

國立臺灣大學電機資訊學院資訊工程學系

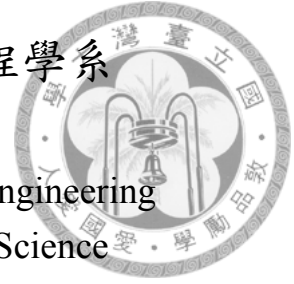
碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis



OpenCL 2.0 模擬器開發及程式特性分析

Workload Characterization and Simulator Development for

OpenCL 2.0

王立

Li Wang

指導教授：楊佳玲 博士

Advisor: Chia-Lin Yang, Ph.D.

中華民國 105 年 8 月

August, 2016



致謝

此篇論文能夠順利完成，由衷感謝在兩年間一路給予我幫助的貴人們。

首先，我必須感謝我的指導教授，楊佳玲楊博士，在我研究的過程中教導了我正確的研究方法及邏輯思考，並指引我正確的研究方向，沒有楊老師的指引解惑，我絕對無法順利完成碩士學位。感謝王柏翰學長的照顧，總是在我遭遇瓶頸時給予我建議，使我能夠解決不斷出現的難題。感謝在口試當日撥冗指導的口試委員陳依蓉陳教授、呂仁碩呂教授，以及陳坤志陳教授，您們給予的實質建議使這篇論文能夠更加完整。

感謝已經畢業的學長吳柄璇、王少甫、許貴松，讓我在剛踏入碩士生活的一年中，研究上，以及課業上，能夠有請教的對象。感謝大我們兩屆的尚軒學長，即使已經畢業，仍然在實驗室網管的工作上十分熱心地給予我許多的指導及協助。

感謝一同和我努力攻讀碩士的同學立展、立維還有君濠，你們和我一起崩潰讓我覺得自己並不孤單，也因此才能夠堅持下去完成碩士學位。感謝碩一的學弟妹們，學俊、益昌、建誼和依柔，你們的垃圾話為實驗室增添了一份人情味。

另外還要感謝交大曹孝櫟教授的學生蔡仁瑋和許勝傑，以及清大李政崑教授的學生王紹仲、甘禮禎、楊峻傑等人，還有所有參與了模擬器開發的人，非常感謝你們能包容我這個不是很會做事的 PM。

最後，我要感謝我的母親和姨媽，有了她們的陪伴及無私的付出，能夠讓我無後顧之憂的完成碩士學位。



摘要

GPU 在異質系統中的定位，已經從過去的圖形加速器，演變到如今能夠處理各種類型的大量運算，也就是所謂的 GPGPU 架構。為了能夠更好地運用 GPU 強大的運算能力，在未來的異質系統架構上，CPU 和 GPU 將會更加緊密地整合在一起。這種架構上的演進為系統架構研究的領域提供了許多不同設計方向上的可能性，然而因為學術界目前缺乏這樣的 CPU 和 GPU 整合的異質系統模擬器，直到目前在這個領域上並沒有太多的研究成果。

本篇論文將修改一個現有的模擬器 gem5-gpu，使其能支援異質運算標準 OpenCL 2.0。選擇 OpenCL 是因為 OpenCL 現今已被各家廠商的硬體所支援，因此我們相信 OpenCL 這個標準足夠代表未來的異質系統架構和運算標準。除此之外我們也會在修改過後的模擬器上估量 OpenCL 2.0 標準中新增加的功能對程式效能的影響，這些功能包括了動態平行、共享虛擬記憶體和原子運算，它們提供了 GPU 更強大的運算功能以及 CPU 和 GPU 間的資料共享，更能體現異質運算功能的強大。

關鍵字 — 異質運算, GPGPU 運算, OpenCL, 模擬器



Abstract

GPU as a computing node in a heterogeneous system, has evolved from an accelerator to a general-purpose computing device that can handle various kinds of tasks. To better utilize the computing power of GPUs, many future heterogeneous systems will integrate CPUs and GPUs more closely. Such heterogeneous system architecture exposes many future architecture research domain, but the lack of a heterogeneous system simulator stops researchers from further exploring this domain.

In this thesis, we'll extend the existing integrated CPU-GPU simulator gem5-gpu to support OpenCL 2.0 standard. We believe that OpenCL as a standard widely adapted by industry will best represent the future design of heterogeneous systems. In addition, we'll conduct some evaluation on our simulator to see the impact of the new features introduced in OpenCL 2.0. These features including device kernel enqueue, shared virtual memory, and enhanced atomic operations, make GPUs computing capability even stronger and enable the opportunity of fine-grained data sharing between CPUs and GPUs, which can demonstrate the powerfulness of heterogeneous computing.

Keywords — Heterogeneous computing, GPGPU computing, OpenCL, Simulator



Contents

致謝	i
摘要	ii
Abstract	iii
1 Introduction	1
2 Background	5
2.1 Baseline Heterogeneous System and gem5-gpu	5
2.2 GPU Architecture and GPU Programming Model	5
2.3 OpenCL	7
2.3.1 Shared Virtual Memory	8
2.3.2 Dynamic Parallelism	9
2.3.3 Platform Atomics and Enhanced Atomic Operations	10
2.3.4 Work-Group Built-in Functions	10
2.4 HSA 1.0	11
3 Simulator Development	13
3.1 Simulator Overview	14
3.2 Customized OpenCL Host API	16
3.3 OpenCL to PTX Compiler	16
3.4 Support newer version of PTX	17
3.4.1 Dynamic Parallelism	17

3.4.2	Platform Atomics and Enhanced Atomic Functions	18
3.5	Work-Group Built-in Functions	19
4	Evaluation	20
4.1	Benchmarks	20
4.2	Experimental Results	22
4.2.1	Validation	22
4.2.2	Shared Virtual Memory	23
4.2.3	Dynamic Parallelism	24
4.2.4	Work-Group Built-in Functions	26
4.2.5	Platform Atomics	28
5	Related Works	31
5.1	Heterogeneous Computing Simulator	31
5.2	Heterogeneous Workload Analysis	32
6	Conclusion	34
	Bibliography	36





List of Figures

2.1	The heterogeneous architecture gem5-gpu can simulate	6
2.2	OpenCL 2.0 work-group built-in functions demonstration	12
3.1	High level overview of the gem5-gpu simulator. The gray parts indicate modified components.	13
3.2	Overview of our proposed changes in gem5-gpu.	15
4.1	Real Hardware and Our Simulator's Normalized Running Time Correlation.	23
4.2	Performance of applications using SVM normalized to applications using memory copy	24
4.3	Running time breakdown for applications using SVM	24
4.4	Performance of applications using dynamic parallelism normalized to OpenCL 1.2	26
4.5	Bilateral filter's overlapping effect using dynamic parallelism	27
4.6	Demonstration of performance difference between OpenCL 1.2 and 2.0.	28
4.7	Performance of applications using work-group built-in functions normalized to OpenCL 1.2	29
4.8	Implementation difference behind different work-group built-in functions.	29
4.9	Number of Directory Access Every 100 Cycles	30



List of Tables

4.1 Simulation Configuration	22
--	----



Chapter 1

Introduction

General-purpose graphics processing unit (GPGPU) computing has evolved from simply a method to accelerating highly data-parallel applications, to now including a broader range of applications. Moreover, latest GPUs introduce the new feature, dynamic parallelism, or device side enqueueing, which makes a GPU able to launch a kernel for itself without CPU's intervention, and thus makes GPU computing more efficient. All these impressive computing capabilities make GPUs an important role from various domains such as Big Data [23], Machine Learning [9], and Signal Processing [16].

On the other hand, CPU and GPU are becoming more tightly integrated as heterogeneous systems are becoming the design paradigm for today's computing systems. The integration of CPUs and GPUs comes in two ways. First, they are being physically integrated in the same chip. This type of integration can be found in AMD's APU and Intel's integrated graphics processor. Second, CPU and GPU are being logically integrated through the software framework. This type of integration abstracts away the low-level hardware details, eases the burden of GPU programming, and makes the programming more flexible for programmers. For example, a unified (shared) memory address space frees programmers from using explicit copies and enables the use of unmodified pointer-based data structures. Currently, there are a few standards that manage such logical integration. The Compute Unified Device Architecture (CUDA) is a proprietary heterogeneous computing standard for NVIDIA GPU devices. OpenCL [28] is another heterogeneous computing standard created by Khronos Group [4] and it targets on various types of


devices from AMD, ARM, Imagination, MediaTek and Qualcomm...etc. Heterogeneous System Architecture (HSA) is a industry heterogeneous computing standard created by HSA foundation [3], and it defines from the high level programming language interface to the low-level ISA and hardware architecture specification.



Traditionally, computer architecture research relies on simulators to evaluate a design's property. While both the software and hardware of heterogeneous computing are being quickly developing in industry, the researching for heterogeneous computing in academia has not been progressed very much due to the lack of such simulator. Although there are currently a few heterogeneous CPU-GPU simulators, none of them supports the new introduced heterogeneous computing standard – OpenCL 2.0. Here we just name a few heterogeneous CPU-GPU simulators:

1. *gem5-gpu* [26]: *gem5-gpu* is an event-driven heterogeneous CPU-GPU simulator that integrates the CPU simulator *gem5* [11] and GPU simulator *gpgpu-sim* [10] into having the same physical memory system and address space. It supports NVIDIA's CUDA 4.0 programming model, but the underlying GPU architecture (*gpgpu-sim*) is capable of running OpenCL 1.2 kernels.
2. *Multi2Sim* [31]: *Multi2Sim* is also an execution-driven heterogeneous CPU-GPU simulator. It supports both CUDA 4.0 and OpenCL 1.2 programming model, and the GPU side supports various of GPU architectures such as NVIDIA's Fermi, AMD's Southern Islands. Although it can simulate more architectures than *gem5-gpu*, the memory systems of its CPU and GPU are separated.
3. *MARSSx86-PTL-SIMT-GPU* [34]: *MARSSx86-PTL-SIMT-GPU* is a trace-driven heterogeneous CPU-GPU simulator. It supports CUDA 3.1 programming model. Like *Multi2Sim*, its memory systems for CPU and GPU are separated. And because it's trace-driven, it cannot simulate as realistic as *gem5-gpu* or *Multi2Sim*, thus is less used in academia.

In this thesis, we will extend *gem5-gpu* to support OpenCL 2.0. We choose *gem5-gpu* among several CPU-GPU simulators is because *gem5-gpu* has already integrated CPU and



GPU to have the same address space, which is closest to the highly integrated architecture we mentioned above, and thus is the most suitable architecture for us to build OpenCL support on top of it. We believe that OpenCL 2.0 as a standard widely adapted by industry will best represent the future design of heterogeneous systems. Mainly there are 4 new features in OpenCL 2.0 compared to OpenCL 1.2: Shared Virtual Memory, which unifies address space for CPU and GPU, eliminating memory copy between CPU and GPU memory space in the old OpenCL 1.2 programming model, Dynamic Parallelism, which supports GPU launching kernels to itself, increasing the flexibility of GPUs, Work-Group Built-in Functions, which efficiently performs reduction operation in GPU kernels, increasing the performance and programmability compared to OpenCL 1.2, and Platform Atomic Functions, which guarantee the write atomicity between CPU and GPU, making CPU-GPU fine-grained data sharing possible.

There are two main challenges to support OpenCL 2.0 on gem5-gpu. First, we need to understand the functionality of these new features. Second, we need to understand the detailed software architecture of gem5-gpu so that we know where to modify to support all new OpenCL 2.0 features. In addition to the simulator development, we will also give some analysis about characteristics of applications using OpenCL 2.0 will have. This thesis makes the following contributions:

- We extend existing gem5-gpu CPU-GPU simulator to support OpenCL 2.0 and verify the simulation by correlating running time with real hardware. To the best of our knowledge, this is the first simulator that can run OpenCL 2.0.
- We conduct a set of experiments to analyze an application's behavior using a certain OpenCL 2.0 feature.

The rest of the thesis is organized as follows. In Chapter 2, we'll explain the background of heterogeneous computing model, GPU architecture, and our simulation framework overview. In Chapter 3, we'll break down simulator changes that are needed to support OpenCL 2.0 into independent software components and give a conceptual explanation on how these components are implemented and how are OpenCL 2.0 features

mapped to these components in the simulator. In Chapter 4, we'll show analysis of applications using OpenCL 2.0 features and their program behavior. Finally, we present related work in Chapter 5 and conclude this thesis in Chapter 6





Chapter 2

Background

This chapter introduces the baseline heterogeneous simulator we are going to extend. We also describe the background knowledge of GPU programming, OpenCL, and the functionalities of the new features in OpenCL 2.0.

2.1 Baseline Heterogeneous System and gem5-gpu

Our baseline heterogeneous system is based on gem5-gpu, an integrated CPU-GPU simulator. The system consists of two parts: a CPU cluster with any number of CPUs modeled by gem5 [11], and a GPU modeled by gpgpu-sim [10]. Here we demonstrate a high level architecture overview with CPU cores and an GPU integrated on the same chip shown in figure 2.1. Both CPU cluster and GPU are connected to a single off-chip memory through a shared directory that manages a directory-based cache coherence protocol between them. Currently gem5-gpu only supports CUDA.

2.2 GPU Architecture and GPU Programming Model

A general-purpose GPU consists of multiple stream-multiprocessors (SMs) or computing units (CUs) containing many scalar units or stream processors (SP) in it. An SM is a Single Instruction Multiple Threads (SIMT) processor that uses thousands of hardware threads to handle Single Instruction Multiple Data (SIMD) computing. The right side of

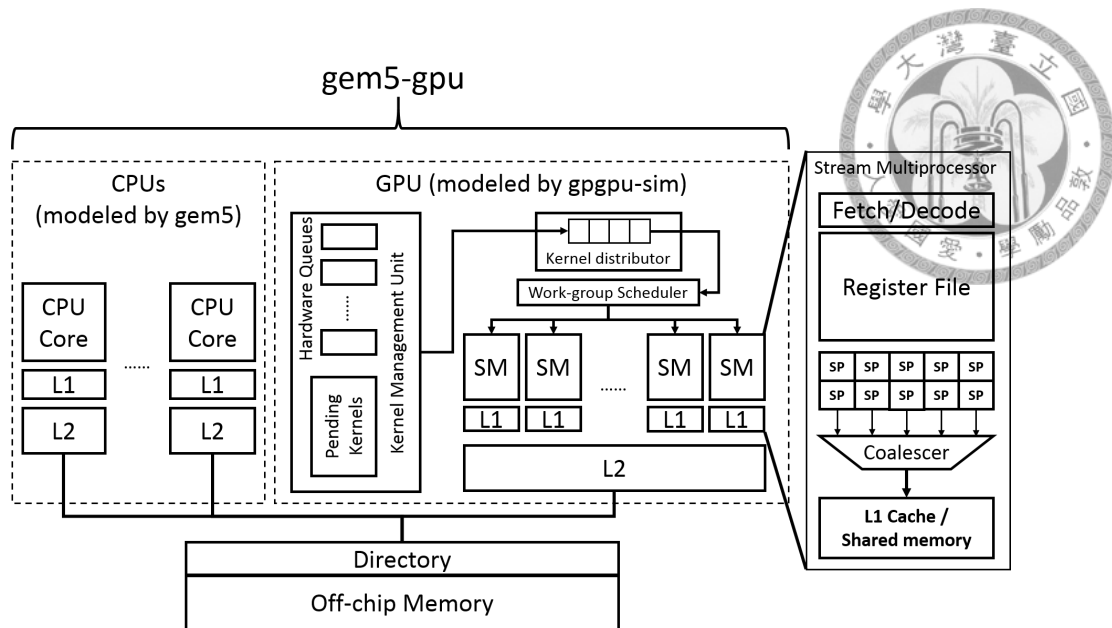


Figure 2.1: The heterogeneous architecture gem5-gpu can simulate

figure 2.1 shows the internal components of an SM, each of which has fetch/decode logic, a large register file to store massive thread contexts, many SPs to compute in parallel, a memory coalescer that merges memory accesses to the same cache line, and an on-chip SRAM which is configurable to be split into L1 cache and scratchpad memory.

A GPGPU application uses massive number of threads that run on the same code called *kernel* to tackle problems. 32 or 64 threads are grouped into a basic running element called *warp* (or *wavefront* in AMD's terminology). Threads in the same warp are running in a lock-step fashion, and memory accesses from a warp will be first coalesced by the memory coalescer mentioned above.

Launching a kernel works as follows. First, programmers must specify the total number of threads in a kernel and the size of a group of threads, called work-group or thread blocks in OpenCL/CUDA's terminology. A kernel launch API call will push a kernel launch command into the software command queue (or stream in CUDA's terminology). The command queue serves as a synchronization method for commands (i.e. kernel launch and memory copy), and each command queue will be mapped to a hardware queue in the kernel management shown in figure 2.1. Second, the kernel management unit will select a kernel from the head of a hardware queue and dispatch it to the kernel distributor. The

work-group scheduler will take one entry from the kernel distributor and dispatch a work-group on a SM every cycle. And finally, a work-group in a SM will be divided into warps to run.



The GPU side simulation in gem5-gpu is modeled by gpgpu-sim, a cycle-level simulator, but all the memory accesses from gpgpu-sim are redirected to gem5's Ruby memory system. The default GPU architecture modeled in gpgpu-sim is NVIDIA's Fermi architecture [5], and the GPU machine language used to simulate is PTX, a virtual ISA created by NVIDIA. Currently gpgpu-sim supports OpenCL by compiling OpenCL kernel code to PTX using real GPU driver, but this driver compilation can only support up to OpenCL 1.2. To support OpenCL 2.0, we'll extend the LLVM compiler [18], which already includes a PTX back-end, to compile OpenCL 2.0 kernel code to PTX. We'll also extend gpgpu-sim to support more PTX instructions to run the new OpenCL 2.0 features more efficiently.

2.3 OpenCL

Open Computing Language, short for OpenCL, is an open standard that is widely adopted by academia and industry for heterogeneous computing. The standard can be broken down into two parts. The first part is the host (i.e. CPU) side C like API [15, 17] that is used to interact with the device (e.g. GPU). The second part is the device side programming language specification based on C99 standard. OpenCL was initially developed by *Apple Inc.*, and is now maintained by Khronos Group.

The first few updates of OpenCL (OpenCL 1.1, 1.2) added new functionality to make it more flexible, including new data types and built-in functions, commands handling from multiple host threads, and IEEE-754 compliance for single precision floating math. It is not until the release of OpenCL 2.0 that we can see the grand picture of heterogeneous computing. OpenCL 2.0 introduced new features such as Shared Virtual Memory (SVM), Dynamic Parallelism, and Platform Atomics. These features give GPU more control over data and execution compared to OpenCL 1.2, and also make CPU-GPU cooperation more flexible. These new features will be further explored in detail in the following sections.



2.3.1 Shared Virtual Memory

Traditional GPU programming model treats CPU and GPU memory space as two different memory space. The only way to make CPU side data accessible by GPU is through explicit memory copies (i.e. *clReadBuffer* and *clWriteBuffer* in OpenCL). The data copy has problems like duplicated contents in both CPU and GPU memory, which leads to resource utilization inefficiency, and data transfer overhead, which will be a potential bottleneck if the input data size is large. In addition, the disjoint memory spaces of CPU and GPU make using pointer-based data structure like linked-lists and trees very difficult.

But the problems mentioned above no longer exist, as OpenCL 2.0 adds the new feature, Shared Virtual Memory (SVM). SVM in OpenCL guarantees that CPU and GPU will have a common virtual address region as long as this region is a memory buffer allocated by OpenCL-provided API (*clSVMAlloc*). For parallel computing, it is an important issue that how SVM buffer, as a shared resource, will be updated. The OpenCL specification defines two types of SVM implementations [8] based on the synchronization granularity of the SVM buffer:

1. **Coarse-grained:** Synchronization is provided when mapping (*clEnqueueSVMMMap*) of unmapping (*clEnqueueSVMUnmap*) of SVM buffer, as well as kernel launch and completion. Coarse-grain SVM buffer has a fixed virtual address for all the devices.
2. **Fine-grained:** Synchronization is provided on the points including those defined for coarse-grained SVM, as well as atomic operations.

Here synchronization refers to an update to SVM buffer so that the new contents can be visible to all other devices. Since gem5-gpu already supported hardware fully coherent memory between CPU and GPU, the SVM implementation will be fine-grained. We believe that fine-grained SVM is the trend for the future CPU-GPU system because compared to coarse-grained SVM, fine-grained SVM not only is easier to use but also makes CPU and GPU co-working on shared data possible.

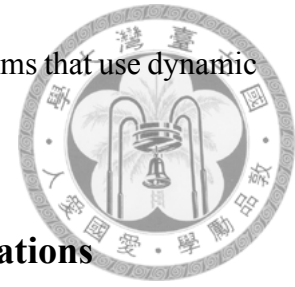
2.3.2 Dynamic Parallelism

The Dynamic Parallelism, or device enqueueing feature in OpenCL 2.0 allows a GPU to launch kernels when running a kernel without giving communicating back to CPU. This feature enables more flexible algorithm design and also makes it possible to handle some irregular or data-dependent applications more efficiently for GPU. Here we name two use cases that could benefit from dynamic parallelism:

1. **Nested Parallelism:** In some irregular applications, because the data structure's property or the algorithm design itself, the loading of work is unable to know until runtime, which makes it hard to partition tasks equally for each thread. For example, in a graph algorithm like BFS, a kernel may map each thread to each nodes in the graph and make them search for their neighbor nodes. Since the number of neighbor nodes for each node is unknown at compile time, normally programmers will use looping in each thread. However, due to the different loop iteration in each thread, there could be serious load imbalance problems. With dynamic parallelism, programmers can easily launch new kernels to search a node's neighbors in parallel to balance the load.
2. **Data-dependent Parallelism:** Data-dependent parallelism happens when one task depends on the other task's result. In the traditional GPU programming model, a programmer can only wait for the first kernel to finish so that he can launch a second kernel if the two kernels are data dependent. The overhead of control transfer between GPU and GPU could drag down the performance. With dynamic parallelism, programmers can launch new kernels once a part of a kernel is done to process on the partial output, and the degree of parallelism is determined at runtime, which makes it flexible enough to handle any kind of data-dependent parallelism.

Although there are many potential benefits from dynamic parallelism, previous work [33] shows that improper use of dynamic parallelism might result in too many kernels being launched, which causes high kernel launching overhead and large memory footprint. As a

consequence, programmers must be cautious when designing algorithms that use dynamic parallelism.



2.3.3 Platform Atomics and Enhanced Atomic Operations

The SVM feature in OpenCL 2.0 gives opportunity for CPU and GPU cooperating on the same data set. But if CPU and GPU thread will touch the same memory address during runtime, a method for synchronization is needed to guarantee their updates in a way programmers want it to be. Traditional GPU programming model uses atomic operations and barriers to synchronize among GPU threads. OpenCL 2.0 inherits atomic operations from the traditional GPU programming model, and adds up a new level of atomic operations, the platform atomics. Using platform atomics will make an update atomically visible to all other devices connected to the SVM. The platform atomics feature is designed to be compatible with the CPU side atomic operations so it doesn't require any change in CPU code. Programmers can use C11 or C++11 atomic operations on the CPU side and use platform atomics on the GPU side to make sure that the sharing data between CPU and GPU will be updated correctly.

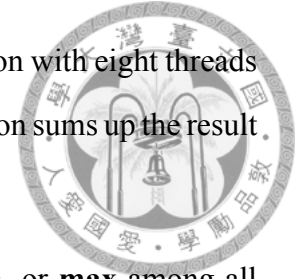
In addition to platform atomics, OpenCL 2.0 replaces the old OpenCL 1.2 atomic functions with the new C11-like atomic functions, which is a more flexible interface and make programmers easier to use.

2.3.4 Work-Group Built-in Functions

The new work-group built-in functions introduced in OpenCL 2.0 provide programmers a high-level function interface to let threads perform reduction, broadcasting, and scanning within a work-group. These functions ease the burden for programmers to write complex code, and further increase the programmability of OpenCL. There are three types of such functions:

1. *Scan*: The scan operation returns the result of **sum**, **min**, or **max** for all threads with thread ID less than current thread, optionally including current thread. Figure 2.2a

shows an example of *work_group_scan_inclusive_add* operation with eight threads in a work-group. Here the "inclusive" indicates that this operation sums up the result including the thread calling this function.



2. *Reduce*: The reduce operation returns the result of **sum**, **min**, or **max** among all threads in a work-group. Figure 2.2b shows an example of *work_group_reduce_min* operation with eight threads in a work-group, which returns the minimal value in a work-group.
3. *Broadcast*: The broadcast operation returns the data of target thread by specifying target thread ID. Figure 2.2c shows an example of *work_group_broadcast* operation with eight threads in a work-group. All threads in a work-group will have the data value from thread 0.

2.4 HSA 1.0

Heterogeneous system architecture (HSA) 1.0 is a standard introduced by HSA foundation. It defines the hardware architecture, programming interface, and an intermediate language called HSAIL that represent a general ISA for devices supporting HSA. HSA supports all of the features in OpenCL 2.0 while providing additional features. Its programming interface can be treated as a high level language directly using by programmers, or as a middle layer that implements OpenCL interface.



Thread ID	0	1	2	3	4	5	6	7
Thread data (before)	3	1	7	0	4	1	6	3



Thread data (after)	3	4	11	11	15	16	22	25
---------------------	---	---	----	----	----	----	----	----

(a) work_group_scan_inclusive_add

Thread ID	0	1	2	3	4	5	6	7
Thread data (before)	3	1	7	0	4	1	6	3



Thread data (after)	0	0	0	0	0	0	0	0
---------------------	---	---	---	---	---	---	---	---

(b) work_group_reduce_min

Thread ID	0	1	2	3	4	5	6	7
Thread data (before)	3	1	7	0	4	1	6	3



Broadcast data from thread 0

Thread data (after)	3	3	3	3	3	3	3	3
---------------------	---	---	---	---	---	---	---	---

(c) work_group_broadcast

Figure 2.2: OpenCL 2.0 work-group built-in functions demonstration



Chapter 3

Simulator Development

This chapter describes the implementation of OpenCL 2.0 on gem5-gpu.

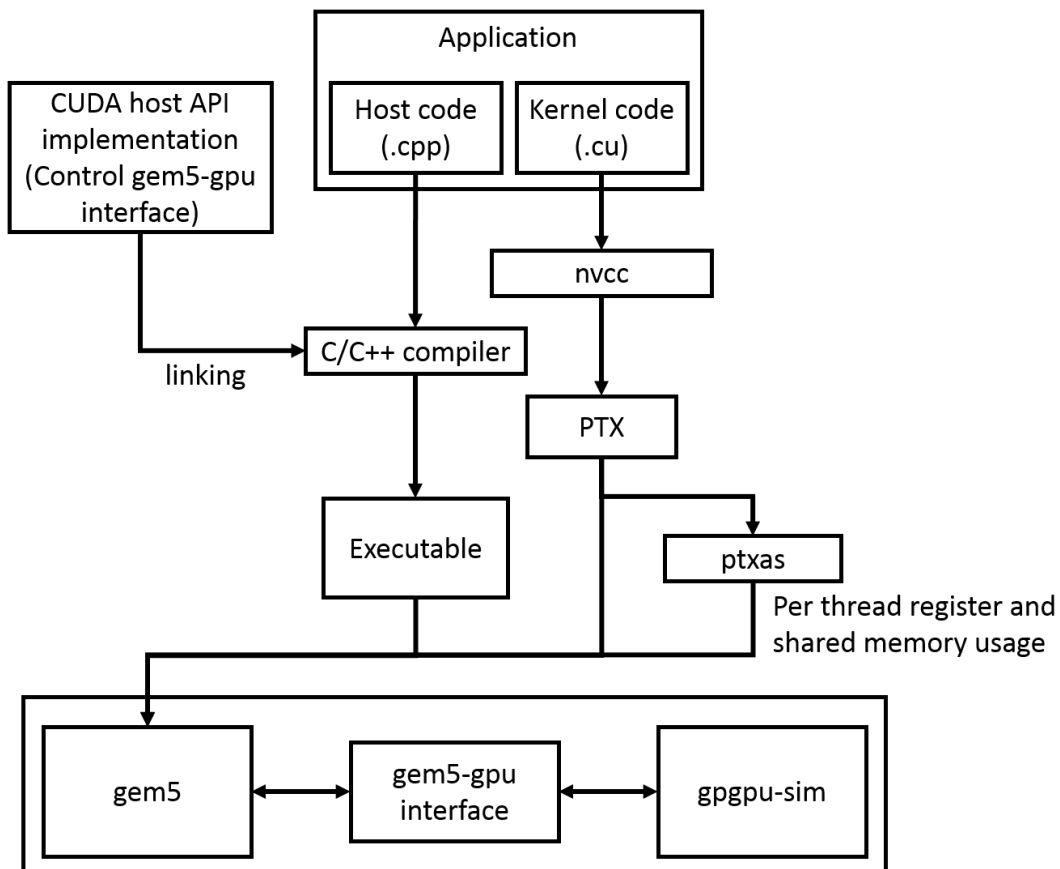


Figure 3.1: High level overview of the gem5-gpu simulator. The gray parts indicate modified components.

3.1 Simulator Overview

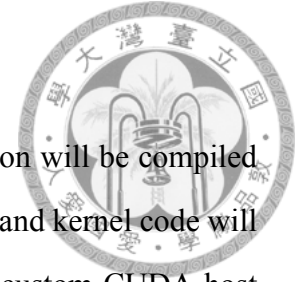


Figure 3.1 shows the high level overview of how a CUDA application will be compiled and run on original gem5-gpu simulator. First, the application’s host and kernel code will be compiled separately. The host C/C++ compiler will link with a custom CUDA host API implementation to compile the host code into executable. The kernel compiler *nvcc*, which is provided in NVIDIA CUDA toolkit, will compile CUDA kernel code into PTX instructions, which serves as GPU assembly code in the GPU side of gem5-gpu (labeled "gpgpu-sim" in figure 3.1). Because PTX as a virtual ISA basically has no restrictions on register usage, gem5-gpu needs information from the PTX assembler (labeled "ptxas" in figure 3.1) that performs realistic register allocation and outputs the resource usage information. The executable, PTX code and resource usage information will all be the input for gem5-gpu. During execution time, when gem5 calls the OpenCL host API, gem5 will execute our custom CUDA host API implementation, which calls into the gem5-gpu interface (labeled "gem5-gpu" in figure 3.1) to send commands to gpgpu-sim. If the command sent to gpgpu-sim is a kernel launch command, gpgpu-sim will read the PTX code and start executing instructions in it. When executing instructions that will send memory requests, gpgpu-sim will call back to gem5-gpu to send memory requests to memory system.

Figure 3.2 shows our proposed changes to support OpenCL 2.0 on gem5-gpu as grey components. Because of the similarity of OpenCL and CUDA, the high level software flow doesn’t change and what we will modify is the internal structure of each component. This is a joint work with National Chiao Tung University (NCTU) and National Tsing Hua University (NTHU). Mainly these changes can be categorized into three parts, and each team is responsible for the development of a part. The three parts are:

1. *Customized OpenCL host API* (developed by NCTU): Like CUDA host API, we need to build up our OpenCL host API for gem5-gpu to make CPU control GPU in a way OpenCL does. This part includes modification in OpenCL host API implementation and gem5-gpu interface.

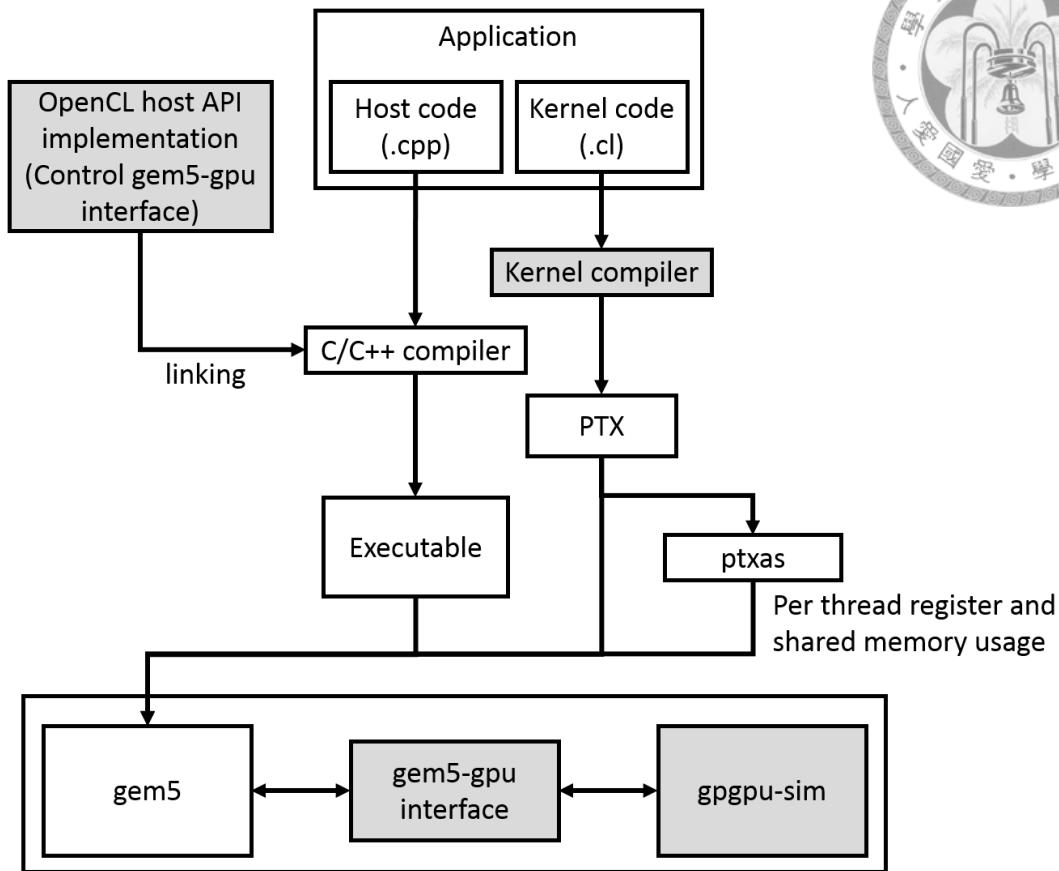
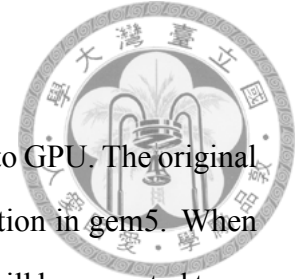


Figure 3.2: Overview of our proposed changes in gem5-gpu.

2. *OpenCL to PTX compiler* (developed by NTHU): Because `nvcc` can't compile OpenCL kernel code, we need to build up a compiler that can compile OpenCL kernel code into PTX instructions.
3. *Support newer version of PTX* (developed by us): Some new features in OpenCL 2.0 require new instruction support. We will upgrade the supported version of PTX in gem5-gpu from 2.3 to 3.1 as in our survey, PTX 3.1 can cover all new features we're planned to add. The modifications include two components, gem5-gpu and gpgpu-sim. For each instruction added we have to add a corresponding function in gpgpu-sim to handle it, and for each instruction that sends memory request we have to add corresponding condition in gem5-gpu interface. This part is the main contribution of this thesis.

The following section describes the implementation of changes mentioned above.

3.2 Customized OpenCL Host API



The OpenCL host API serves as a bridge for CPU to send commands to GPU. The original way gem5-gpu handles CUDA interface is through a pseudo instruction in gem5. When an application runs into the CUDA host API, this pseudo instruction will be executed to go through the gem5-gpu interface to drive gpgpu-sim. This pseudo instruction interface remains unchanged in our OpenCL implementation. The main modifications we make here are labeled "OpenCL host API implementation" and "gem5-gpu interface" in figure 3.2. The OpenCL host API implementation covers functions for the CPU-GPU interface defined by the header file of OpenCL (i.e. `cl.h`). The gem5-gpu interface is a tunnel for gem5 to control gpgpu-sim. Because OpenCL host API maintains a similar interface to CUDA host API (e.g. `clCreateBuffer` and `cudaMalloc`, `clEnqueueNDRangeKernel` and `cudaLaunch`), we can leverage many of the codes from the original gem5-gpu CUDA implementation.

3.3 OpenCL to PTX Compiler

Since gpgpu-sim simulates PTX ISA, a OpenCL to PTX compiler is needed to launch an OpenCL kernel on gpgpu-sim. The original gpgpu-sim uses NVIDIA GPU driver to achieve the compilation, but it only supports up to OpenCL 1.2. Therefore, we need to develop our own OpenCL 2.0 to PTX compiler, labeled "Kernel compiler" in figure 3.2.

To build up such a compiler from nothing will require lots of work, but luckily, LLVM's Clang [2] as an open source project, already has an OpenCL 1.2 frontend and PTX backend, which relieves us from doing most of the hard work. What we have to do is to add new compiling rules for new syntaxes in OpenCL 2.0, including dynamic parallelism, enhanced atomic functions, and work-group built-in functions.

3.4 Support newer version of PTX



This section describes features that need to extend the current PTX ISA and architecture changes in the GPU simulator, which include modifications in `gem5-gpu` interface and `gpgpu-sim`. Originally, `gem5-gpu` and `gpgpu-sim` support up to PTX 2.3 ISA, which is enough to support OpenCL 1.2, but some features in OpenCL 2.0 require GPU hardware changes, such as dynamic parallelism. To support these features, we've surveyed HSAIL to find a suitable version of PTX that can support OpenCL 2.0. Our final target is PTX 3.1 ISA with additional instruction modifiers for atomic operations. These extensions all rely on the kernel compiler's support to generate our extended instructions. From now on we'll use "OpenCL 1.2" to call PTX 2.3 and "OpenCL 2.0" to call our target extended PTX 3.1 ISA.

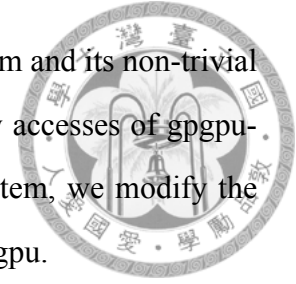
3.4.1 Dynamic Parallelism

The dynamic parallelism code sequence in the PTX level consists of three PTX built-in functions, which act like runtime API to abstract away the detailed implementation. The three built-in functions are:

1. *cudaStreamCreateWithFlags*: This function creates a device command queue to push child kernels in. Threads in the same work-group will get the same command queue calling this function.
2. *cudaGetParameterBuffer*: This function takes the child kernel function pointer and metadata of the child kernel (i.e. the number of work-groups in this kernel and the size of a work-group) as the input and allocates a memory space, called parameter buffer, for parent thread to write parameters to.
3. *cudaLaunchDevice*: This function takes a parameter buffer and a stream as the input, and it will push the kernel that relates to input parameter buffer into input stream.

The implementation is to create a new data path from SM to kernel management unit so that a thread can map its device command queue to kernel management. We leverage

the codes from [32]. This work has implemented dynamic parallelism and its non-trivial overhead modeling on stand-alone gpgpu-sim. Because all memory accesses of gpgpu-sim in gem5-gpu has to be redirected to gem5's Ruby memory system, we modify the memory access code and port the rest of it onto gpgpu-sim in gem5-gpu.



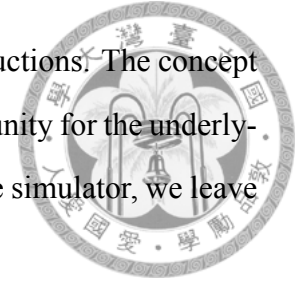
3.4.2 Platform Atomics and Enhanced Atomic Functions

Atomic operation plays an important role in parallel programming as it provides a way to achieve inter-thread communication. Atomic functions have already been implemented in OpenCL 1.2 with the ability to guarantee the write atomicity in a GPU. However, as OpenCL 2.0 brings CPU and GPU closer, it also defines the platform layer atomic functions, which guarantee the write atomicity of all devices connected to the SVM. Besides, to further increase the programmability, OpenCL 2.0 defines a new series of atomic functions and atomic types, which are all syntactically and functionally compliant with new C11 atomic functions. These new atomic functions are parameterized for programmers to decide what type of memory ordering will be used when performing these atomic functions just like the normal C11 atomic functions. Besides, OpenCL 2.0 atomic functions provide three levels of write atomicity to be guaranteed, called *scope*. The three kinds of scopes are:

1. **memory_scope_work_group**: Guarantee write atomicity for all threads in the same work-group.
2. **memory_scope_device**: Guarantee write atomicity for all threads in the same computing device (e.g. a GPU). This is the default scope in OpenCL 2.0, which guarantees the same level of write atomicity as OpenCL 1.2 does.
3. **memory_scope_all_svm_devices**: Guarantee write atomicity for all threads in a device connected to SVM.

Since gem5-gpu has implemented locked read-modify write instruction for GPU and GPU, and PTX already has atomic instructions, what we've done here is to extend new

options (i.e. scope and memory ordering) on the existed atomic instructions. The concept of scope and memory ordering can be seen as an optimization opportunity for the underlying hardware. Currently we haven't realized such optimization on the simulator, we leave it as our future work.



3.5 Work-Group Built-in Functions

The functionality of work-group built-in functions can be implemented using the old PTX 2.3 ISA, which is supported by original gem5-gpu. But PTX 3.0 introduces a new instruction, called warp shuffle, that can effectively exchange thread's data within a warp. Here we choose to use this new instruction to implement work-group built-in functions. Section 4.2.4 will discuss the difference between the PTX 2.3 and 3.0 implementation in detail.



Chapter 4

Evaluation

4.1 Benchmarks

We use benchmarks from AMD's APP SDK v3.0 [1], Pannotia Benchmark Suite, and some computer vision-related applications. The following gives a brief description of these OpenCL 2.0 applications.

Bilateral Filter (BF) [30]: BF is an image filtering technique to smooth images while preserving edges. It has been widely used in different image-processing applications, such as denoising, texture editing, demosaicking... etc. The formulation used in BF is very simple: each pixel is replaced by an average of its neighbors. Here the BF application we use is implemented by recursive bilateral filtering [35]. This is a faster approach to approximate bilateral filtering. The OpenCL 2.0 version of BF will exploit the recursive property using dynamic parallelism.

Optical Flow (OF) [21]: OF is an application that recognizes the pattern of motion of objects. It is an important technique in the field of computer vision and can be used in motion estimation, video compressing. The algorithm used here to implement OF is Lucas-Kanade method, and a pyramidal implementation from [36] is used to increase the robustness.

Ray Tracing (RT) [27]: RT is a global illumination algorithm that generates an image by tracing the path of light through pixels and simulating the effects of light encountering objects (e.g. reflection). It is capable of producing high degree of visual realism and thus

is important in the field of computer graphics. In the implementation of the RT application we use, each thread corresponds to a pixel, and the max recursion depth is 7.

Scale Invariant Feature Transform (SIFT) [20, 19]: SIFT is an algorithm that identifies local image features, which are invariant to image scaling, translation, and rotation, and partially invariant to illumination changes and affine transformation. These features share similar properties with neurons in inferior temporal cortex that are used for object recognition in primate vision. The implementation of this algorithm is through a four-stage computation. First, it detects global potential keypoints. Second, it filters out low-contrast keypoints and edge keypoints to get strong interest points. Third, it assigns orientation to each keypoint to achieve invariance to rotation. Finally, it constructs the keypoint descriptor, which can be used in algorithms that identify objects such as motion tracking, image panorama stitching.

BuiltInScan (BIS): BIS is an application from AMD's APP SDK v3.0 [1] that tests work-group built-in functions. It receives an array of numbers as input and outputs the prefix sum of the array.

RangeMinimumQuery (RMQ): RMQ is an application from AMD's APP SDK v3.0 that tests work-group built-in functions. It receives an array of numbers as input and outputs the minimum value within a range of the array.

PageRank (PRK): PRK is an algorithm used by Google to rank websites in their search engine results [24]. It counts the rank by the number and the quality of links to a website. In each step of PRK, every website (vertex) will contribute its rank value to each vertex it links to. The algorithm terminates until convergence or a user-defined number. Here we use PRK from Pannotia Benchmark Suite [13], which is a benchmark suite containing many irregular GPGPU applications. We rewrite PRK to a version that uses dynamic parallelism to evaluate the performance of dynamic parallelism.

FineGrainSVMCAS (FGSC): FGSC is an application from AMD's APP SDK v3.0 that tests the compare and swap function of platform atomics.

SVMAtomicBinaryTreeInsert (SABTI): SABTI is an application from AMD's APP SDK v3.0 that tests platform atomics. It implements a binary tree insertion algorithm for



Table 4.1: Simulation Configuration

Parameters	Value
CPU Clock	2GHz
CPU cores	4
CPU core L1 Data Cache	64KB
CPU core L2 Cache	1MB
GPU Clock	1GHz
SMs	16
Warp Size	32
Max number of threads / SM	1536
Max number of thread block / SM	8
Number of registers / SM	32768
Shared Memory	48KB
L1 Data Cache	16KB, 128B line, 4-way
L2 Cache	128B line, 8-way associated, total 768KB

both CPU and GPU, and let CPU and GPU insert tree nodes in parallel.

4.2 Experimental Results

The following gives a simple analysis on applications using new features in OpenCL 2.0, which are shared virtual memory, dynamic parallelism, work-group built-in functions, and enhanced atomic functions. For shared virtual memory, work-group built-in functions, and dynamic parallelism, we'll show the performance difference between OpenCL 1.2 and 2.0 implementation and how programmers can benefit from them. And for the new platform atomic functions, we'll focus on the coherence traffic imposed by CPU and GPU co-working, which is a potential bottleneck stated in previous work. The simulation configuration is set up as shown in table 4.1.

4.2.1 Validation

We validate our simulator by comparing the normalized running time of our simulator and real hardware. We choose AMD Kaveri A10-7850k APU as our comparing target. Figure 4.1 shows the correlation of normalized running time of BF, OF, RT, and SIFT, with 0.94, 0.98, 0.95, 0.87 correlation coefficient. Overall, the data shows that applications

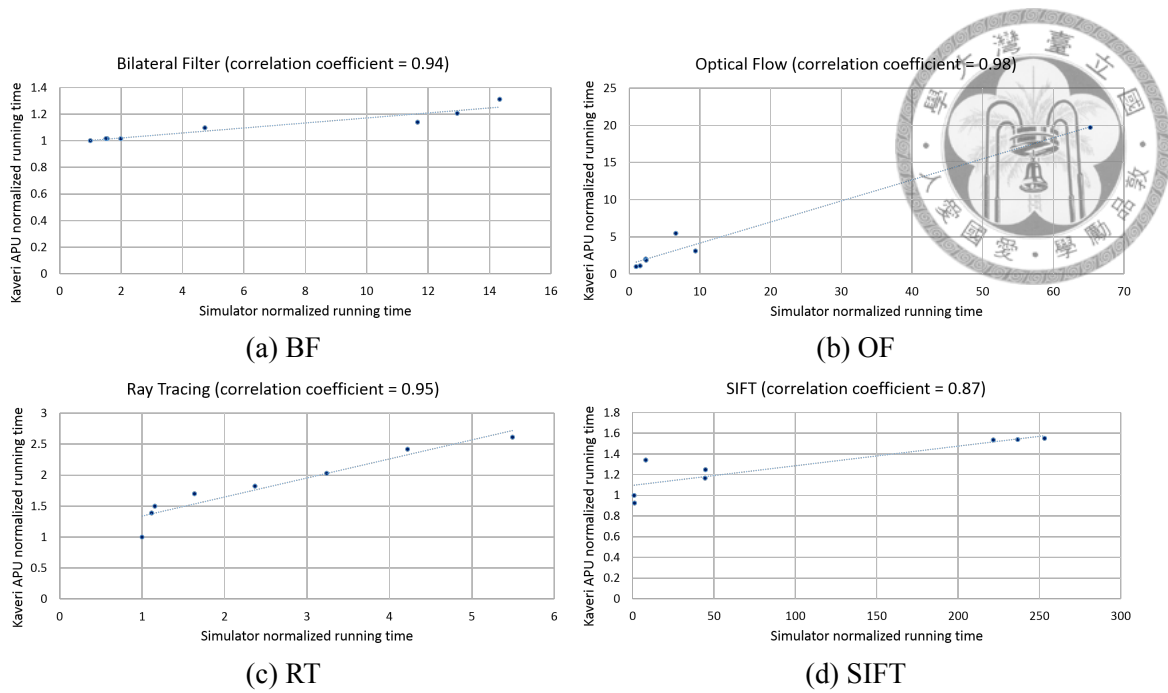


Figure 4.1: Real Hardware and Our Simulator's Normalized Running Time Correlation.

perform well in real hardware perform well in our simulator, and applications perform poorly in real hardware perform poorly in our simulator.

4.2.2 Shared Virtual Memory

As described earlier, SVM frees programmers from copying data between CPU and GPU back and forth. This is the most easy-to-use feature as all OpenCL 1.2 applications can be rewritten to use SVM by replacing all the buffer management API in the host code with SVM API, and it doesn't require any modification in the kernel code.

Figure 4.2 shows the speedup of four applications using SVM compared to the ones using OpenCL 1.2 explicit memory copy API. Among the four applications, only BF's OpenCL 2.0 kernel code has been modified to use dynamic parallelism, the other three use same kernel code as the OpenCL 1.2 version to run the experiment. The speedup for BF, OF, RT, and SIFT are 71.3%, 11.2%, 16.5%, and 4.6% respectively. Figure 4.3 breaks down the running time of applications into two parts: memory copy time and kernel execution time. As we can see, the kernel running time remains almost the same except for BF because of different kernel code used in its OpenCL 2.0 version, and the portion of memory copy time for BF, OF, RT, and SIFT are 40.0%, 10.2%, 14.3%, and 4.6%

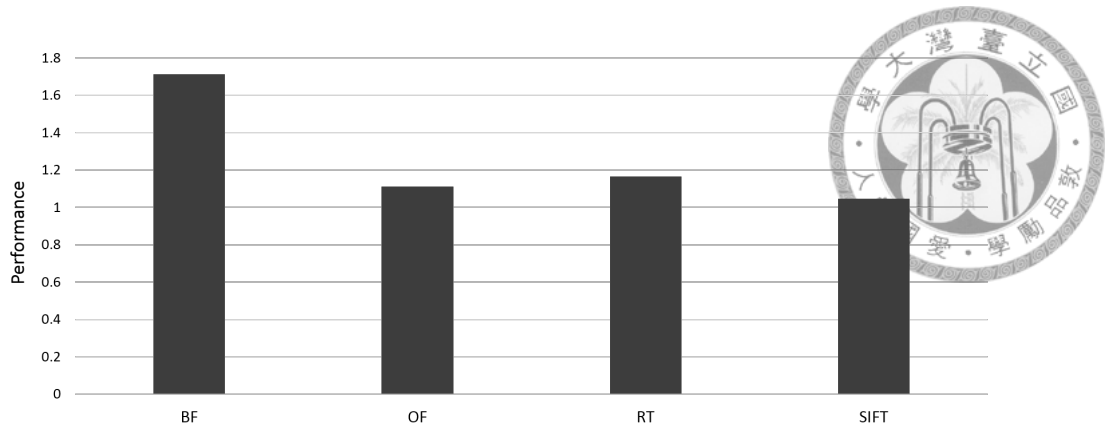


Figure 4.2: Performance of applications using SVM normalized to applications using memory copy

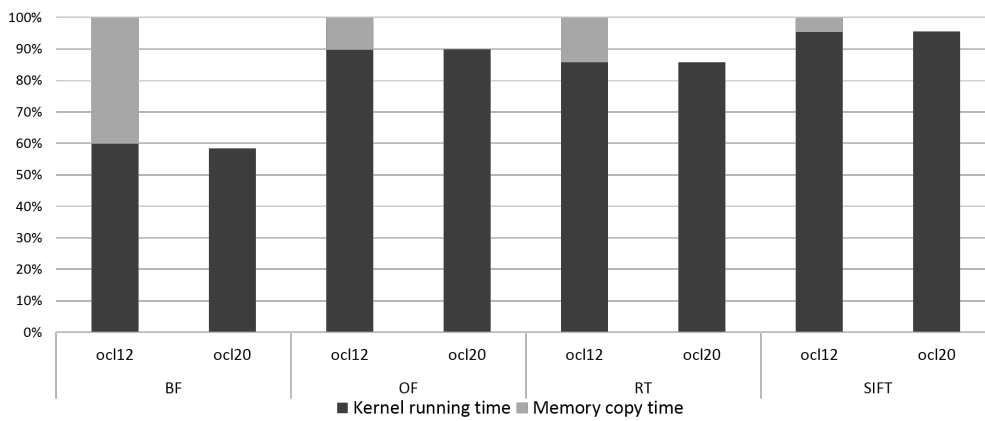



Figure 4.3: Running time breakdown for applications using SVM

respectively, which match the speedup shown in figure 4.2. To sum up, for our tested applications, SVM can improve the performance of traditional OpenCL 1.2 applications by eliminating their memory copy time, and the impact of replacing memory copy with SVM on kernel execution seems to be very little.

4.2.3 Dynamic Parallelism

Figure 4.4 shows the performance of applications using dynamic parallelism normalized to the ones not using dynamic parallelism. BF has 5.2% speedup and PRK has 19.5% degradation. The different impact on the two applications comes from their different dynamic parallelism usage. For PRK, this is a typical case of using dynamic parallelism to conquer nested parallelism: each thread will spawn a child kernel to process the task in



parallel. Wang et al. [33, 32] have done a thorough study on the deficiency of dynamic parallelism. PRK also has the same problems. First, on average, the number of threads in a child kernel spawned in PRK is only 160. The small child kernels cause SMs to be underutilized and thus degrade the performance. Second, the child kernels of PRK are very memory-intensive. The child kernel code contains six memory operation in only 3 lines of code. Memory stalls caused by such memory intensity is hard to be hidden when the number of threads in an SM is small. On the other hand, BF exhibits different usage of dynamic parallelism, which leads to its performance improvement. The OpenCL 1.2 version of BF is implemented as a producer-consumer program where the host will not launch consumer kernel until producer kernel has finished, as shown in figure 4.5a, but in fact, a thread in the consumer kernel only depends on a part of the producer kernel's results. In the algorithm level, it is more ideal for a part of the threads in consumer kernel to start once their data is ready rather than waiting for the whole data to be ready. The former has potential to have a part of producer and consumer kernel run together to increase the resource utilization, but it is the OpenCL 1.2 programming model that restricts the implementation to be the latter. The OpenCL 2.0 version of BF uses dynamic parallelism to achieve this. As shown in figure 4.5b, each thread of the producer kernel can launch a child kernel, which is a part of the job in the OpenCL 1.2 consumer kernel, when it's finished. Such implementation can have opportunity for the running time of producer and consumer to be overlapped, and thus increase the SM utilization and performance. A key factor to the overlapped time is the producer kernel's work-group running time divergence. If the running time divergence among work-groups is high, then there will be more chance to start a child kernel to run with the producer kernel in parallel. In addition to this overlapping effect, the number of threads in BF's child kernel is 1920, which is much larger than PRK's, so that the SM utilization is higher and the capability of hiding memory latency is better.

More recent GPU architecture, e.g. NVIDIA's Kepler and Maxwell [7, 6], has introduced a new hardware feature – concurrent kernel execution within an SM. When there are multiple kernels running on a GPU, this feature can achieve more fine-grained spatial

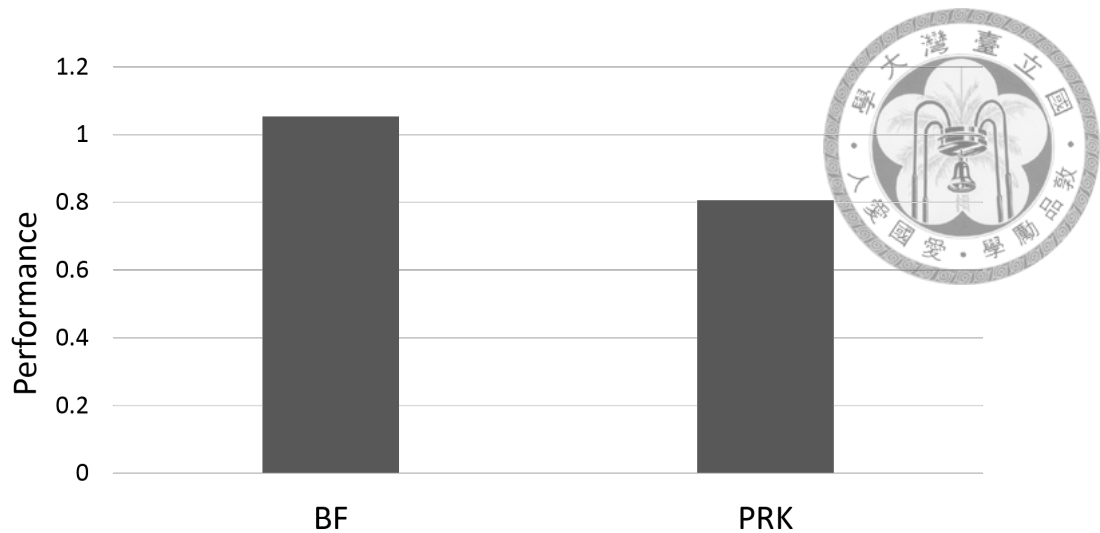
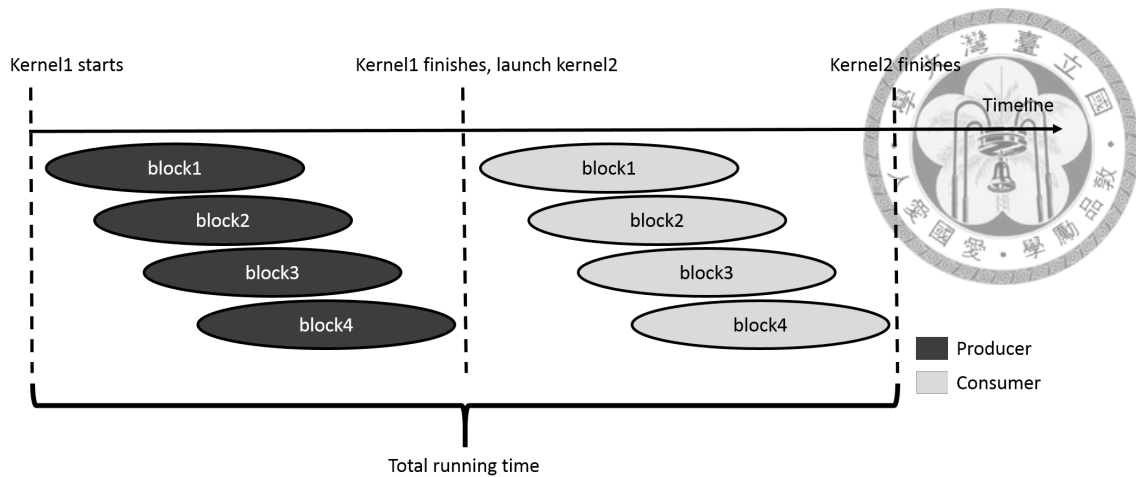


Figure 4.4: Performance of applications using dynamic parallelism normalized to OpenCL 1.2

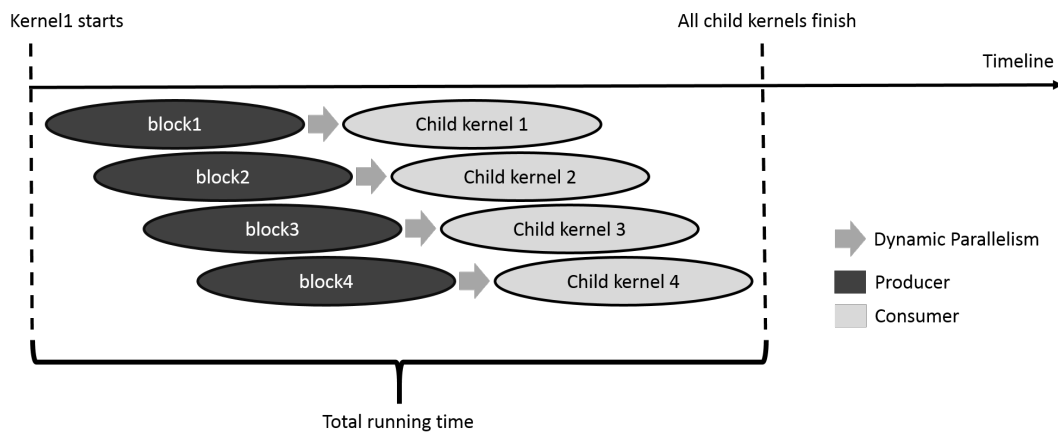
sharing and raise the SM utilization rate. For applications launching small child kernels like PRK, this feature can help alleviate the SM underutilization problem. Currently we haven't implemented this feature in the simulator. We'll leave it as our future work.

4.2.4 Work-Group Built-in Functions

In this section, we'll compare the performance of two applications using OpenCL 2.0 work-group built-in functions and using OpenCL 1.2 functionally equivalent implementation. The main difference between the OpenCL 2.0 and 1.2 implementation is that OpenCL 2.0 uses new PTX *warp shuffle* instruction while OpenCL 1.2 uses shared memory buffers and barriers to exchange data. We illustrate the difference further in detail in figure 4.6, where we use warp shuffle and shared memory to achieve data exchange, which will be frequently used by work-group built-in functions. Figure 4.6a shows a typical way in OpenCL 1.2 to perform data exchange. All threads write to a shared memory buffer and use barrier instruction to make sure all threads have done writing. Then each thread reads from that shared memory buffer to retrieve other thread's data. The whole procedure is very inefficient as threads have to synchronize using barrier and write redundant value to shared memory, but that's not the case in OpenCL 2.0. As shown in figure 4.6b, the warp shuffle instruction can let one thread to retrieve another thread's register value. This can



(a) OpenCL 1.2 producer-consumer workflow



(b) OpenCL 2.0 producer-consumer using dynamic parallelism

Figure 4.5: Bilateral filter's overlapping effect using dynamic parallelism

eliminate the redundant shared memory read / write compared to OpenCL 1.2, and thus improve the performance.

Figure 4.7 shows the performance of two applications normalized to OpenCL 1.2 shared memory implementation. BIS and RMQ have 23.1%, 12.8% improvement respectively. The reason why BIS has better performance improvement is because the implementation difference behind the work-group built-in functions they use. Figure 4.8a, 4.8b show the implementation difference behind work-group reduce min and work-group scan add, where threads in read are running. As we can see, when performing the operation, work-group reduce min will have less threads running compared to work-group scan add. This makes work-group scan add have more chances to use warp shuffle to improve the

performance compared to work-group reduce min. And because BIS uses work-group scan add, it shows better performance improvement compared to RMQ.



4.2.5 Platform Atomics

The platform atomic functions enable CPU and GPU processing on the same data, making heterogeneous computing one step further to embrace a broader range of applications. To support the true data sharing between CPU and GPU, some cache coherence mechanism is required. In previous work, Jason et al. [25] have stated that the massive thread accesses from GPU may overwhelm the directory for a traditional directory coherence protocol, which may be a potential bottleneck for heterogeneous system coherence. Their analysis also shows that the number of directory accesses from GPU can be more than one per GPU cycle, which is hard to support in terms of power and area overhead. However, their experiments are done on the traditional OpenCL-1.2-like applications where CPU launches tasks to GPU, and then waits for GPU to finish. Such applications' coherence needs only occur at the point of a kernel's beginning and end, which are predictable and easily to be optimized. In this section, we'll analyze two applications from AMD's APP SDK that exist the behavior of CPU-GPU co-working. Their coherence needs are not only at the point of kernel's beginning and end but throughout the whole running time of a kernel.

Figure 4.9 shows the directory accesses every 100 cycles. Both of the applications' running time have a similar trend. We divide them into two phases: the initial phase, which is the problem pointed out by previous work, existing burst access behavior and

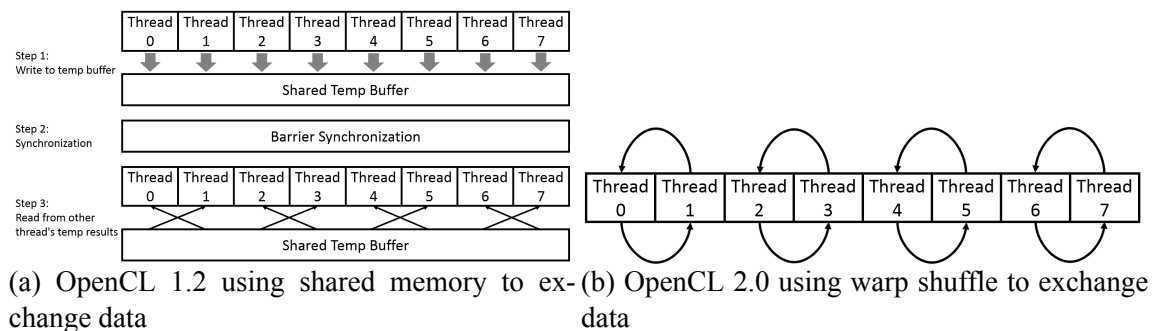


Figure 4.6: Demonstration of performance difference between OpenCL 1.2 and 2.0.

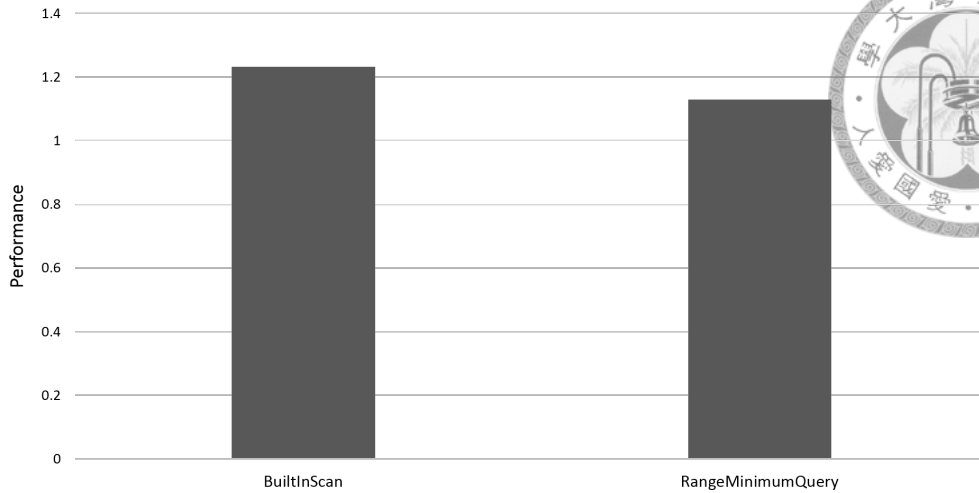
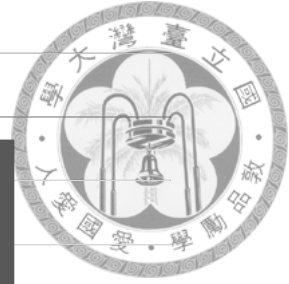
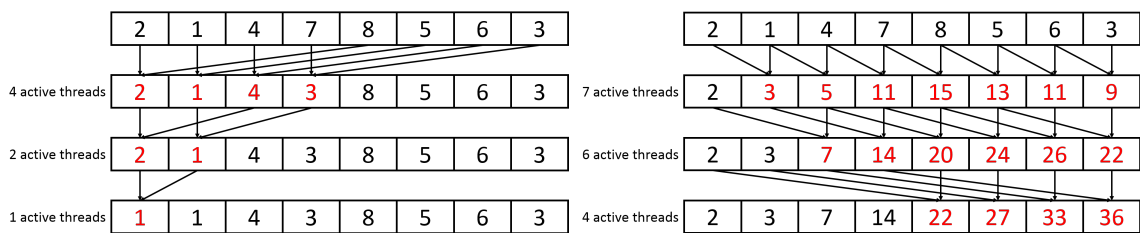


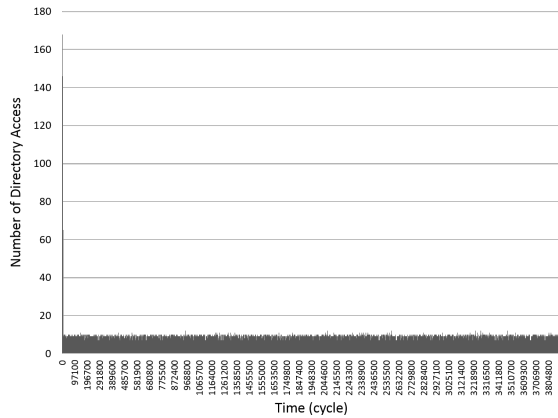
Figure 4.7: Performance of applications using work-group built-in functions normalized to OpenCL 1.2



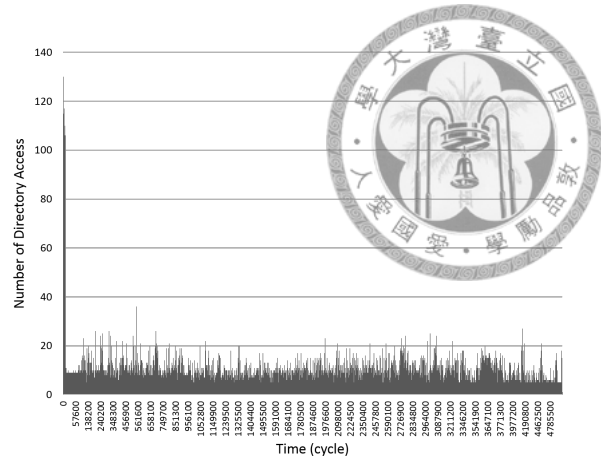
(a) Work-group reduce min implementation demonstration. (b) Work-group scan add implementation demonstration.

Figure 4.8: Implementation difference behind different work-group built-in functions.

lasting only for a very short time at the beginning of the kernel, and the execution phase, which includes the rest of the running time and has a more steady access behavior, is what we will focus on. As we can see, the directory access behavior during the execution phase is not as intensive as it is during the initial phase. In fact, figure 4.9 shows that the number of directory accesses in both of the applications never exceed more than one access per cycle, which indicates that the coherence request will not overwhelm the directory. The reason is due to the significant data processing speed difference between CPU and GPU. For GPU, it will generate lots of memory accesses in parallel, but for CPU, it can only access data in memory sequentially. As a result, even though there will be burst accesses to directory during the initial phase because of the compulsory miss in GPU L2 cache, during the execution phase, most of the accesses will hit in GPU L2 cache and only those cache lines touched by CPU will miss, which is unlikely to cause many directory accesses



(a) FGSC



(b) SABTI

Figure 4.9: Number of Directory Access Every 100 Cycles

as CPU's data processing speed is not fast enough.

Currently all the experiments are done with a single thread CPU program co-working with a GPU program. According to figure 4.9, the max number of directory accesses per cycle during the execution time is about 40. We believe that by running a multi-threaded program, this number can still scale up, so it is possible that the directory is a bottleneck for this type of applications. We'll leave it as our future work.



Chapter 5

Related Works

5.1 Heterogeneous Computing Simulator

Currently in the field of computer architecture research, the following are simulators that are widely used to do GPU and heterogeneous computing-related research. `gpgpu-sim` [10] is a cycle-level execution-driven GPU simulator that simulates NVIDIA's Fermi architecture GPU. It supports CUDA 4.0 and OpenCL 1.2, and provides a fake interceptor-like library. Once GPU calls the library, it will be trapped into the `gpgpu-sim` session, simulating execution of a real discrete GPU. `gpgpu-sim` does not simulate CPU-side execution, so it's unable to simulate a full system CPU-GPU environment. Wang et al. [32] have modified the source code of `gpgpu-sim` to make it simulate a part new hardware features in NVIDIA's Kepler architecture, including dynamic parallelism. `gem5-gpu` [26] is a detailed event-driven heterogeneous CPU-GPU simulator that integrates CPU simulator `gem5` and GPU simulator `gpgpu-sim` into having the same physical memory and address space. Currently `gem5-gpu` only supports CUDA 4.0. `Multi2Sim` [31] is an execution-driven heterogeneous CPU-GPU simulator that is able to simulate various GPU architectures, such as NVIDIA's Fermi and AMD's Southern Islands, and support both CUDA 4.0 and OpenCL 1.2 programming model. `MARSSx86-PTL-SIMT-GPU` [34] is a trace-driven CPU-GPU heterogeneous simulator, but due to its trace-driven nature, its simulation accuracy is not as realistic as `gem5-gpu` or `Multi2Sim`, thus is less used in academia. Both `Multi2Sim` and `MARSSx86-PTL-SIMT-GPU` only have separated CPU and GPU mem-

ory. Among all these simulators, only gem5-gpu integrates CPU and GPU into connecting to the same physical memory and sharing the same address space. Our work supports OpenCL 2.0 and is the first simulator that supports OpenCL 2.0.



5.2 Heterogeneous Workload Analysis

There are several GPGPU workload analysis study, and most of them are done by using CUDA workloads. gpgpu-sim [10], Rodinia [14], and Parboil [29] have done general analysis using their own GPGPU workloads. Most of the benchmarks they use are regular; only a few are graph or tree-based algorithms, like *breadth-first-search* from Parboil and Rodinia and *B+tree* from Rodinia. Pannotia [13] features more on quantitative study of irregular graph processing GPGPU applications, for instance, graph coloring and friend recommendation, and the benchmark suite is written in OpenCL 1.2. The analysis of irregular applications present significant challenge on GPGPU architectures such as branch and memory accessing divergence, and input-dependent load imbalance and parallelism. Burtscher et al. [12] also perform a quantitative study of irregular GPGPU applications. Different from Pannotia, they use more types of irregular applications than graph applications, such as data compression, pointer-to analysis. In our work, we use benchmarks collecting from the above, and also benchmarks that run computer vision-related workloads, which are not included in any of the benchmark suite mentioned above. We believe that by conducting experiments on such diversity of workloads can represent characteristics in different perspective and bring out the full potential of heterogeneous computing.

Wang et al. [33] studies a certain type of irregular GPGPU applications, called nested parallelism applications. These applications have to use loop iteration in each thread to tackle different size of input data for it, causing workload imbalance and some cache problems. They rewrite these applications with dynamic parallelism, and compares their performance difference. They discover that although the workload imbalance and cache problems can be mitigated by launching child kernels to process in parallel, they will suffer from serious SM under-utilization problem because the version that uses dynamic parallelism tends to launch child kernels with few threads. DTBL [32] fixes the problem

by merging the launched child kernels together to have a larger aggregated child kernel.

Jason et al. [25] have done a quantitative study of cache coherence behavior on Rodinia and AMD's APP SDK and conclude that the hardware directory can be the major bottleneck for a heterogeneous system that enables cache coherence between CPU and GPU because of the massive memory requests sent by GPU. They apply a more coarse-grained cache coherence protocol, called region coherence, to tackle this problem because they claim that most of the GPGPU applications have good spatial locality. By exploiting such spatial locality, their work can effectively reduce times that GPU sends coherence requests to the hardware directory.

Currently there are only one work that focuses on OpenCL 2.0 applications analysis. Saoni et al. [22] analyze OpenCL 2.0 applications' performance by using AMD's Kaveri APU, but this work focuses more on real machine's driver overhead and performance difference rather than applications' behavior using new features in OpenCL 2.0. Also they release a heterogeneous benchmarks called *Hetero-Mark*, which is written 3 different versions, including OpenCL 1.2, OpenCL 2.0, HSA 1.0.



Chapter 6

Conclusion

In this thesis, we extend an existing CPU-GPU integrated simulator, `gem5-gpu`, to support OpenCL 2.0 and do a simple analysis on a set of OpenCL 2.0 applications. To make `gem5-gpu` support the four main features in OpenCL 2.0, which are shared virtual memory, dynamic parallelism, work-group built-in functions, and enhanced atomic functions, we add up the OpenCL host API based on the original CUDA host API, modify GPU side simulation's internal structure to support dynamic parallelism and new PTX instructions, and extend LLVM's Clang compiler's PTX backend to support OpenCL 2.0 kernel to PTX code compilation. The analysis has pointed out some behaviors existing in OpenCL 2.0 applications. For SVM, it can effectively reduce the time of memory copy, especially for applications that execute short kernels. For dynamic parallelism, programmers must be aware of the child kernel's thread count so that GPU's utilization can be maximized. For work-group built-in functions, we've shown that using the new PTX warp shuffle instruction, the instructions executed by a kernel can be reduced thus improving the performance. And for the new platform atomic functions, we've shown that coherence traffic as a potential bottleneck stated in previous work may not be a problem when using one CPU thread to co-work with GPU.

In future work we plan to extend current simulator to support more GPU architectural features and optimization features in OpenCL 2.0, such as concurrent kernel execution within an SM and optimizing atomic functions specified with a scope. We also plan to support the latest version of OpenCL, as Khronos released OpenCL 2.1 last year and re-

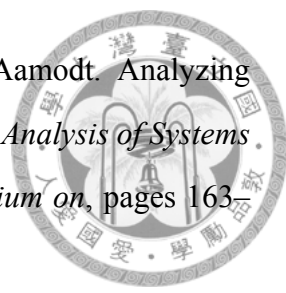
cently OpenCL 2.2 is just heading in the provisional stage.







Bibliography

- [1] Amd's app sdk download page. <http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/>.
- [2] clang: a c language family frontend for llvm. <http://clang.llvm.org/>.
- [3] Hsa foundation. <http://www.hsafoundation.com/>.
- [4] Khronos group. <https://www.khronos.org/>.
- [5] Nvidia fermi architecture whitepaper. http://www.nvidia.com.tw/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.
- [6] Nvidia geforce gtx 980 whitepaper. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF.
- [7] Nvidia kepler gk110 architecture whitepaper. <http://www.nvidia.com.tw/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>.
- [8] The opencl 2.0 specification. <https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf>.
- [9] A. Ashari, S. Tatikonda, M. Boehm, B. Reinwald, K. Campbell, J. Keenleyside, and P. Sadayappan. On optimizing machine learning workloads via kernel fusion. *SIGPLAN Not.*, 50(8):173–182, jan 2015.

- 
- [10] A. Bakhoda, G. L. Yuan, W. W. L. 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*, pages 163–174, April 2009.
- [11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.
- [12] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151, Nov 2012.
- [13] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. Pannotia: Understanding irregular gpgpu graph applications. *2013 IEEE International Symposium on Workload Characterization (IISWC)*, 2013.
- [14] 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 *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.
- [15] B. Gaster. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, 2012.
- [16] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 2:1–2:12, Piscataway, NJ, USA, 2008. IEEE Press.
- [17] D. Kaeli, P. Mistry, D. Schaa, and D. Zhang. *Heterogeneous Computing with OpenCL 2.0*. Elsevier Science, 2015.
- [18] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Gen-*

eration and Optimization: Feedback-directed and Runtime Optimization, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

- 
- [19] D. Lowe. Method and apparatus for identifying scale invariant features in an image and use of same for locating an object in an image, Mar. 23 2004. US Patent 6,711,293.
- [20] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, Nov. 2004.
- [21] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81, pages 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.
- [22] S. Mukherjee, Y. Sun, P. Blinzer, A. K. Ziabari, and D. Kaeli. A comprehensive performance analysis of hsa and opencl 2.0. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 183–193, April 2016.
- [23] C. H. Nadungodage, Y. Xia, J. J. Lee, M. Lee, and C. S. Park. Gpu accelerated item-based collaborative filtering for big-data applications. In *Big Data, 2013 IEEE International Conference on*, pages 175–180, Oct 2013.
- [24] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.
- [25] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous system coherence for integrated cpu-gpu systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 457–467, New York, NY, USA, 2013. ACM.

- 
- [26] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, Jan. 2015.
- [27] P. Shirley and R. Morley. *Realistic Ray Tracing, Second Edition*. Ak Peters Series. Taylor & Francis, 2003.
- [28] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.
- [29] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.
- [30] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Proceedings of the Sixth International Conference on Computer Vision, ICCV '98*, pages 839–, Washington, DC, USA, 1998. IEEE Computer Society.
- [31] 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, PACT '12*, pages 335–344, New York, NY, USA, 2012. ACM.
- [32] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili. Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on gpus. *SIGARCH Comput. Archit. News*, 43(3):528–540, June 2015.
- [33] J. Wang and S. Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured gpu applications. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 51–60. IEEE, 2014.
- [34] P. H. Wang, G. H. Liu, J. C. Yeh, T. M. Chen, H. Y. Huang, C. L. Yang, S. L. Liu, and J. Greensky. Full system simulation framework for integrated cpu/gpu architecture.

In *VLSI Design, Automation and Test (VLSI-DAT), 2014 International Symposium on*, pages 1–4, April 2014.

[35] Q. Yang. Recursive bilateral filtering. In *Proceedings of the 12th European Conference on Computer Vision - Volume Part I, ECCV'12*, pages 399–413, Berlin, Heidelberg, 2012. Springer-Verlag.

[36] J. yves Bouguet. Pyramidal implementation of the lucas kanade feature tracker. *Intel Corporation, Microprocessor Research Labs*, 2000.

