data transfers can be performed while higher-priority tasks are already in execution. As data-center workloads often exhibit repeated

"copy-then-execute" patterns, our policy can effectively reduce data transfer interference among different tenants.

We develop a Throughput-Oriented Scheduler (TOS) in the driver, as shown on the left side of Figure 4.9. TOS uses *Strict Priorities* for VC arbitration, where *VC7* has the highest priority and *VC0* has the lowest priority. The arbiter always handles requests from highest priority VCs first.

We classify packets into *two* types ($N_T = 2$): 1) request packets ($ID_T = 0$, read packets and write packets) and 2) response packets ($ID_T = 1$, read response packets). TOS always gives response packets higher priority. By granting responses higher priority, data transfer transactions can be completed sooner, enabling programs to start running on the GPU sooner.

Priorities are assigned to the packets of different tasks by using TC/VC remapping. Priorities depend on the size of the remaining data transfer associated with each memory transfer request issued by different tasks. Inspired by the idea of "small flow first" in network, tasks with fewer data to be transferred are granted higher priority, which should reduce GPU stall time and achieve higher execution throughput. A remaining data-transfer size list is managed in the TOS to keep track of the remaining data for each task. When a new memory transfer command is received, the TOS ranks the priorities by sorting the remaining data sizes for each task and updating the TC/VC remapping. For tasks with lower priorities, it is possible that starvation may happen in some extreme cases. Thus, we set a threshold x empirically for each VC. If the head request in the VC is blocked for x cycles, it will be handled immediately to avoid starvation.

Compared with RR, TOS reduces the bandwidth contention for all users under multi-tenancy. However, if a task with a strict QoS requirement is given low priority by the PCIe scheduler, the QoS goal will be violated due to excessive wait time. In Section 4.3.4, we propose a priority switching policy to achieve the QoS goal for those tasks with lower priority.

4.3.4 Priority Switching for Semi-QoS Management

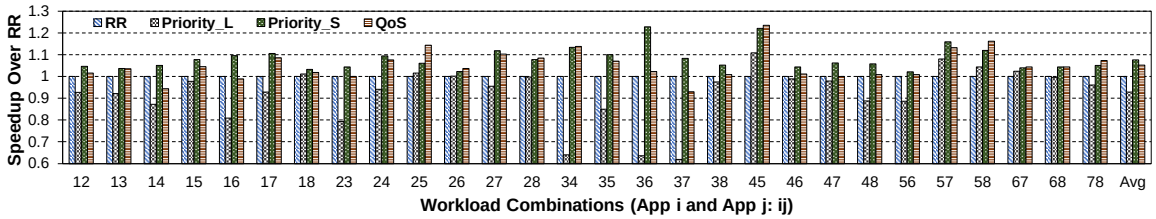


Figure 4.10: Total throughput of the tested multi-GPU system.

Priority-based scheduling will increase the PCIe and GPU utilization, and hence improve

the throughput of the entire multi-tenant GPU system. However, in many domains, some applications may have QoS requirements (e.g., the application should complete within a deadline). When such a workload is hosted in a multi-GPU architecture, priority-based PCIe scheduling alone is insufficient since the QoS task may need more resources than other tasks.

For QoS tasks, the goal of scheduling is to achieve the QoS target. Once the QoS target can be met, the scheduler will then attempt to maximize the global throughput. In this work, we define the latency of each workload as the QoS target. The true QoS requirement will account for end-to-end latency, which includes the GPU execution time. We assume the OS will be able to define a set of partial QoS goals for various resources. Our scheduling policy will provide semi-QoS management as part of the QoS management deployed in the OS. In this work, our partial QoS goal is thus the memory transfer time that is spent on the PCIe bus.

We introduce a QoS-Oriented Scheduler (QoOS), as shown on the right side of Figure 4.9 to implement semi-QoS management. To predict whether the memory transfer can complete in time, QoOS keeps track of the transfer time of each command for the QoS tasks associated with each GPU on the PCIe interconnect. As the remaining data size can be collected by the TOS, and the PCIe bandwidth is known, we can predict the time to send the remaining data for this command. We compute the highest priority using Equation 4.3:

$$T_{trans} = B_R/BW, \tag{4.3}$$

where T_{trans} is the time left to transfer the data, B_R is the remaining number of byte to transfer, and BW is the PCIe bandwidth.

Initially, priorities are ranked by data sizes of different tasks, in order to achieve high throughput for the entire system. For QoS tasks, we set a deadline for each iteration of the task (recall that we run multiple iterations of the same workload for testing purposes) to approximate the QoS requirements in a real scenario.

Once QoOS predicts the T_{trans} is too long and will miss the deadline, the priority of the current task is raised to the highest level to try to meet the QoS goal and updating the TC/VC remapping.

4.3.5 Preliminary Evaluation

We extend MGPUSim [93] to evaluate our priority-based scheduling Scheme. The modeled multi-GPU system consists of a CPU with four AMD R9Nano [38] GPUs. In this work, we run two

APP ID	Abbr.	Workload	Size Per Iteration (KB)
1	SC	Simple Convolution [90]	328.4
2	MM	Matrix Multiplication [90]	128
3	MT	Matrix Transpose [90]	1024
4	AES	AES-256 Encryption [91]	65.2
5	FIR	FIR Filter [91]	256.1
6	KS	KMeans Clustering [91]	131.1
7	MP	Maxpooling [92]	64
8	RL	Relu [93]	32

Table 4.3: Benchmarks

kinds of tasks, launched by two users to model multi-tenant sharing. For experimental purposes, a task with a larger data size is assumed to be the QoS task, while the task with smaller data size is assumed as a non-QoS task. The deadline of the memory transfer time is set as *3 times* the memory transfer time without bandwidth contention. The 8 workloads we used to evaluate the proposed solution are from the *AMDAPPSDK* [90], *Hetero-Mark suite* [91] and *clCaffe* [92] suites, as shown in Table 4.3. We select the problem size to find a good trade-off between the simulation time and common use-cases of applications.

To explore the concurrent execution of tasks, we generate 28 (8*7/2) combinations in total. Each task is repeatedly launched on a GPU until the total measured time exceeds 0.3 seconds. To evaluate system throughput, we use the total number of iterations executed during 0.3 seconds, normalized to the total number of iterations running on a single GPU system, as our figure of merit. The number of total iterations for all workloads is 81.75 on average, thus it is large enough to avoid the effect of the job arrival skew.

Four PCIe scheduling policies are implemented, including (**RR**), Priority scheduling with large data size first (**Priority_L**), Priority scheduling with small data size first (**Priority_S**) and QoS support of Priority scheduling with small data size first (**QoS**).

Figure 4.10 shows the total multi-GPU throughput for four PCIe scheduling policies, normalized to RR. We make three observations. First, *Priority_S* achieves the highest performance and outperforms *RR* by 7.6% on average. Second, giving high priority to memory commands that possess a smaller data size is better than giving high priority to memory commands with larger data sizes, as the idle time of the GPUs is reduced. Third, although *QoS* cannot perform as well as

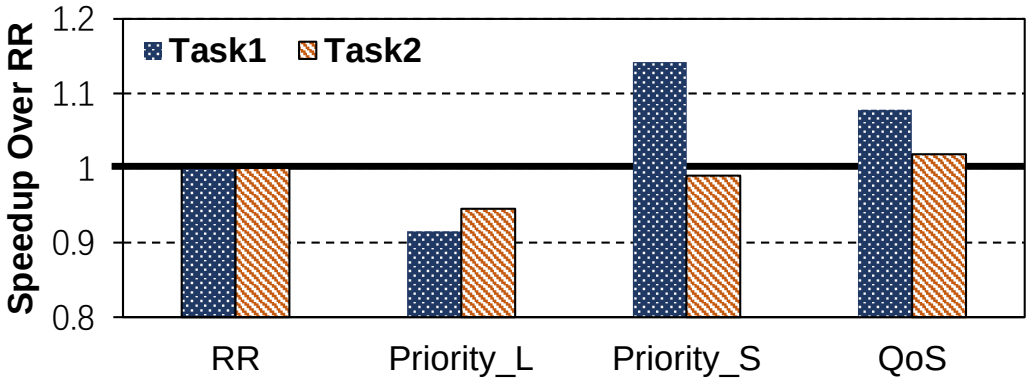


Figure 4.11: Throughput of two concurrent tasks running on the Multi-GPU system.

Priority_S due to QoS requirements, it still outperforms *RR* by 5.3% on average.

Figure 4.11 shows the performance of two concurrent tasks. Task 1 has a smaller data size, while task 2 has a larger data size. Both tasks in *Priority_L* perform worse than in *RR* due to long stall times, waiting for large datasets to complete the transfer. In *Priority_S*, although task 1 can outperform *RR* by 14.2%, the performance of task 2 is decreased by 1% due to the low priority of this task. In priority scheduling policy With QoS support, though there is no complete fairness, both tasks perform better than *RR*. Moreover, the QoS achievement rate can be higher.

Chapter 5

Multi-GPU Collaborative Computing

Multi-GPU computing is a promising solution that can continue the performance scaling of GPU systems. Today, a large number of high-performance computing workloads support multi-GPU computing. In this dissertation, we perform a thorough study of multi-GPU system performance characteristics and design multi-GPU architecture support for the emerging multi-GPU applications.

5.1 MGPUMark

5.1.1 Multi-GPU Collaborative Computing Patterns

Execution patterns include types of behavior that repeatedly appear in program execution. The pattern of a program is usually determined by both algorithm constraints and implementation decisions. In this work, we consider a scenario where the data to be processed is large, so that duplicating the data to each GPU adds too much overhead, and is impossible to run on a single GPU due to memory size limitations.

Studying multi-GPU collaborative execution patterns can help us cover most types of multi-GPU execution with a smaller number of benchmarks. It can also guide programmers and system designers to optimize programs and systems for specific targets. Note that the patterns introduced here are not meant to be exhaustive, nor mutually exclusive. One multi-GPU program may use more than one pattern, or may use patterns that we do not characterize in this dissertation.

Partitioned Data: The *Partitioned Data* pattern describes a type of algorithm that naturally allows both the input and output data to be partitioned on each GPU. The result is that no inter-GPU memory accesses are required. This pattern is frequently observed in streaming applications, such

as AES encryption [94], and the Blacksholes algorithm [95], where the input and output have a one-to-one mapping. This pattern usually relies on a head node (a CPU or a GPU) to partition the data and broadcast the data to each GPU to process each batch. As no inter-GPU communication is required, this pattern is likely to achieve good scalability, and hence, should be used whenever possible.

Adjacent Access: The *Adjacent Access* describes a pattern where the GPUs need to access data, that is closely related to their own local data, from other GPUs. This pattern is frequently observed in signal processing [96], stencil algorithms [97], and physical simulations [98, 99], as calculating one output at one particular index needs the input data from surrounding indices that are resident on a neighboring GPU. If the data that needs to be accessed from another GPU is read-only, we can maintain multiple copies of the data to avoid inter-GPU access, at the cost of using more GPU memory space. Otherwise, we can keep the data partitioned on each GPU and allow each GPU to issue inter-GPU accesses occasionally. Adjacent accesses involve a relatively small amount of inter-GPU communication, and therefore, can be a good option compared to data duplication.

Gather: This pattern describes a commonly used computing paradigm, where every GPU in the system needs to read remote data from the other GPUs, but each GPU will only write to its own local memory. The *Gather* pattern can be used in reduction style computing (e.g., adding two vectors element-wise or calculating the sum of a vector) as each GPU needs to synthesize a larger amount data to create a smaller output. When the data is too large to fit in one GPU's memory, or the data is already on each GPU, we can use a Gather operation. The Gather pattern requires the system to process inter-GPU read requests with rather low latency.

Scatter: Opposite but similar to Gather, *Scatter* describes a pattern where each GPU needs to input data from a local GPU and output data to the entire GPU address space. This pattern is used when the input data can be partitioned on each GPU, while the output location is non-deterministic.

Irregular: We summarize all other patterns as following an *Irregular* pattern, and includes patterns when any GPU needs to both read and write data from/to the entire GPU address space. This data reference pattern occurs in many sorting and graph algorithms, as the access pattern is data-dependent. The Irregular pattern presents performance challenges since it may result in frequent inter-GPU communication. The programmer should try to use other patterns before settling for an Irregular pattern. Also, whenever this pattern is used, the programmer should make every effort to keep memory accesses within a local GPU and avoid inter-GPU accesses.

5.1.2 Workloads

We select a suite of workloads from public-domain libraries and benchmark suites, including the AMDAPPSDK 3.0 [31] (BS, MT, SC) and HeteroMark [100] (AES, FIR, KM), as well as one benchmark (GD) developed from scratch. Workloads are modified with new OpenCL kernels supporting multi-GPU execution, and extended with a Go main program compatible with the simulator.

Advanced Encryption Standard (AES): AES 256-bit encryption [94] is an encryption algorithm widely used in the security domain today. It involves a large number of bitwise operations to convert the plaintext to ciphertext, making it a compute-intensive workload. Our partitioned implementation breaks up the plaintext into chunks and broadcasts the chunks to the GPUs. Each GPU then works on its own chunk of the data, with no need to access any remote data.

We include this benchmark to test the Partitioned Data pattern. We also use this benchmark to validate our model for sub-dword-addressing, a distinct feature of the GCN3 and later AMD ISAs [35].

Bitonic Sort (BS): Bitonic Sort [101] is a sorting algorithm that suits the GPU's massively parallel architecture. It has a predefined order to compare pairs of values in the array to be sorted, making it highly data-parallel.

We include the Bitonic Sort algorithm to test the Irregular pattern. Although the memory access order is predefined, each GPU needs to read from, and write to, any location in the unified memory address space. It also scans a wide range of memory addresses repetitively, putting significant stress on the cache system.

Finite Impulse Response Filter (FIR): FIR [96] is a fundamental algorithm from the digital signal processing domain. In FIR, each work-item multiplies the filter kernel with a portion of the input data in an element-wise manner and sums all the results together.

We include FIR to test the Adjacent Access pattern, as for each GPU, the first few work-items on each GPU need to access the input data that is stored on another GPU. Its large memory footprint can help us analyze how inter-GPU memory access may have a significant performance impact.

Gradient Descent (GD): Gradient descent [102] is an important step used in optimization problems such as DNN Training. Gradient descent evaluates the gradient values for a set of mathematical functions and uses the gradient value to update each function's parameters. When running on a multi-GPU system, gradient descent is usually performed in a data-parallel fashion, as each GPU

processes a mini-batch of the data (i.e., the Partitioned Data pattern). At the end of calculating the gradient on each GPU, the gradient values need to be averaged. Calculating the average inevitably involves inter-GPU communication.

We include the GD workload as it is one of the most widely used algorithms that requires the Gather pattern. Its large memory footprint is also a good test case to stress the inter-GPU interconnect.

KMeans (KM): KMeans [103] is an important clustering algorithm widely used in unsupervised machine learning applications. The GPU is responsible for calculating the distance from each input node to each of the centroids, while the CPU updates the centroid location.

We select the KMeans benchmark to evaluate the Partitioned Data pattern. This workload is different from AES, which also follows the Partitioned Data pattern, in two respects: i) KMeans is a more memory intensive workload, and ii) KMeans repetitively accesses the same memory locations in multiple kernels, making it more sensitive to the cache design.

Matrix Transpose (MT): Matrix Transpose is one of the building blocks common in more complex matrix operations. Work-items from one work-group first load matrix data to the local data share memory (an addressable memory space with similar latency to the L1 caches), and then write the data back to the memory in the transposed locations.

Although MT can be implemented using both the Gather pattern and the Scatter pattern, we include the Matrix Transpose benchmark to test the Scatter pattern. Each GPU is responsible for a specific number of columns in the output matrix. Since each GPU stores a few rows of both the input and output matrix, each GPU can read from local memory and write to other GPUs. We also use MT to validate the simulator on Local Data Store (LDS) operations.

Simple Convolution (SC): Simple convolution is a common operation in the image processing domain. It is also a fundamental step in convolutional neural networks (CNNs). SC performs a convolution operation on 2-dimensional images.

We include SC to test the Adjacent Access pattern in a 2-dimensional problem. Although the image to be convolved can be partitioned across multiple GPUs, each GPU needs to access a remote partition for the input pixels on the margins.

5.2 MGPUSim

5.2.1 GPU Simulator Design Principles

Architectural simulators have been one of the most important tools to guide early design space exploration, performance optimization, and pre-silicon verification. Developing an accurate and extensible simulator is essential for the research community to explore a wide range of design possibilities.

In the following paragraphs, we discuss a number of design principles that simulators should follow, though are absent in many current simulators.

DP-1: Simulate state-of-the-art machine-level ISA. Cutting-edge research explores cutting-edge features, and hence, new ISAs and new microarchitectures need to be evaluated. Existing simulators are generally simulating old ISAs or intermediate representations. For example, Multi2Sim [23] emulates the GCN1 ISA, which is four generations older than current AMD product. GPGPU-Sim [33] mainly models the NVidia Fermi architecture that was released in 2010. In addition, researchers have highlighted major issues when performing performance analysis while simulating at an intermediate language level versus using the actual machine code ISA [24, 65], resulting in misleading performance. Therefore, while any simulator will immediately become quickly dated due to the pace of development in GPU technology, the research community needs a simulator that can simulate a new and feature-rich machine-level ISA.

DP-2: "Open to Extension, Closed to Modification." When studying performance/power/reliability with an architectural simulator, researchers usually need to reconfigure, or more commonly, modify, the simulator to fit the needs of their intended study. Modifying the inter-dependent components in a simulator is non-trivial and may require modifying a large number of files. It tends to be more problematic when combining the modifications from different developers, as each developer may need to modify common files.

According to the "Open-Closed Principle" [104], one should be able to extend a simulator without modifying it. When adding more functionality to a simulator, researchers should not need to modify source files. Instead, they should write new extensions for the simulator and plug the new extensions into the existing simulator to realize new configurations. This approach can also help support the reproducibility of results, since each module can be clearly defined and reused [105].

DP-3: No magic. It is tempting for simulator developers to overuse the flexibility that a software design offers to overcome the complexity of the simulated hardware design, typically

manifested in intricate queuing systems, asynchronous buffers, or low-level communication protocols.

As an example of "magic", the implementation of a GPU may directly invalidate the caches by invalidating all directory entries, ignoring the fact that in real hardware, this action involves a message to be sent from the command processor to each cache module. Manipulating the state of one module from another is a clear sign that the simulator is not tracking the behavior of real hardware, and this may impact simulation accuracy. When a simulator developer uses "magic", it hurts both the accuracy of the simulator, as well as encapsulation and modularity of the code.

DP-4: Track both timing and data. Directly inferred from the "no-magic" rule, a simulator should model the actual data-flow in both the memory system and the instruction pipelines, rather than only calculating the simulated time. Execution simulation that maintains data values offers two advantages: (1) Minor mistakes in the simulator will be detected as a mismatch of output values, rather than a difference in the estimated time. If the result generated by the simulator matches execution on the target hardware, we can guarantee that the modeled hardware is at least feasible. (2) A performance model or power model may be data dependent [106, 107]. Maintaining data in each module under simulation helps us support data-dependent modeling, which can improve accuracy.

DP-5: Simulate multi-threaded hardware with multi-threaded software. A GPU supports a massively parallel execution model. There are a large number of units concurrently executing independently on a GPU. Therefore, it should be possible to use multiple CPU threads to simulate GPU execution. In addition, properly applying locks in a multi-threaded program to prevent race conditions and avoid deadlocks is usually a difficult job. The design of the simulator should provide a locking scheme that both guarantees performance and avoids the hazards described above.

DP-6: No busy ticking. Busy ticking (i.e., constant checks of module states) is a common reason for low simulation performance, and should be avoided. In current simulator designs (e.g., GPGPUSim), modules usually need to check their internal state every cycle, even if the states do not need to be updated. This is a common problem for cycle-based simulation. Multi2Sim [23] partially solves the problem by using a hybrid cycle-based and event-driven simulation scheme. However, some modules still need to keep retrying actions each cycle, such as cache reads to the cache while the network is busy. To achieve good simulation performance, a next-generation simulator should avoid busy-ticking whenever possible.

5.2.2 Akita Simulator Framework Desgin

Central to our design is the simulator framework, Akita. We embrace a domain-agnostic design approach so that the Akita can be used to model any component such as a different GPU model, a CPU, or an accelerator device. Akita consists of the following four parts:

1. The Event-Driven Simulation Engine: We define an event as a state update of a component. Akita's event-driven simulation engine maintains a queue of events for the whole simulation and triggers events in chronological order.

2. Components: Every entity of the simulated computer platformis a component. In our case, a GPU, a Compute Unit (CU), and a cache module are examples of components.

3. Connections: Two components can only communicate with each other through connections using requests. Connections are also used to model the intra-chip interconnect network and inter-chip interconnect network.

4. Hooks: Hooks are small pieces of software that can be attached to the simulator to either read the simulator state or update the simulator state. The event-driven simulation engine, all the components, and the connections are hookable. Hooks can perform non-critical tasks such as collecting execution traces, dumping debugging information, calculating performance metrics, recording reasons for stalls, and injecting faults (for reliability studies).

The Akita event engine supports parallel simulation, fulfilling *DP-5*. Leveraging the fact that the events that are scheduled at the same time do not depend on each other, the event-driven simulation engine harnesses multiple CPU threads to process events. We embrace a conservative parallel event-driven scheme (the chronological order of events are not interrupted), so that we guarantee the parallel simulation results will match a serial simulation.

The component system and the request-connection system enforce strict encapsulation of components. We restrict a component from scheduling events for other components, and at the same time, we do not allow a component to access another component's state (by reading/writing field values, using getter/setter functions or function calls). All communication must use the request-connection system. This design choice forces the developer to explicitly declare protocols between components. The benefits of this design are three-fold. *First*, a developer can implement a component, considering only the communication protocol. *Second*, we gain flexibility since we can replace a component with any other component following the same protocol. Extending the simulator just involves adding a new component that implements the same protocol and wiring the new component with other components. By adopting this model, we fulfill the requirement of *DP-2*. *Third*, we can

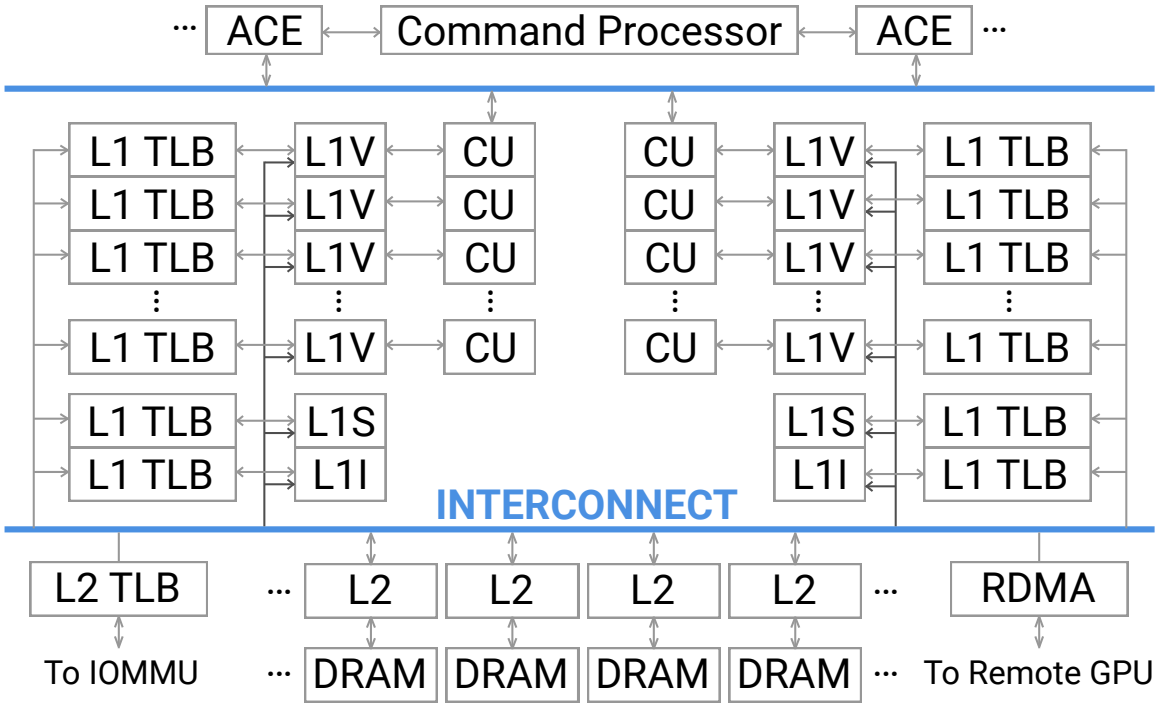


Figure 5.1: The Modeled GPU Architecture.

improve simulation accuracy as no information can "magically" flow from one component to another, without being explicitly transferred through the interconnect. Therefore, we can satisfy both *DP-3* and *DP-4* with our design approach.

The event-driven simulation and the connection system can help avoid busy ticking (*DR-6*). For example, a DRAM controller may be able to calculate that a request takes 300 cycles to complete when the request arrives, and nothing needs to be modeled in detail during the 300 cycles. So the DRAM controller can schedule an event in the event-driven simulation engine after 300 cycles and skip state updates until then. In addition, another type of busy ticking in GPU architectures is caused by components that repeatedly try to send data. Since a component has no information about when a connected connection becomes available, the component has to retry each cycle. To avoid this type of busy ticking, we allow the connections to explicitly notify connected components when the connection is available. Therefore, a component can avoid updating the state if all of the out-going connections are busy, and update its state after a connection is available.

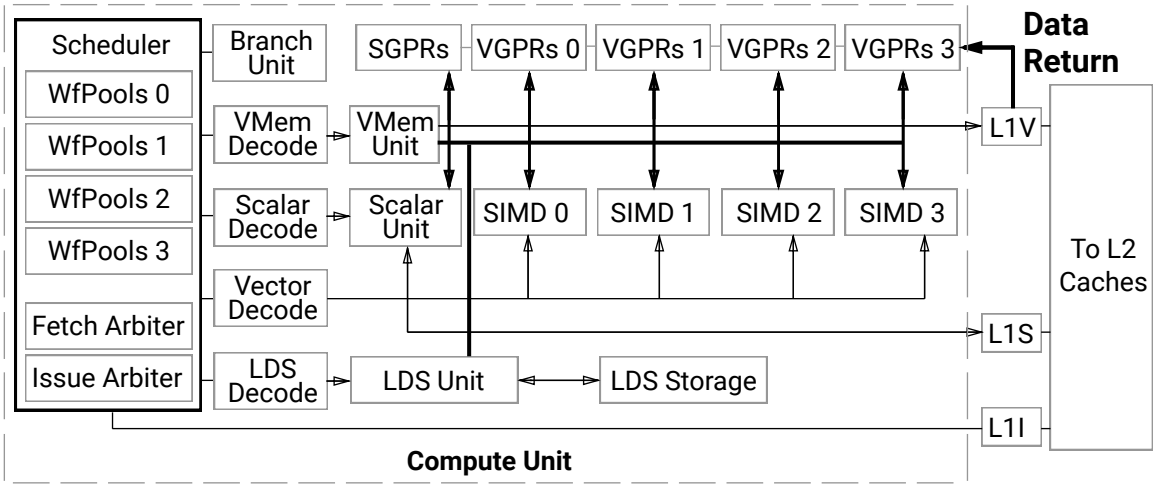


Figure 5.2: The Compute Unit Model.

5.2.3 GPU Modeling

MGPUSim applies the Akita simulation framework to model the GPU model shown in Figure 5.1. MGPUSim faithfully models the Graphics Core Next 3 (GCN3) ISA. We configure the model of the GPU according to the publicly available AMD documentations and through microbenchmarking. While the latest ISA on AMD Vega GPU's runs GCN5 [108], GCN5 only extends the memory access instructions. The compute instructions in GCN5 are the same as for GCN3. Simulating GCN5 can be achieved in MGPUSim by just adding support for the new memory access instructions. We do not need to change the remaining core components of MGPUSim.

The GPU architecture is composed of a Command Processor (CP), Asynchronous Compute Engines (ACEs), Compute Units (CUs), caches, and memory controllers. The CP is responsible for communicating with the GPU driver and starting kernels with the help of ACEs. The ACEs dispatch wavefronts of kernels to run on the CUs.

In our model, a CU (as shown in Figure 5.2) incorporates a scheduler, a set of decoders, a set of execution units, and a set of storage units. The CU includes a scalar register file (SGPRs), vector register files (VGPRs), and a local data share (LDS) storage. A fetch arbiter and an issue arbiter decide which wavefront can fetch instructions and issue instructions, respectively. Decoders require 1 cycle to decode each instruction, before sending the instruction to the execution unit (e.g., SIMD unit). Each execution unit has a pipelined design that includes read, execute, and write stages.

MGPUSim includes a set of cache controllers, including a write-through cache, a write-around cache, a write-back cache, and a memory controller. By default, the L1 caches and the L2

caches use a write-through and write-back policy, respectively. The cache controllers do not enforce coherence as the GPU memory model does not require cache coherency. The compute units send virtual addresses for read and write requests to the L1 cache. Virtual addresses are translated to physical addresses at L1 cache with the help of two levels of TLBs. We show the default configuration in Figure 5.1. However, both the number of layers of caching and the number of layers of TLB are fully configurable. Finally, we equip each GPU with a Remote Direct Memory Access (RDMA) engine to manage the inter-GPU communication.

5.2.4 Simulator APIs

MGPUSim can run in two different modes, native mode and Go mode. In native mode, we provide a customized implementation of the OpenCL runtime library in the C programming language. Users can link the MGPUSim-provided OpenCL library with the workload executables so that the customized OpenCL library can redirect the API calls to MGPUSim and run the GPU kernels on the simulated GPUs. In Go mode, we allow user to write a main program in Go to define memory layout and launch kernels.

MGPUSim's GPU driver provides a set of OpenCL-like APIs to allow workloads to control the simulated GPUs in Go mode. Each user workload should start by calling an `Init` function to create an execution context for the following API calls. Then, the workload can invoke device discovery functions and use the `SelectGPU` function to specify the GPU to use. Finally, the main body of the workload can be implemented by using memory allocation, memory copy and kernel launch APIs. Since the APIs are similar to OpenCL, an experienced OpenCL programmer should feel very comfortable when using the MGPUSim APIs. In addition, we let each workload, the driver, and the simulation each are run in individual threads, allowing multiple workloads to run in parallel in the simulator.

5.2.5 Simulator Validation Methodology

In this section, we describe the simulation configurations, the full set of microbenchmarks and full benchmarks that we utilize for validation and evaluation of the design studies. For validation of MGPUSim, we compare the execution of microbenchmarks and full benchmarks against a multi-GPU hardware platform that has 2 AMD R9 Nano GPUs (Section 5.2.9).

Parameter	Property	# per GPU
CU	1.0 GHz	64
L1 Vector Cache	16KB 4-way	64
L1 Inst Cache	32KB 4-way	16
L1 Scalar Cache	16KB 4-way	16
L2 Cache	256KB 16-way	8
DRAM	512MB	8
L1 TLB	1 set, 32-way	96
L2 TLB	32 sets, 32-way	8
IOMMU	shared by all GPUs	-
Intra-GPU Network	Single-stage XBar	1
Inter-GPU Network	PCIe-v3 16GB/s	-

Table 5.1: Specifications of the Modeled R9 Nano GPU.

5.2.6 Simulation Configuration

We validate MGPUSim against a multi-GPU platform with 2 Intel Xeon E2560 v4 CPUs and 2 AMD R9 Nano GPUs (details provided in Table 5.1) using execution time as the validation metric. The CPUs and the GPUs are connected via a 16GB/s PCIe 3.0 interconnect. The system runs the Radeon Open Compute Platform (ROCm) 1.7 software stack on a Linux Ubuntu 16.04.4 operating system. We lock the GPUs to run at the maximum frequency to avoid the impact of Dynamic Voltage and Frequency Scaling (DVFS). All the kernels are compiled with official ROCm compiler. All the timing results are collected using the Radeon Compute Profiler [109]. All the experiments presented are performed in the Go mode.

We evaluate the simulator speed and multi-threaded scalability using a host platform with a 4-core Intel Core i7-4770 CPU. We use the environment variable `GOMAXPROCS` to set the number of CPU cores that the simulator can use.

5.2.7 Microbenchmarks

To fine-tune the GPU model in MGPUSim, we develop a set of 57 microbenchmarks that cover a wide range of instruction types and memory access patterns. Each microbenchmark is composed of a manually written or script-generated GCN3 assembly kernel, a C++ host program

used in native execution, and an additional host program written in Go for simulation. For the sake of brevity, out of the 57 microbenchmarks used in this work, below we discuss four microbenchmarks that serve as a good representative of the complete set:

ALU-focused microbenchmark: This Python-generated microbenchmark generates kernels with a varying number of ALU operations (`v_add_f32 v3, v2, v1`) followed by an `s_endpgm` instruction to terminate the kernel. Using the ALU microbenchmark, we validate instruction scheduling, instruction pipeline, and instruction caches.

L1 Access-focused microbenchmark: This microbenchmark generates a varying number of memory reads to the same address. All accesses, except for the first one are L1 cache hits, which allows us to measure the cache latency.

DRAM Access-focused microbenchmark: This microbenchmark repeatedly accesses the GPU DRAM using a 64-byte stride. Since all cache levels use 64-byte blocks, all accesses are expected to incur both L1 and L2 cache misses, and ultimately read from the DRAM. We use this microbenchmark to measure the DRAM latency.

L2 Access-focused microbenchmark: This microbenchmark first reads each cache line in a 1MB block of memory, loading the whole 1MB into the L2 cache. The L1 cache is expected to retain the last 16KB, which is equal in size to its total capacity. After this, a second scan sweeps the same 1MB of data from the beginning, causing L1 misses and L2 hits. We use this strategy to find the L2 cache latency.

5.2.8 Full benchmarks

Out of the wide variety of full benchmarks available in the AMD APP SDK [31] and the Hetero-Mark suite, we select a set of representative benchmarks (listed in Table 5.2) for both simulator validation and our 2 case studies. We select these benchmarks to ensure a wide coverage on the inter-GPU memory access patterns. We modify the benchmarks to run on multi-GPU platforms. For validation experiments, we duplicate the workload to run on two GPUs, while for the design studies, the workloads remain the same, and we dispatch portions of the workload to each individual GPU. We use a different approach during the design studies versus the validation experiments, because we want to focus primarily on multi-GPU collaboration in the design studies. Therefore, we let the multiple GPUs work on a single set of data.

Abbr.	Workload	Multi-GPU Memory Access Pattern
AES	AES-256 Encryption	**Partition**: Each GPU works on its own batch of data. No inter-GPU communication is needed.
BS	Bitonic Sort	**Random**: Any GPU can read/write from/to any other GPU. Memory access patterns are different from kernel to kernel.
FIR	FIR Filter	**Adjacent Access**: The input array is equally divided into batches for each GPU. The filter data, which is small, is duplicated to each GPU. The calculation on each GPU needs to access a small portion of data close to the batch division from another GPU.
KM	KMeans Clustering	**Partition**: KMeans contains two kernels, one matrix transpose and one calculates the distance from each input point to the cluster centroids. In the second kernel, each GPU works on its own batch of data. We have frequent CPU-GPU communication in this benchmark.
MT	Matrix Transpose	**Scatter**: Each GPU reads data from their local DRAM, but writes data to remote GPUs' DRAM.
MM	Matrix Multiplication	**Gather**: Each GPU reads data from local and remote GPUs, but only writes data to local DRAMs.
SC	Simple Convolution	**Adjacent Access**: The input image is divided into sub-images and copied to each GPU. Each GPU needs to access some of the pixels that are copied to another GPU.

Table 5.2: Full Benchmarks and their multi-GPU memory access patterns.

5.2.9 Simulator Validation

In this section, we discuss the results of validating MGPUSim against real hardware. For emulation results, by running MGPUSim in either functional emulation mode or timing simulation mode, we are able to compare the simulation results with the results on AMD GPU hardware at bit-level granularity. This comparison enables us to build confidence in the correctness of instruction emulation and memory consistency in our simulator. For timing simulation accuracy, we show the results of running microbenchmarks and full-benchmarks in the following subsections.

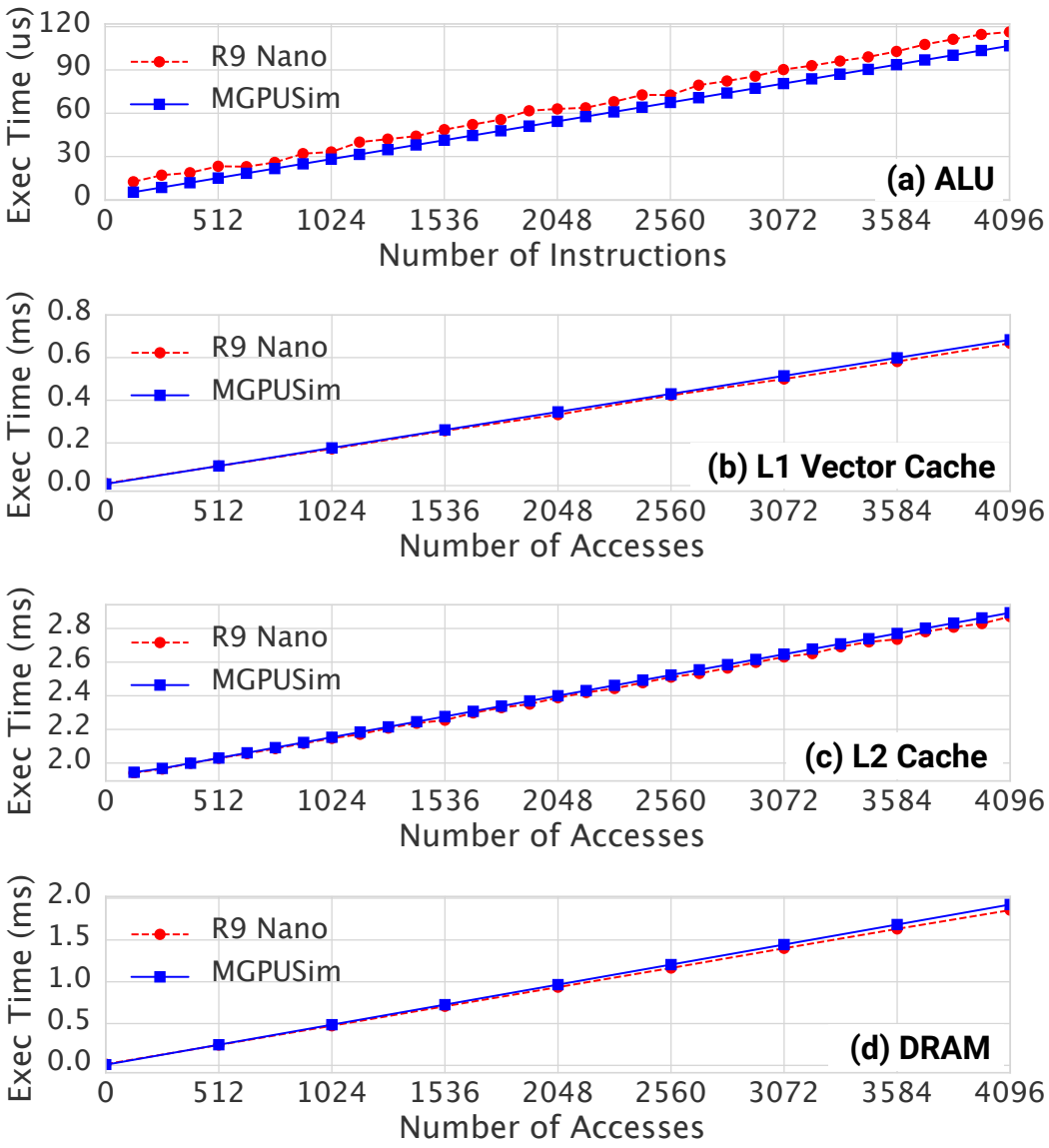


Figure 5.3: Simulator Validation with Microbenchmarks.

Microbenchmark Validation: Figure 5.3 shows a comparison of the execution time of the different microbenchmarks discussed in Section 5.2.5 when runing on an R9 Nano GPU and in MGPUSim. MGPUSim achieves very high accuracy when running these microbenchmarks. For the L1 Vector Cache, L2 Cache and DRAM microbenchmarks, the two curves overlap indicating the high accuracy of our simulator. In the ALU benchmark, there is an offset of several microseconds between the two lines, which is introduced by random DRAM refreshes. We also validate our simulator with microbenchmarks that test other important components such as the ACEs, the L1 constant cache, and

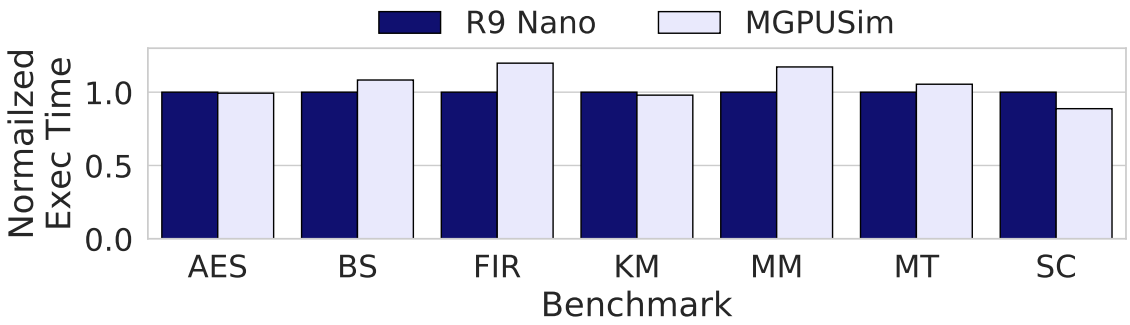


Figure 5.4: Execution time comparison between R9 Nano and MGPUSim for the benchmarks listed in Table 5.2.

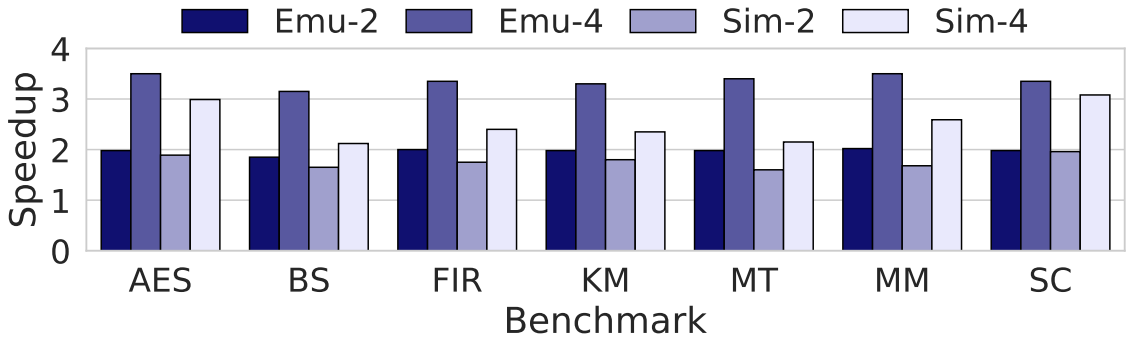


Figure 5.5: Speedup of Functional Emulation (Emu-), and Detailed Timing Simulation (Sim-) using 2 and 4 CPU Threads.

the TLBs. Since these experiments all produce similar results, as the simulator estimated execution time curves fully overlap or track closely with the real-GPU execution time curve, they are not included here. In light of all our simulation results using the microbenchmarks, we can confirm that MGPUSim can model the key GPU components with high fidelity.

Full-benchmark validation: Next, we validate our simulator with full benchmarks running on 2 R9 Nano GPUs. Figure 5.4 shows a comparison of the simulator estimated execution time and the real hardware execution time. The difference between the two values across all benchmarks has an average value of 5.5% (peak value of ≈20% in FIR and SC benchmarks). After a comprehensive study, we can confirm the differences are mainly due to undocumented GPU hardware details. Although we try to model every individual GPU component, we are not able to capture all the hardware implementation details, such as subtle pipeline structures in the cache modules and sizes of the network buffers.

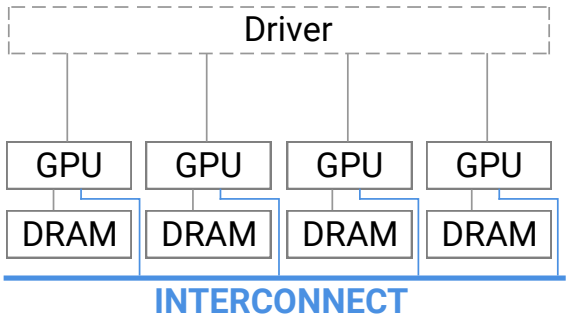
Parallel Simulation Performance: To compare MGPUSim with Multi2Sim 5.0 and GPGPU-Sim, we run all three simulators configured with a single-GPU running the MT benchmark on an Intel Core i7-4770 CPU. Our experiment reveals that our simulator can reach $\approx$ 27 Kilo-instruction per second (KIPS) with 4 CPU threads. For Multi2Sim 5.0 and GPGPU-Sim, we obtain a simulation throughput of ≈ 1.6 KIPS and ≈ 0.8 KIPS, respectively. MGPUSim is 16.5$\times$ and 33.8$\times$ faster than Multi2Sim 5.0 and GPGPU-Sim, respectively.

To support efficient design-space exploration in the context of multi-GPU platforms, unlike contemporary GPU simulators, we designed MGPUSim with built-in multi-threaded execution to further accelerate the speed of simulations. Our simulations can take advantage of the multi-threaded/multi-core capabilities of contemporary CPU platforms. As shown in Figure 5.5, MGPUSim achieves good scalability when using multiple threads to run simulations. In particular, when 4 cores are used in the Intel Core i7-4770 CPU platform, MGPUSim can achieve $3.5\times$ and $2.5\times$ speedups in functional emulation and architectural simulation, respectively, while preserving the same level of accuracy as in single-threaded simulation. In addition, our parallelization approach is domain-agnostic, allowing the parallelization approach to remain valid as we extend the features of the simulator.

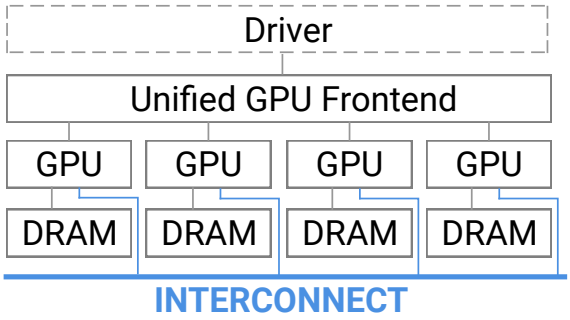
5.3 Reducing Inter-GPU Traffic with Software-Based and Hardware-Based Approaches

Multi-GPU systems require GPUs to access the data placed on remote GPUs. Data associated with remote memory accesses is sent through a low-bandwidth, high-latency, interconnect. As we have shown in Figure 4.6, the CPU-GPU and GPU-GPU interconnects provide much lower bandwidth as compared to accessing data local on the local GPU's memory. Also, the latency over the interconnect can be tens of microseconds long [63, 64], which is equivalent to tens of thousands cycles considering that modern GPUs typically run at 1-2 GHz. The latency and the bandwidth of the interconnects are limited by the electrical properties of these networks, so can not be easily improved. To improve the performance of multi-GPU systems, we need to reduce the inter-GPU traffic.

Both software and the hardware can help reduce inter-GPU traffic in multi-GPU systems. For software-based solutions, we introduce the Locality API, which relies on programmers to adjust the placement of the data and the associated computing threads that process that data, reducing inter-GPU communication traffic and improving multi-GPU system performance. For hardware-based



(a) Discrete Multi-GPU Model.



(b) Unified Multi-GPU Model.

Figure 5.6: Multi-GPU Configurations.

solutions, we design Progressive Page-Splitting Migration (PASI), which is a fully programmer-transparent solution that can automatically adjust the data placement according the placement of the computing threads running on the GPU cores.

5.3.1 Locality API

According to commonly-used GPU programming models, multi-GPU systems can be classified into two categories. The discrete multi-GPU model, as shown in Figure 5.6a, is used by the most commonly used GPU programming frameworks, including OpenCL [36] and CUDA [110]. Both programming frameworks expose all the GPUs to users, enabling them to select where data is stored and how kernels are mapped to devices. Exposing all GPUs to the user delivers the maximum flexibility. However, it can be difficult to adapt single-GPU applications to a multi-GPU platform [16, 25]. Recent studies have explored adopting a unified multi-GPU model, which hides multiple GPUs behind a single GPU interface [16, 17, 25, 71], as shown in Figure 5.6b. A single kernel launch can map to all the GPUs automatically with the help of the GPU driver and the GPU hardware. Thus, the unified multi-GPU model provides better programmability, as the programmer

does not need to modify the GPU program when moving to multi-GPU platforms. However, the unified multi-GPU model may suffer from high-latency inter-GPU communication and non-scalable performance [25, 71].

The proposed Locality API attempts to find a middle ground between the discrete multi-GPU model and the unified multi-GPU model, adding a runtime extension to the unified multi-GPU model. Using the Locality API, a programmer can either treat multiple GPUs as a single large GPU (i.e., a *UGPU*) or as individual GPUs (i.e., an *IGPU*). In addition, as the Locality API is a runtime API extension, a similar set of API extensions can be implemented in any GPU programming framework, such as OpenCL, CUDA, or HSA [111].

5.3.2 API Design

The Locality API is based on the observation that a large portion of regular GPU workloads has a regular and predictable memory access pattern. It is common that GPU programmers know exactly what data is accessed by each work-item. In this case, the programmer can utilize algorithm-specific knowledge to ensure most memory accesses reference the local GPU and avoid costly inter-GPU communication. Since this knowledge is algorithm specific, it is very difficult for a pure hardware-based solution to achieve the same level of optimization.

Our Locality API includes three groups APIs: 1) extended GPU Discovery APIs, 2) Memory Placement APIs, and 3) Compute Placement APIs.

The **extended GPU Discovery API** allows the host program to discover both the UGPU and each IGPU. We assume that there is a memory region associated with each IGPU, so accessing the associated memory is much faster than accessing a remote memory that belongs to another IGPU.

The **Memory Placement API** allows the programmer to explicitly map a range of memory to an IGPU. Since the OS, the Memory Management Unit (MMU), and Input Output Memory Management Unit (IOMMU) manage the Page Table to keep track of both the virtual and physical addresses of pages (a page is usually a contiguous 4KB memory space), we use the GPU driver to modify the page table to map the specified range of virtual addresses to physical addresses on the target IGPU. For example, assuming that we have four IGPUs and each has a 1GB memory space, the physical address space is banked into ranges 0 — 1GB, 1 — 2GB, etc, such that a 16KB vector that has a virtual address of 0 — 16KB can map to a physical address of 0 — 0+4KB, 1GB — 1GB+4KB, etc. If we have a vector-add application, we can simply launch the work-groups that work on the first 4KB to GPU 0 and the wavefronts that work on the second 4KB to GPU 1, and a

similar pattern for GPU2 and GPU3, completely avoiding inter-GPU communication. On a typical 4KB-page virtual memory system, we require that a full (versus partial) page is mapped to a specific GPU to guarantee the correctness of address translation.

The **Compute Placement API** extends the existing kernel-launching API by allowing programmers to specify the IGPU ID to launch the kernel and the list of work-groups to execute on the submodule. The Locality API allows the programmers to provide a callback function to the kernel-launching API. The GPU driver can evaluate the callback function to determine if a GPU needs to execute a work-group.

5.3.3 Progressive Page-Splitting Migration

The Locality API allows programmers to apply their domain-specific knowledge to avoid inter-GPU communication. However, in many cases, allocating the data properly to each GPU is a difficult task. Also, when a single-GPU application is directly migrated to a multi-GPU platform, before it can be manually optimized, we need a scalable solution.

To allow hardware to help with improving data locality, we propose using *Progressive Page Splitting Migration (PASI)*. The goal of PASI is to enable the GPU hardware to automatically improve the data placement for any workload. We explain the design of PASI in the next 3 subsections.

5.3.3.1 Page Migration

When describing Locality API, we assumed that the RDMA engine lies between the L1 and the L2 caches as shown in Figures 5.8a and 5.8b. In the case of an L1 miss, depending on the requested address, the L1 cache will either send the request to a local L2 cache or to the RDMA engine. This Direct Cache Access (DCA) design forces all inter-GPU communication packets to have a payload that is smaller than or equal to the cache line size (64B or 128B in typical systems), resulting in poor utilization of the interconnect bandwidth and spatial locality.

To address this issue, we employ *Page Migration* by rearchitecting the system as shown in Figure 5.8c. A Page Migration Controller (PMC) is integrated in each memory controller. A PMC has its internal page directory stored in the GPU DRAM directly. It introduces a very small amount of extra storage. Assuming a 4GB GPU DRAM and a 4KB page size, PMC comprises at most 1M entries to store the tag data (0.2% overhead assuming each entry is 8B). Each entry contains the physical address tag of the page, and a single valid bit, indicating if the page is mapped to current GPU.

In the case of an L2 miss, a memory access request arrives at the PMC. The PMC checks its internal directory to determine whether the page is currently in the local DRAM. If the page is not present in the local DRAM, it communicates with the RDMA engine to send a page migration request to the input-output memory management unit (IOMMU), which is a hardware component located on the CPU. The IOMMU also maintains a table that tracks which page is located on which GPU. The IOMMU identifies the GPU that owns the data and forwards the migration request to the destination GPU. The owning GPU sends the page data to the requester GPU and then marks the page as invalid on the receiver GPU's local PMC. The page invalidation is also followed by a TLB shootdown (invalidating the page in the TLBs) on the receiver to avoid a translation error. Since the IOMMU knows both the source and destination of the page migration request, it updates its internal directory to reflect the migration.

5.3.3.2 Cache-only Memory Architecture

Page-migration can help improve utilization of the inter-GPU interconnect by increasing the network packet size and can also increase spatial locality. However, due to the fact that multiple GPUs may share the same data, a single page may ping-pong back and forth between GPUs. This can significantly impact workloads such as matrix multiplication, as all pages containing the input data are accessed by all of the GPUs to calculate the output.

In general-purpose GPU computing, memory access patterns of data items can be categorized into 4 types: 1) Single Read; 2) Multiple Read; 3) Single Write; 4) Multiple Write. Types 1) and 3) are commonly seen in streaming workloads and can be fairly easily resolved by the Locality API and Page Migration approaches. However, page migration does not help address issues with access pattern types 2) and 4). Therefore, we use a *Cache-only multi-GPU memory architecture* and *Page Splitting* to solve type 2) and 4), respectively.

To allow the same piece of data to be shared by multiple GPUs, we extend the page migration approach with a memory coherency protocol to unify the multi-GPU system as a cache-only memory system. The concept of a cache-only memory architecture (COMA) [112] describes memory systems that are only composed of cache modules. In COMA, there is no "root" node in the system to always maintain a copy of all the data. In COMA, any piece of data is stored in at least one GPU. But the exact location depends on which GPU uses the data. In contrast to a page-migration approach, a piece of data is allowed to reside in multiple locations with the support of a memory coherency protocol. When the GPU memory is full because of page sharing, we spill the data to

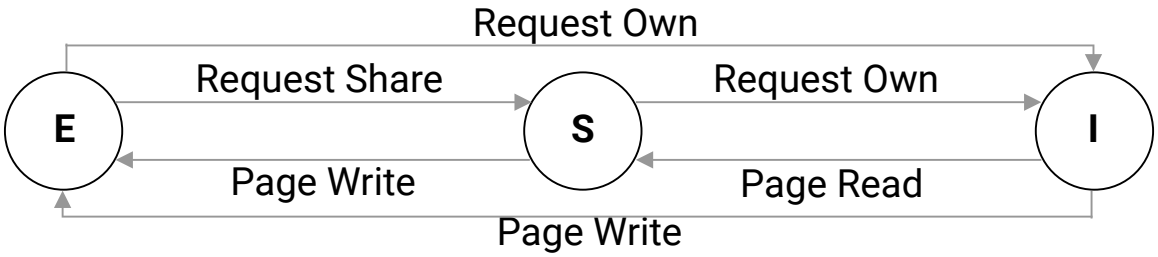


Figure 5.7: The ESI memory coherency protocol for multi-GPU cache-only memory architecture.

system memory, under the control of the IOMMU.

We introduce a light-weight ESI memory coherency protocol, similar to the standard MSI protocol [113], to manage the multi-GPU COMA, where E, S, and I stand for Exclusive, Shared, and Invalid states, respectively. As M in the MSI emphasizes the dirtiness of a cacheline, and the concept of dirtiness does not exist in a cache-only system, we use E to emphasize the write-exclusiveness of a page.

Note that this coherency protocol works at a page granularity rather than a cacheline granularity, and only manages a subsystem that is composed of the memory controllers. It is independent of the cache system. The cache system can still apply cache coherency protocols. The memory controllers of the cache-only memory system can collectively be treated as the root node for the cache system since any piece of data is always available in at least one memory controller.

The ESI coherency protocol works as follows. Assuming a page starts with an "Invalid" (I) state, when an L2 cache reads the data from the page, the PMC requests the data to be migrated, as we described in 5.3.3.1. The IOMMU maintains a memory coherency directory and checks which GPU owns the data. In case only one GPU owns the data in an "Exclusive" (E) state, the IOMMU requests the owning GPU to send the data to the requesting GPU, while marking the state of the page on each GPU as "Shared" (S) state. On the other hand, if multiple GPUs own the data in the "S" state, the IOMMU will select one of the data owners to send the data to the requesting GPU. The selection algorithm is configurable according to the interconnect topology. In our case, as we use a bus to connect multiple GPUs, we let the IOMMU to randomly select an owner to send the data.

The processor writes take a similar approach. When a processor requests to write to a page that is currently in states I or S, the PMC requests page migration from the IOMMU. The IOMMU invalidates the page from all other owners. The page will also change to the "E" state, as it has acquired exclusively and is ready to be written into.

5.3.3.3 Page splitting

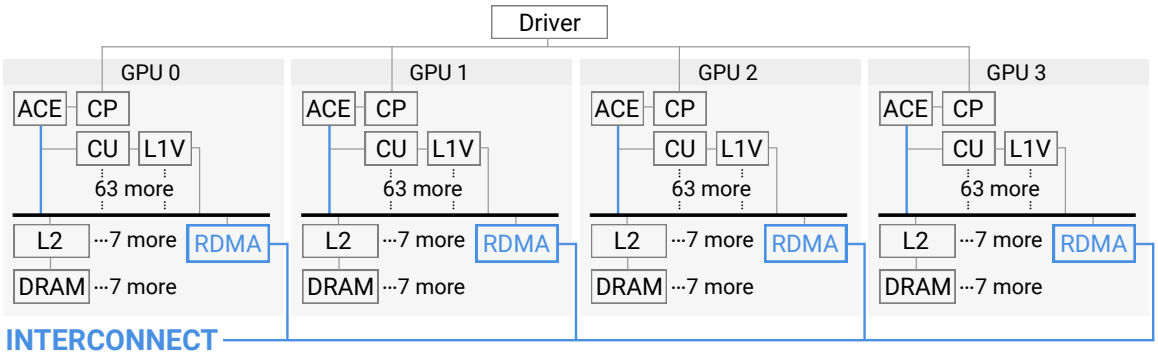
The Cache-only System can avoid useless page-migration when multiple GPUs read from the same piece of data (i.e., the "2) Multiple Read case" described earlier). However, when the same page needs to be written by different GPUs (i.e., the "4) Multiple Write" case), a page still needs to be migrated due to the requirement of exclusiveness of writing. In general, different GPUs should not write into the same address unless a system-level atomic write is used. Writing into different parts of the same page from different GPUs (i.e., false sharing) triggers unnecessary page migration, and should be avoided.

Decreasing the page size is a solution to avoid false sharing. However, smaller pages reduce the coverage of the TLBs and may potentially increase the address translation latency, causing the compute unit to stall. In general, there is no single page size that fits all applications, and even for the same application, different kernels that are part of the application can have a bias towards a particular page size. Therefore, we need a dynamic approach, allowing the hardware to find the best page size during execution.

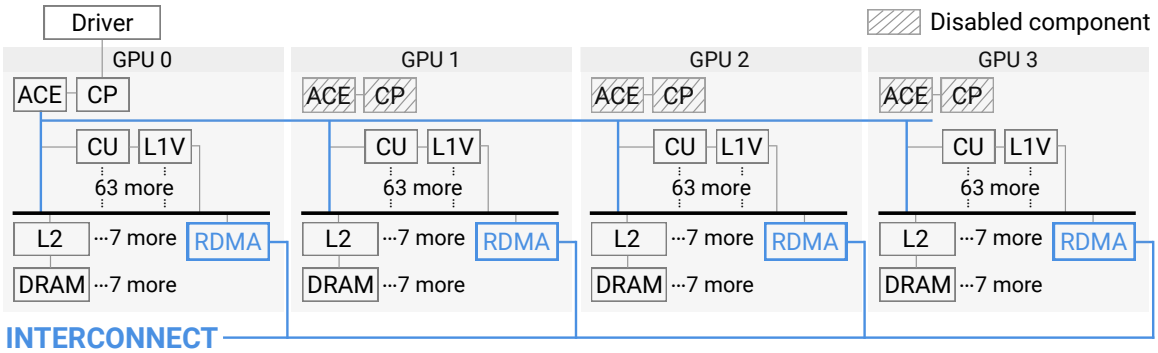
We use a *Page Splitting* approach, built upon a cache-only memory system and the ESI protocol. Starting from a large page size (e.g., 2MB), when a page needs to transit from state "E" to state "I" or "S", rather than migrating the whole page, we split the page in half and only transfer the requested half of the page. We allow the page to continue to be split down until the smallest page size (e.g., 1KB) is reached. Since we split the page in half, each half becomes a page that has an ESI state and an owner list. The IOMMU and the TLB also need to keep track of the page size to guarantee translation accuracy. In addition, whenever adjacent pages arrive at one GPU, the IOMMU merges them into a larger page.

5.3.4 Methodology

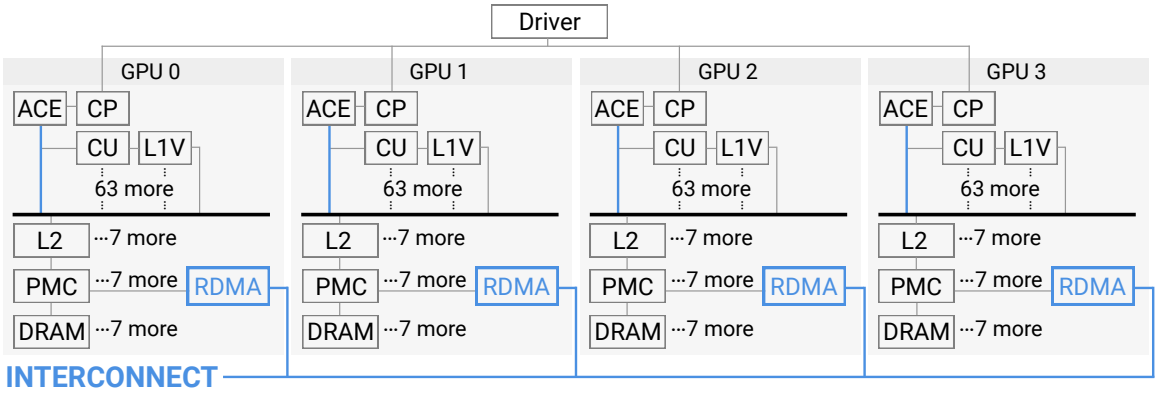
We evaluate the performance of the Locality API and PASI with an extended version of MGPUSim. Modeling discrete multi-GPU platforms (see Figure 5.6a) is straightforward with MGPUSim. For a unified multi-GPU model (baseline), we need to update the platform-configuration function to rewire the GPU components (see Figure 5.8b). Since we use the ACE of GPU0 to dispatch work-groups to the compute units of all the GPUs, we need to add an extra connection to wire the ACE of GPU0 to all the compute units in the platform. We keep the connections between the compute units and the cache modules, the connections between cache modules and memory controllers, and inter-GPU RDMA-RDMA connections unchanged.



(a) Modeling a Discrete Multi-GPU Model.



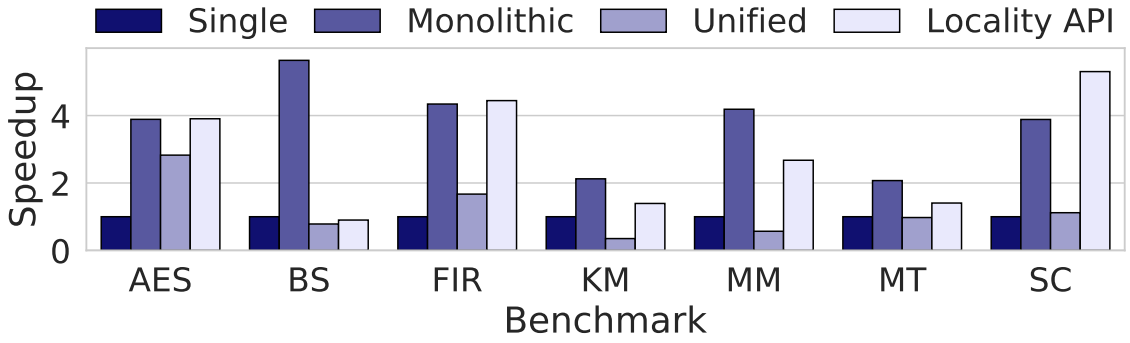
(b) Modeling a Unified Multi-GPU Model.



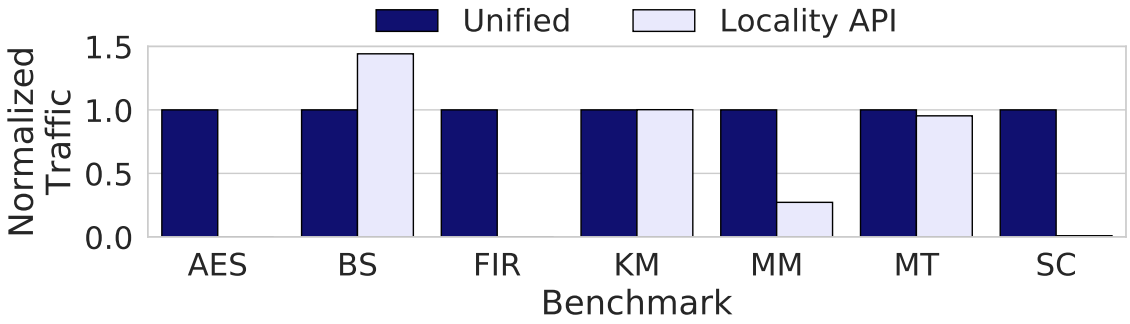
(c) Modeling a Multi-GPU Platform with Page-Migration.

Figure 5.8: Modeling Different Multi-GPU Configurations.

To implement the Locality API in MGPUsim, we replace the driver component to support the extra driver API functions described in subsection 5.3.2. The new driver component wraps the standard driver to avoid re-implementing existing driver APIs. Since we define a communication protocol between the driver and the GPU, and between the driver and the MMU, replacement is easy



(a) Locality API Speedup.



(b) Normalized Inter-GPU Traffic.

Figure 5.9: Locality API Performance.

and straightforward.

To implement PASI, we first introduce a new component named the Page Migration Controller (PMC). The PMC has three ports, including the `L2CachePort`, `DRAMPort`, and the `RDMAPort`, which are responsible for handling the communication between the PMC and its connected components. The PMC can process read and write requests from the L2 cache and determines if the data needs to be fetched from a remote GPU. We provide three different implementations for the PMC, including: 1) page migration only, 2) page migration with the ESI protocol, and 3) page migration, ESI protocol, and page-splitting.

To evaluate these designs, we use the same set of benchmarks as we used to validate MGPUSim (listed in Table 5.2). We use a unified 4-GPU system as the baseline design and compare the time required to execute the kernels.

5.3.5 Evaluation Results

To evaluate the Locality API, we run full benchmarks on four different configurations. We use both single GPU and a monolithic multi-GPU configuration as baselines for comparison. The monolithic GPU is built by integrating the resources of 4 GPUs (CUs, cache modules, memory controllers) into one chip. Note that the monolithic GPU is impractical to build as it requires a large die size $> 2000mm^2$, since each R9 Nano requires a die size of $596mm^2$ [38]. We also compare against a unified 4-GPU configuration, without the Locality API enabled, as another baseline design. The Locality API configuration is based on the same unified 4-GPU configuration, but allows us to apply a locality-based optimization to avoid inter-GPU communication. Using the locality API is equivalent to custom programming for each individual GPU, and therefore, it achieves the same performance as a discrete multi-GPU model.

The inter-GPU traffic, as shown in Figure 5.9b, can be significantly reduced when the Locality API is used. We see that in benchmarks such as AES, as the programmer can perfectly partition the data, with inter-GPU communication can be fully eliminated. For benchmarks that follow the Adjacent Access pattern (e.g., FIR and SC), the inter-GPU traffic can also be minimized. However, in benchmarks such as MT and BS, manual optimization is not easy to apply, and hence, the Locality API cannot reduce traffic, and sometimes may even introduce more inter-GPU traffic.

In terms of the execution time (see Figure 5.9a), we observe that the performance of a monolithic GPU generally scales well and can sometimes even provide speedups of more than $4\times$. This superlinear speedup is due to reduced bank conflicts as we add more L2 cache modules and memory controllers. In benchmarks such as KM and MT, due to an inherent lack of parallelization, a monolithic GPU can only speed up execution by $2\times$.

The benchmark execution time of the Unified and the Locality-API configuration are correlated with the inter-GPU traffic, indicating that inter-GPU communication is a major bottleneck in the system. We see that in many cases (e.g., AES and FIR), the locality API can nearly obtain the same level of scalability as a monolithic GPU, while a Unified configuration is not able to run as fast as a single-GPU design. In FIR and SC, the Locality API can even outperform a Monolithic GPU. This is because the monolithic GPU has a large network connecting the L1 to L2 caches, and so it is more likely to have congestion in the network and the input buffers of the L2 caches. In most of the other benchmarks (AES, KM, and MM), the Locality API can easily improve the unified multi-GPU model. As a special case here, the AES benchmark shows relatively good scalability on all configurations due to the data-parallel compute-intensive nature of the workload. Finally, the

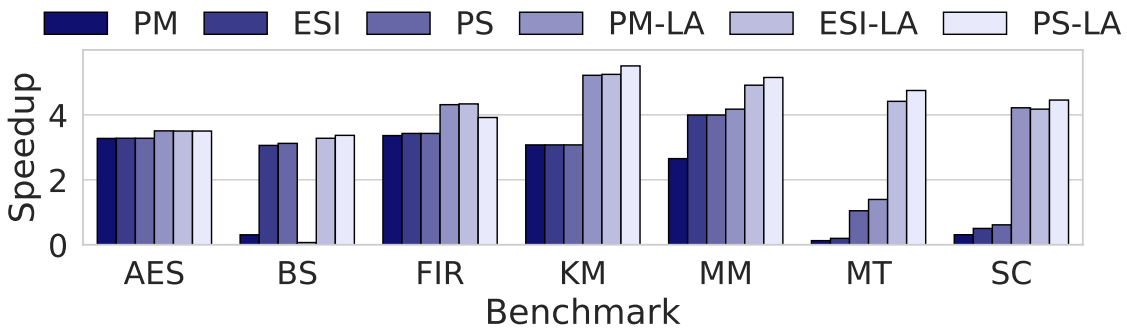


Figure 5.10: The speedup of PASI on a 4-GPU platform over a single GPU with incrementally added features. Here, PM = Page Migration, PS = Page Splitting, and LA indicates that the Locality-API is used.

Locality API cannot speed up the MT and BS benchmarks, since we cannot easily split the data on each IGPU.

According to the evaluation results shown in Figure 5.10, Page Migration alone supports scalability on a unified 4-GPU platform to run the AES, KM, and MM benchmarks. When simulating 4 GPUs, we can achieve a speedup of 3.3×, 3.1×, and 2.7×, as compared to a single GPU execution, when running the AES, KM, and MM benchmarks, respectively, while DCA only achieves a speedup of 2.8× for AES, and a slowdown of 0.35× and 0.56× for KM and MM, respectively. However, in some cases, such as BS, MT, and SC, we see that Page Migration alone slows down the execution up to 4×, as compared to single-GPU execution, mainly because of read-only sharing and false sharing. In these cases, a page cannot reside stably on a single GPU, since ping-ponging between GPUs will occur frequently.

The ESI bars in the Figure 5.10 show the speed up of an approach that combines the ESI coherency protocol and Page Migration. As the combination of these mechanisms can allow the read-only memory to reside in multiple GPUs, a page only needs to be migrated once to each GPU. We observe the effect of ESI in the BS and MM benchmarks, as their inputs are usually shared by multiple GPUs. However, we also notice that ESI and Page Migration are not able to effectively improve MT and SC performance.

Finally, as we integrate Page Splitting with ESI and Page Migration (the PS and PS-LA bars in Figure 5.10), we improve performance by up to 4× (in MT), compared to the ESI + Page Migration approach. This is mainly because: 1) a larger initial page size improves TLB coverage; 2) the initial migration takes more time at the beginning of the execution, but we can avoid future small

page migrations, and thus reduce the wait time of the ALU pipelines; and 3) for applications that have false sharing, Page Splitting can migrate smaller pages and reduce the inter-GPU traffic. Overall, we see that PASI can improve the performance of a unified 4-GPU platform by 2.65× compared to the DCA approach.

Chapter 6

Conclusion

This dissertation proposed Collaborative Heterogeneous Computing. Collaborative Heterogeneous Computing exploits the capabilities of modern computing devices to perform fine-grained, concurrent inter-device communication and computation. We explored Collaborative Heterogeneous Computing down two paths: i) CPU-GPU collaborative computing and ii) multi-GPU collaborative computing. This dissertation carried out different types of performance analysis and performance modeling studies, while proposed new system designs for both the CPU-GPU collaborative computing and the multi-GPU collaborative computing.

6.1 CPU-GPU Collaborative Execution

Although heterogeneous computing allows an application to use both the CPU and the GPU concurrently, most existing applications only use the GPU to perform computing tasks and use the CPU as a scheduler, wasting the computational resources of the CPU. To harness the full capabilitis of the CPU in CPU-GPU heterogeneous computing applications, we need to explore the dynamics of CPU-GPU interaction and propose system designs that can improve performance.

In this dissertation, we first summarized seven common CPU-GPU collaborative computing patterns, including three traditional patterns and four collaborative patterns. Following these design patterns, we introduced Hetero-Mark, a full benchmark suite dedicated to exploring CPU-GPU collaborative execution patterns. We provided at least one benchmark for each design pattern. Using those workloads, we analyzed the efficiency of the Heterogeneous System Architecture (HSA) when leveraging the unified memory space on an APU device. We also presented an evaluation of design patterns, showing how performance can benefit from overlapped execution of the CPU and the GPU.

For the selected applications, we observed as much as a 2.8X speedup (1.8X on average) resulted from employing these patterns on an HSA-enabled device.

Second, this dissertation introduced Multi2Sim-HSA, a HSA runtime API and instruction emulator that is capable of emulating the behavior of both CPUs and GPUs in the system. We evaluated Hetero-Mark benchmarks on Multi2Sim-HSA to understand the vulnerability of the benchmarks to random register bit flips.

Third, we observed that CPU-GPU communication can be a major performance bottleneck for heterogeneous CPU-GPU systems. We found that congestion on the PCIe interconnect can lead to poor device utilization. Therefore, we proposed a runtime-defined, priority-based PCIe scheduling policy to improve system throughput. Experimental results showed that system throughput can be improved by 7.6% on average with the priority-based PCIe scheduling as compared with the Round-Robin-based PCIe scheduling.

6.2 Multi-GPU Collaborative Computing

For multi-GPU collaborative computing, this dissertation first introduced a set of multi-GPU collaborative computing patterns, capturing how multi-GPU workloads access the data on a remote GPU. We also include seven benchmarks that cover the multi-GPU collaborative computing patterns.

Second, we designed and implemented a high-performance, high-flexibility, multi-GPU simulator, MGPUSim. MGPUSim is implemented in the Go programming language for better flexibility and parallel-simulation capability. MGPUSim can faithfully model GPUs that run the AMD GCN3 instruction set. We extensively validated the simulator with both microbenchmarks and full benchmarks. MGPUSim can reduce the average simulation error to as low as 5% in tested benchmarks.

Third, we identified that inter-GPU communication is the main performance bottleneck for multi-GPU systems. Reducing inter-GPU communication traffic can significantly improve performance. To allow programmers to reduce inter-GPU communication traffic with application-specific information, we developed the Locality API. The Locality API is an API extension that allows single-GPU applications to run on multi-GPU systems with minimal code changes. Programmers can explicitly define how to place the data and the computing threads on each individual GPU so that inter-GPU communication can be minimized. We also introduced Progressive Page-Splitting Migration (PASI), a programmer-transparent mechanism that can automatically adjust the data

placement according to the placement of the computing threads in the GPU cores. PASI introduces a cache-only memory architecture and an ESI memory coherency protocol to allow pages to be shared on multiple GPUs. PASI also allows large pages to be split into smaller pages to avoid false sharing. On a 4-GPU system, the Locality API and PASI can improve the performance by 1.6X and 2.6X on average, respectively.

6.3 Contributions of this Dissertation

This dissertation has focused on Collaborative Heterogeneous Computing research. We believe this thesis helps to define this new field of work, and makes a number of fundamental contributions that future work can build on. The range of these contributions includes: novel programming paradigms, new emulators and simulators, new benchmarks suites, and fundamental advances in microarchitectural and runtime design. Specifically, the scope of the many contributions of this dissertations include:

1. A set of seven charateristic CPU-GPU collaborative computing execution patterns that can guide the development of future CPU-GPU collaborative computing applications. We have developed Hetero-Mark, a CPU-GPU collaborative computing benchmark suite comprising real-world workloads, designed to support major GPU programming frameworks, including OpenCL, HC++, HIP, and CUDA. [100]

2. Multi2Sim-HSA, a runtime API and instruction emulator for HSA. Multi2Sim-HSA can emulate the behavior of CPUs and GPUs. [114]

3. A priority-based PCIe scheduling scheme that can greatly reduce the cost of data transfers between the CPU and the GPU over the PCIe interconnect, significantly improving device utilization. [115]

4. A set of five multi-GPU communication patterns and MGPUMark, a benchmark suite that explores fine-grained communication mechanisms in multi-GPU execution. [116]

5. MGPUSim, a state-of-the-art high-performance, highly flexible, high-fideligy, parallel multi-GPU simulator that is based on the AMD GCN3 instruction set. [93]

6. A number of new mechanisms that address scalability in multi-GPU systems, that include systematic software-based and hardware-based solutions, including the Locality API and PASI.

The proposed solutions can significantly reduce inter-GPU traffic and improve multi-GPU system performance. [93]

6.4 Future Work

The main goal of this dissertation was to deliver a rich set of tools that enable Collaborative Heterogeneous Computing and to provide solutions to improve the performance of multi-device Collaborative Heterogeneous Computing. However, for Collaborative Heterogeneous Computing, even if we only limit the scope to CPUs and GPUs, this is a large design space to explore. Next, we suggest a number of future research directions that were enabled by the work presented in this dissertation.

Many modern data processing workloads, such as gene-alignment [117] and graph-based neural network [118], include computing phases that can be easily parallelized and phases that cannot. To improve the performance of these workloads, CPUs and GPUs need to work even closer and switch control flow frequently. Currently, control flow transition between the CPU and the GPU is limited by the kernel launch overhead and by memory synchronization overhead. To solve these problems, hardware-based GPU schedulers built into the CPU cores can avoid the involvement of the GPU driver and can potentially reduce the kernel-launch overhead.

Multi-GPU systems suffer from extensive data movement over the low-bandwidth, high-latency interconnects. In this dissertation, we introduced the Locality API and Progressive Page-Splitting Migration (PASI), both targeted at reducing the inter-GPU communication traffic. However, we still observe that the multi-GPU system performance cannot scale linearly with the number of GPUs in the system. In the future, new approaches, such as GPU-level cooperative thread array (CTA) scheduling and thread migration, need to be explored to further reduce multi-GPU system performance overhead.

Heterogeneous Collaborative Computing is not limited to CPU-GPU and multi-GPU collaborative computing. A wide range of novel computing devices have been proposed in recent years. For example, using dedicated accelerators (e.g., NVIDIA's Tensor Core [119], Google's TPU [120]) are a promising solution that can significantly improve performance and energy efficiency. Meanwhile, memory devices are no longer limited to the traditional SRAM and DRAM technologies. Non-volatile memory [121] and in-memory processing [122] are two main directions that extend the capabilities of current memory devices. Harnessing the power of these new memory technologies needs holistic designs that involve the software, the operating system, and the computing hardware.

Also, researchers have been enhancing network switches with computing capabilities [123] so that they can process the data while forwarding the data to the destination. Future Heterogeneous Collaborative Computing system designs need to incorporate a wider range of devices, including hardware accelerators, non-traditional memory devices, and smart network devices, to achieve performance, energy efficiency, reliability, and security goals.

Bibliography

[1] E. Christophe, J. Michel, and J. Inglada, "Remote sensing processing: From multicore to gpu," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 4, no. 3, pp. 643–652, 2011.

[2] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger, "Fast gpu-based ct reconstruction using the common unified device architecture (cuda)," in *2007 IEEE Nuclear science symposium conference record*, vol. 6. IEEE, 2007, pp. 4464–4466.

[3] Q. Fang and D. A. Boas, "Monte carlo simulation of photon migration in 3d turbid media accelerated by graphics processing units," *Optics express*, vol. 17, no. 22, pp. 20 178–20 190, 2009.

[4] L. Yu, F. Nina-Paravecino, D. Kaeli, and Q. Fang, "Fast monte carlo photon transport simulations for heterogeneous computing systems," in *Clinical and Translational Biophotonics*. Optical Society of America, 2018, pp. JTh3A–38.

[5] J. Salavert Torres, I. Blanquer Espert, A. Tomas Dominguez, V. Hernendez, I. Medina, J. Terraga, and J. Dopazo, "Using gpus for the exact alignment of short-read genetic sequences by means of the burrows-wheeler transform," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 9, no. 4, pp. 1245–1256, 2012.

[6] A. W. Götz, M. J. Williamson, D. Xu, D. Poole, S. Le Grand, and R. C. Walker, "Routine microsecond molecular dynamics simulations with amber on gpus. 1. generalized born," *Journal of chemical theory and computation*, vol. 8, no. 5, pp. 1542–1555, 2012.

[7] R. Salomon-Ferrer, A. W. Götz, D. Poole, S. Le Grand, and R. C. Walker, "Routine microsecond molecular dynamics simulations with amber on gpus. 2. explicit solvent particle mesh ewald," *Journal of chemical theory and computation*, vol. 9, no. 9, pp. 3878–3888, 2013.

[8] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th annual international conference on machine learning*. ACM, 2009, pp. 873–880.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[10] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE micro*, vol. 30, no. 2, pp. 56–69, 2010.

[11] J. J. Dongarra, H. W. Meuer, E. Strohmaier *et al.*, "Top500 supercomputer sites," 2019.

[12] N. A. Gawande, J. A. Daily, C. Siegel, N. R. Tallent, and A. Vishnu, "Scaling deep learning workloads: Nvidia dgx-1/pascal and intel knights landing," *Future Generation Computer Systems*, 2018.

[13] NVIDIA, "Nvidia dgx-2," 2018. [Online]. Available: https://www.nvidia.com/en-us/data-center/dgx-2/

[14] J. Wells, B. Bland, J. Nichols, J. Hack, F. Foertter, G. Hagen, T. Maier, M. Ashfaq, B. Messer, and S. Parete-Koon, "Announcing supercomputer summit," ORNL (Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States)), Tech. Rep., 2016.

[15] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013, pp. 245–256.

[16] G. Kim, M. Lee, J. Jeong, and J. Kim, "Multi-gpu system design with memory networks," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 2014, pp. 484–495.

[17] A. K. Ziabari, Y. Sun, Y. Ma, D. Schaa, J. L. Abellán, R. Ubal, J. Kim, A. Joshi, and D. Kaeli, "Umh: A hardware-based unified memory hierarchy for systems with multiple discrete gpus," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 4, p. 35, 2016.

[18] AMD, "Amd radeon vii specification," 2019. [Online]. Available: https://www.amd.com/en/products/graphics/amd-radeon-vii

[19] NVIDIA, "NVIDIA A100 tensor core GPU architecture, unprecedented acceleration at every scale," 2020. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf

[20] S. Mojumder, M. Louis, Y. Sun, A. K. Ziabari, J. Abellan, J. Kim, D. Kaeli, and A. Joshi, "Profiling dnn workloads on a volta-based dgx-1 system," in *Proceedings of the International Symposium on Workload Characterization*, ser. IISWC'18. IEEE, 2018.

[21] W. Li, G. Jin, X. Cui, and S. See, "An evaluation of unified memory technology on nvidia gpus," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 2015, pp. 1092–1098.

[22] AMD, "Rocm, a new era in open gpu computing," 2018. [Online]. Available: https://rocm.github.io/index.html

[23] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: a simulation framework for CPU-GPU computing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 335–344.

[24] X. Gong, R. Ubal, and D. Kaeli, "Multi2sim kepler: A detailed architectural gpu simulator," in *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on. IEEE*, 2017, pp. 153–154.

[25] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "Mcm-gpu: Multi-chip-module gpus for continued performance scalability," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 320–332, 2017.

[26] H. Jiang, Y. Chen, Z. Qiao, T.-H. Weng, and K.-C. Li, "Scaling up mapreduce-based big data processing on multi-gpu systems," *Cluster Computing*, vol. 18, no. 1, pp. 369–383, 2015.

[27] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun, "Deep image: Scaling up image recognition," *arXiv preprint arXiv:1501.02876*, 2015.

[28] S. Dong, X. Gong, Y. Sun, T. Baruah, and D. Kaeli, "Characterizing the microarchitectural implications of a convolutional neural network (cnn) execution on gpus," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 2018, pp. 96–106.

[29] AMD, "Amd multi gpu box," 2018. [Online]. Available: https://www.exxactcorp.com/AMD-Deep-Learning-Solutions

[30] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang, "A framework for memory oversubscription management in graphics processing units," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 49–63. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304044

[31] AMD, "Amd app sdk 3.0 getting started," 2017.

[32] 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 44–54.

[33] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.

[34] NVIDIA, "Developing a linux kernel module using gpudirect rdma," 2018.

[35] AMD, "Graphics core next architecture, generation 3, reference guide," 2016.

[36] D. R. Kaeli, P. Mistry, D. Schaa, and D. P. Zhang, "Heterogeneous Computing with OpenCL 2.0," 2015.

[37] K. Gregory and A. Miller, "C++ AMP: accelerated massive parallelism with Microsoft Visual C++," 2014.

[38] AMD. (2015) Amd radeon r9 series gaming graphics cards with high-bandwidth memory.

[39] D. Foley and J. Danskin, "Ultra-performance pascal gpu and nvlink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[40] NVIDIA, "NVIDIA TESLA P100," 2020. [Online]. Available: https://www.nvidia.com/en-us/data-center/tesla-p100/

[41] ——, "NVIDIA V100 Tensor Core GPU," 2020. [Online]. Available: https://www.nvidia.com/en-us/data-center/v100/

[42] H. Mujtaba, "Amd vega 20 gpus to feature xgmi gpu-to-gpu pcie 4.0 interconnect , will compete against nvidia's 300 gb/s nvlink 2.0," 2018. [Online]. Available: https://wccftech.com/amd-radeon-vega-20-gpu-xgmi-pcie-4-interconnect-linux-patch/

[43] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[44] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[45] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[46] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, vol. 2011, 2009.

[47] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–11.

[48] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[49] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[50] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 63–74.

[51] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," 2012.

[52] K. Rupp, F. Rudolf, and J. Weinbub, "ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs," in *Intl. Workshop on GPUs and Scientific Applications*, 2010, pp. 51–56.

[53] S. Grauer-Gray, W. Killian, R. Searles, and J. Cavazos, "Accelerating financial applications on the gpu," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM, 2013, pp. 127–136.

[54] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc—first experiences with real-world applications," in *European Conference on Parallel Processing*. Springer, 2012, pp. 859–870.

[55] R. Dolbeau, S. Bihan, and F. Bodin, "Hmpp: A hybrid multi-core parallel programming environment," in *Workshop on general purpose processing on graphics processing units (GPGPU 2007)*, vol. 28, 2007.

[56] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2012, pp. 141–151.

[57] P. Mistry, Y. Ukidave, D. Schaa, and D. Kaeli, "Valar: a benchmark suite to study the dynamic behavior of heterogeneous systems," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM, 2013, pp. 54–65.

[58] Y. Ukidave, F. N. Paravecino, L. Yu, C. Kalra, A. Momeni, Z. Chen, N. Materise, B. Daley, P. Mistry, and D. Kaeli, "Nupar: A benchmark suite for modern GPU architectures," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM, 2015, pp. 253–264.

[59] D. Li, H. Wu, and M. Becchi, "Nested parallelism on gpu: Exploring parallelization templates for irregular loops and recursive computations," in *2015 44th International Conference on Parallel Processing*. IEEE, 2015, pp. 979–988.

[60] L. Wang, M. Huang, and T. El-Ghazawi, "Exploiting concurrent kernel execution on graphic processing units," in *2011 International Conference on High Performance Computing & Simulation*. IEEE, 2011, pp. 24–32.

[61] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in cuda," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014, pp. 1–6.

[62] J. Gómez-Luna, I. El Hajj, L.-W. Chang, V. García-Floreszx, S. G. de Gonzalo, T. B. Jablin, A. J. Pena, and W.-m. Hwu, "Chai: collaborative heterogeneous applications for integrated-architectures," in *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 43–54.

[63] A. Li, S. L. Song, J. Chen, X. Liu, N. Tallent, and K. Barker, "Tartan: evaluating modern gpu interconnect via a multi-gpu benchmark suite," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 191–202.

[64] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. Tallent, and K. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *arXiv preprint arXiv:1903.04611*, 2019.

[65] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor *et al.*, "Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 2018, pp. 608–619.

[66] 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*, vol. 14, no. 1, pp. 34–36, 2015.

[67] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of gpu memory system for multi-application execution," in *Proceedings of the 2015 International Symposium on Memory Systems*. ACM, 2015, pp. 223–234.

[68] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: a gpu memory manager with application-transparent support for multiple page sizes," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 136–150.

[69] D. Bouvier, B. Cohen, W. Fry, S. Godey, and M. Mantor, "Kabini: An amd accelerated processing unit system on a chip," *IEEE Micro*, vol. 34, no. 2, pp. 22–33, 2014.

[70] A. Arunkumar, E. Bolotin, D. Nellans, and C.-J. Wu, "Understanding the future of energy efficiency in multi-module gpus," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 519–532.

[71] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans, "Beyond the socket: Numa-aware gpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 123–135.

[72] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and V. Oreste, "Combining hw/sw mechanisms to improve numa performance of multi-gpu systems," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2018.

[73] H. Kim, R. Hadidi, L. Nai, H. Kim, N. Jayasena, Y. Eckert, O. Kayiran, and G. Loh, "Coda: Enabling co-location of computation and data for multiple gpu systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 3, p. 32, 2018.

[74] HSA Foundation, "HSAIL-Tools," https://github.com/HSAFoundation/HSAIL-Tools, 2015.

[75] J. R. Larus, "Spim s20: A mips r2000 simulator," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1990.

[76] W. H. Wen-mei, *Heterogeneous System Architecture: A new compute platform infrastructure*. Morgan Kaufmann, 2015.

[77] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[78] C. R. Wren, A. Azarbayejani, T. Darrell, and A. P. Pentland, "Pfinder: Real-time tracking of the human body," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 19, no. 7, pp. 780–785, 1997.

[79] C. Stauffer and W. E. L. Grimson, "Adaptive background mixture models for real-time tracking," in *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, vol. 2. IEEE, 1999.

[80] T. Scheuermann and J. Hensley, "Efficient histogram generation using scattering on gpus," in *Proceedings of the 2007 symposium on Interactive 3D graphics and games*. ACM, 2007, pp. 33–37.

[81] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped blast and psi-blast: a new generation of protein database search programs," *Nucleic acids research*, vol. 25, no. 17, pp. 3389–3402, 1997.

[82] W. R. Pearson, "Searching protein sequence libraries: comparison of the sensitivity and selectivity of the smith-waterman and fasta algorithms," *Genomics*, vol. 11, no. 3, pp. 635–650, 1991.

[83] C. Kalra, D. Lowell, J. Kalamatianos, V. Sridharn, and D. Kaeli, "Performance Evaluation of Compiler-based Software RMT in an HSA environment," IEEE, 2016.

[84] Amazon, "Amazon ec2 p3 instances: Accelerate machine learning and high performance computing applications with powerful gpus," 2019. [Online]. Available: https://aws.amazon.com/ec2/instance-types/p3/

[85] Y. Sun, L. Peng, R. A. Lincourt, J. Cardente, Z. Junping *et al.*, "Managing access to a resource pool of graphics processing units under fine grain control," Jun. 27 2019, uS Patent App. 16/287,719.

[86] NVIDIA, "Nvidia dgx-2 system," *https://www.nvidia.com/en-us/data-center/dgx-2/*, 2018.

[87] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant gpu clusters for dnn training workloads," in *USENIX ATC 19*, 2019.

[88] D. Sengupta, R. Belapure, and K. Schwan, "Multi-tenancy on gpgpu-based servers," in *Proceedings of the 7th international workshop on Virtualization technologies in distributed computing*. ACM, 2013, pp. 3–10.

[89] W. Wei, L. Xu, L. Jin, W. Zhang, and T. Zhang, "Ai matrix-synthetic benchmarks for dnn," *arXiv preprint arXiv:1812.00886*, 2018.

[90] AMD, "Amd app sdk 3.0 getting started," 2017.

[91] 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 *IISWC 2016*.

[92] J. Bottleson, S. Kim, J. Andrews, P. Bindu, D. N. Murthy, and J. Jin, "clcaffe: Opencl accelerated caffe for convolutional neural networks," in *IPDPSW 2016*.

[93] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao *et al.*, "Mgpusim: enabling multi-gpu performance modeling and optimization," in *ISCA 2019*.

[94] O. Harrison and J. Waldron, "Aes encryption implementation and analysis on commodity graphics processing units," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2007, pp. 209–226.

[95] J. D. MacBeth and L. J. Merville, "An empirical examination of the black-scholes call option pricing model," *The Journal of Finance*, vol. 34, no. 5, pp. 1173–1186, 1979.

[96] Y. Lim and S. Parker, "Fir filter design over a discrete powers-of-two coefficient space," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 31, no. 3, pp. 583–591, 1983.

[97] A. Schäfer and D. Fey, "High performance stencil code algorithms for gpgpus." in *ICCS*, 2011, pp. 2027–2036.

[98] P. Liu, Z. Qi, H. Li, L. Jin, W. Wu, S.-D. Tan, and J. Yang, "Fast thermal simulation for architecture level dynamic thermal management," in *Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on*. IEEE, 2005, pp. 639–644.

[99] R. Vacondio, A. Dal Palù, and P. Mignosa, "Gpu-enhanced finite volume shallow water solver for fast flood simulations," *Environmental modelling & software*, vol. 57, pp. 60–75, 2014.

[100] 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)*. IEEE, 2016, pp. 1–10.

[101] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–10.

[102] R. Kaleem, S. Pai, and K. Pingali, "Stochastic gradient descent on gpus," in *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*. ACM, 2015, pp. 81–89.

[103] R. Farivar, D. Rebolledo, E. Chan, and R. H. Campbell, "A parallel implementation of k-means clustering on gpus." in *Pdpta*, vol. 13, no. 2, 2008, pp. 212–312.

[104] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[105] C. Pereira, H. Patil, and B. Calder, "Reproducible simulation of multi-threaded workloads for architecture design exploration," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 2008, pp. 173–182.

[106] J. M. Rabaey, A. P. Chandrakasan, and B. Nikolic, *Digital integrated circuits*, 2002, vol. 2.

[107] M. K. Tavana, Y. Fei, and D. R. Kaeli, "Nacre: Durable, secure and energy-efficient non-volatile memory utilizing data versioning," *IEEE Transactions on Emerging Topics in Computing*, p. 1. [Online]. Available: doi.ieeecomputersociety.org/10.1109/TETC.2017.2787622

[108] AMD, "Vega instruction set architecture, reference guide," 2017.

[109] ——, "Radeon compute profiler," 2018. [Online]. Available: https://github.com/GPUOpen-Tools/RCP

[110] Nvidia, "The cuda programming guide," 2008.

[111] W.-m. Hwu, *Heterogeneous System Architecture: A new compute platform infrastructure*. Morgan Kaufmann, 2015.

[112] J. Torrellas, "Cache-only memory architecture," in *IEEE Computer Magazine*, 1999.

[113] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[114] Y.-C. Chung, W.-C. Hsu, S.-H. Hung, T. B. Jablin, D. Kaeli, Y. Sun, and R. Ubal, "Hsa simualtors," 2015.

[115] C. Li, Y. Sun, L. Jin, L. Xu, Z. Cao, P. Fan, D. Kaeli, S. Ma, Y. Guo, and J. Yang, "Priority-based pcie scheduling for multi-tenant multi-gpu systems," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 157–160, 2019.

[116] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, R. Ubal, X. Gong, S. Treadway, Y. Bao, V. Zhao, J. L. Abellán *et al.*, "Mgsim+ mgmark: A framework for multi-gpu system research," *arXiv preprint arXiv:1811.02884*, 2018.

[117] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," in *2008 Symposium on Application Specific Processors*. IEEE, 2008, pp. 101–107.

[118] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008.

[119] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 522–531.

[120] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[121] M. H. Lankhorst, B. W. Ketelaars, and R. A. Wolters, "Low-cost and nanoscale non-volatile memory concept for future silicon chips," *Nature materials*, vol. 4, no. 4, pp. 347–352, 2005.

[122] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Processing data where it makes sense: Enabling in-memory computation," *Microprocessors and Microsystems*, vol. 67, pp. 28–41, 2019.

[123] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating distributed reinforcement learning with in-switch computing," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 279–291.