The Distribution of OpenCL Kernel Execution Across Multiple Devices

by

Steven Gurfinkel

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of the Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2014 by Steven Gurfinkel
Abstract

The Distribution of OpenCL Kernel Execution Across Multiple Devices

Steven Gurfinkel
Master of Applied Science
Graduate Department of the Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
2014

Many computer systems now include both CPUs and programmable GPUs. OpenCL, a new programming framework, can program individual CPUs or GPUs; however, distributing a problem across multiple devices is more difficult. This thesis contributes three OpenCL runtimes that automatically distribute a problem across multiple devices: DualCL and m2sOpenCL, which distribute tasks across a single system’s CPU and GPU, and DistCL, which distributes tasks across a cluster’s GPUs. DualCL and DistCL run on existing hardware, m2sOpenCL runs in simulation. On a system with a discrete GPU and a system with integrated CPU and GPU devices, running programs from the Rodinia benchmark suite, DualCL improves performance over a single device, when host memory is used. Running similar benchmarks, m2sOpenCL shows that reducing the overheads present in current systems improves performance. DistCL accelerates unmodified compute intense OpenCL kernels, obtained from Rodinia, AMD samples and elsewhere, when distributing them across a cluster.
Acknowledgements

I must start by thanking my supervisors, professors Natalie Enright Jerger and Jason H. Anderson. Without their guidance, care, commitment, and knowledge this work would not have been possible. They always came through for me when I needed some last-minute help and for that I am particularly grateful. It was an honour to be mentored by such great researchers. I must also thank my fellow graduate student Tahir Diop. Our close collaboration was both fun and productive. Furthermore, I thank my committee, professors Tarek S. Abdelrahman, Paul Chow, and Wai Tung Ng for their valuable insights and feedback.

Of course, I also thank my family and friends for their support throughout my degree. I thank both my parents for starting me on the path to science and engineering, Their academic achievements serve as an inspiration for my own.
# Contents

1 Introduction  
   1.1 Contributions ................................................. 1  
   1.2 Outline ........................................................... 2  

2 Background  
   2.1 OpenCL ............................................................ 3  
      2.1.1 Device Model ............................................... 3  
      2.1.2 Memory Coherence and Consistency .......................... 6  
      2.1.3 Host Model ................................................... 6  
   2.2 Other Heterogeneous Programming Models ......................... 8  
      2.2.1 CUDA ......................................................... 8  
      2.2.2 BOLT ........................................................... 8  
   2.3 The AMD Fusion APU ............................................. 8  
   2.4 Multi2Sim .......................................................... 8  
   2.5 Related Work ...................................................... 9  
      2.5.1 Distribution with Custom Interfaces ....................... 9  
      2.5.2 Distribution of OpenCL Kernels ............................ 10  
      2.5.3 Making Remote Devices Appear Local ....................... 10  
      2.5.4 Characterizing and Simulating Heterogeneous Execution .... 11  

3 Co-operative OpenCL Execution ...................................... 12  
   3.1 Outline ............................................................ 13  
   3.2 Dividing Kernel Execution ...................................... 13  
   3.3 DualCL ............................................................. 14  
      3.3.1 Memory Allocation .......................................... 15  
      3.3.2 Partitioning Kernels ....................................... 15  
   3.4 m2sOpenCL ........................................................ 15  
   3.5 Partitioning Algorithms ......................................... 17  
      3.5.1 Static Partitioning .......................................... 17  
      3.5.2 Block by Block (block) .................................... 18  
      3.5.3 NUMA (numa) .................................................. 18  
      3.5.4 First-Past-The-Post (first) ................................. 18  
      3.5.5 Same Latency (same) ........................................ 18  
   3.6 Experimental Study ................................................. 19
3.6.1 Benchmarks .................................................. 20
3.7 Results and Discussion ....................................... 22
3.7.1 Memory Placement .......................................... 22
3.7.2 Partitioning Algorithms ................................... 23
3.7.3 Individual Benchmarks ...................................... 26
3.7.4 Simulated Results .......................................... 34
3.8 Conclusions ..................................................... 36

4 CPU Execution for Multi2Sim .................................... 38
4.1 Overall Design .................................................. 38
4.1.1 The OpenCL Device Model for CPUs .................... 39
4.2 Behaviour of a Work-Item ..................................... 39
4.2.1 Work-Item Stack Layout .................................. 39
4.3 Fibres ............................................................. 39
4.4 Scheduling and Allocating Work-Items ..................... 41
4.4.1 Stack Allocation Optimization ............................ 42
4.5 Whole-Kernel Execution and Scheduling .................. 43
4.6 Performance Validation ........................................ 43
4.7 Conclusion ....................................................... 45

5 Distributed OpenCL Execution .................................. 46
5.1 Introduction ..................................................... 46
5.2 DistCL .......................................................... 47
5.2.1 Partitioning .................................................. 48
5.2.2 Dependencies ............................................... 48
5.2.3 Scheduling Work ............................................ 51
5.2.4 Transferring Buffers ....................................... 51
5.3 Experimental Setup .......................................... 52
5.3.1 Cluster ....................................................... 53
5.4 Results and Discussion ........................................ 54
5.5 Conclusion ....................................................... 58

6 Conclusion ........................................................ 60
6.1 Future Work ..................................................... 61

Appendices .......................................................... 62
A The Accuracy of the Multi2Sim CPU Device ................. 63
A.1 Microbenchmarks .............................................. 63
A.1.1 Register ...................................................... 63
A.1.2 Dependent Add .............................................. 63
A.1.3 Independent Add ............................................ 64
A.1.4 Memory Throughput ....................................... 64
A.1.5 Memory Latency ............................................ 65
A.2 Issues and Best Configuration ............................... 65
List of Tables

2.1 OpenCL Work-Item Functions .............................................. 6
2.2 Zero-Copy Memory Creation Flags ....................................... 7

3.1 Contributions to Multi2Sim’s Co-operative Execution ..................... 13
3.2 Parameters Added to Kernel Functions by DualCL .......................... 16
3.3 Substitutions Performed by DualCL ....................................... 16
3.4 Test Systems ................................................................. 20

4.1 Work-Item Information Data-Structure. .................................... 40
4.2 Test Configuration for OpenCL CPU Driver Comparison .................. 43

5.1 Benchmark description ....................................................... 53
5.2 Cluster specifications ........................................................ 54
5.3 Measured cluster performance .............................................. 54
5.4 Execution time spent managing dependencies .............................. 59

A.1 Results from Multi2Sim Configuration .................................... 65
# List of Figures

2.1 Two Dimensional NDRange with work-items and work-groups .................................. 4  
2.2 OpenCL Model of Compute Device ............................................................................. 5  

3.1 Co-operative and Regular Kernel Execution ............................................................... 14  
3.2 Dividing an NDRange into Smaller NDRanges ............................................................ 16  
3.3 Structure of Multi2Sim OpenCL ............................................................................... 17  
3.4 Different Device and Memory Combinations on Fusion ............................................ 23  
3.5 Different Device and Memory Combinations on Discrete System ............................... 24  
3.6 Partition Algorithms on Fusion System ..................................................................... 25  
3.7 Partition Algorithms on Discrete System .................................................................. 25  
3.8 Proportion of Work Done on CPU ............................................................................. 26  
3.9 CPU Kernel Invocation Overhead ............................................................................. 27  
3.10 GPU Kernel Invocation Overhead ............................................................................ 27  
3.11 DualCL Overhead .................................................................................................... 28  
3.12 Kmeans on Fusion System ...................................................................................... 29  
3.13 Kmeans on Discrete System .................................................................................... 29  
3.14 Nearest Neighbour on Fusion System ...................................................................... 30  
3.15 Nearest Neighbour on Discrete System .................................................................... 30  
3.16 Pathfinder on Fusion System ................................................................................... 31  
3.17 Pathfinder on Discrete System ................................................................................... 32  
3.18 Back Propagation on Fusion System ........................................................................ 32  
3.19 First Kernel of Back Propagation on Fusion System ............................................... 33  
3.20 Second Kernel of Back Propagation on Fusion System ............................................ 33  
3.21 Gaussian in Simulation ............................................................................................ 34  
3.22 Smaller Gaussian in Simulation .............................................................................. 35  
3.23 Back Propagation in Simulation .............................................................................. 36  
3.24 Nearest Neighbour in Simulation ........................................................................... 37  

4.1 Stack of a Work-item ................................................................................................. 41  
4.2 Work-group Execution With and Without Barriers ................................................... 42  
4.3 Runtime of Our Runtime and AMD Runtime with Synthetic Benchmarks ............... 44  
4.4 Ratio of Our Runtime vs AMD Runtime for APPSDK Benchmarks ........................... 45  

5.1 Vector’s 1-dimensional NDRange is partitioned into 4 subranges. ............................... 48  
5.2 The read meta-function is called for buffer a in subrange 1 of vector. ......................... 51
5.3 Speedup of distributed benchmarks using DistCL .......................... 55
5.4 DistCL and SnuCL speedups ...................................................... 56
5.5 Breakdown of runtime ............................................................... 57
5.6 HotSpot with various pyramid heights ........................................ 58
Chapter 1

Introduction

Heterogeneous devices have moved into the mainstream. This shift occurred because such devices offer improved performance and energy efficiency [30] over homogeneous platforms. Recently, Intel [35] and AMD [3] have released heterogeneous processors that contain both a CPU and GPU on the same chip. Like their discrete counterparts, the CPUs and GPUs on heterogeneous chips can be programmed using OpenCL [21], an industry-standard API and programming model used for running computations on modern multi-core devices. For certain algorithms, GPUs can provide significant speedups [6] [38], when compared to CPUs.

OpenCL allows a programmer to harness the parallelism provided by a single device; however, despite OpenCL’s data-parallel nature, it does not directly support distributing a task across multiple devices. In fact, AMD’s programming guide [3] brings up several issues, including synchronization and partitioning, that programmers must tackle themselves in order to write a program that efficiently executes across multiple devices. Additionally, most algorithms are simpler to write with only one device in mind, and require extra care to run properly on multiple devices. These issues make distributed OpenCL execution difficult. To address these issues, this work introduces three OpenCL frameworks that distribute the execution of an OpenCL kernel across multiple devices.

1.1 Contributions

This thesis contributes two novel OpenCL implementations that distribute execution across two devices that can access the same memory. The first, DualCL, runs on existing hardware and takes advantage of the memory-sharing capabilities present on current heterogeneous systems. DualCL makes the CPU and GPU appear as a single device by distributing OpenCL kernels between those devices automatically, without any programmer intervention. DualCL runs on systems with discrete graphics cards as well as systems that contain the AMD Fusion APU [10], a processor that contains both a CPU and GPU. Thus, DualCL is able to measure what benefit the physical integration of the Fusion APU yields for executing kernels across both devices. The results generated with DualCL suggest that both single-chip and discrete hardware suffer from memory placement tradeoffs and high GPU invocation overheads that make efficient, automatic, distributed execution difficult.

The second implementation, m2sOpenCL, runs in a simulator called Multi2Sim [43], taking advantage of the simulated hardware’s new cache coherence protocol and tightly-integrated CPU and GPU devices.
Chapter 1. Introduction

For this work, we used Multi2Sim to simulate an x86 CPU and a ‘Southern Islands’-architecture GPU. M2sOpenCL is Multi2Sim’s built-in OpenCL implementation and allows for the execution of OpenCL kernels on individual devices, as well as on a virtual device that represents the combination of the Southern Islands and x86 devices. M2sOpenCL was written in collaboration with Multi2Sim’s developers.

To execute an OpenCL program across CPU and GPU devices, both DualCL and m2sOpenCL must use partitioning algorithms that determine how to divide the data-parallel task and schedule it across the devices. Using DualCL, we introduce and evaluate four partitioning algorithms that dynamically divide a kernel without any a-priori knowledge. To examine how tighter CPU and GPU integration can enhance partitioning algorithms, we run the same partitioning algorithms on m2sOpenCL as well. The results generated with Multi2Sim suggest that tighter levels of integration reduce runtime.

A third implementation, DistCL, distributes work between multiple GPUs that are connected together in a cluster. With clusters, each device has its own memory, and data must be transferred over a network to move from GPU to GPU. DistCL tracks data placement and initiates network operations automatically, freeing the programmer from these tasks. To determine the memory access patterns of the kernels it runs, DistCL includes a novel mechanism, called meta-functions, which allow the programmer to represent the memory access patterns of OpenCL kernels. Meta-functions are short C functions that accompany an OpenCL kernel, and relate the index of a thread to the addresses of memory it uses. With meta-functions, DistCL can determine how to place memory so that the threads assigned to a particular GPU have the data they need to run.

1.2 Outline

This thesis is divided into six chapters. After this chapter, comes Chapter 2, background, which describes the portions of OpenCL relevant to this thesis, and discusses related work. Then in Chapter 3 DualCL and m2sOpenCL are discussed, and results regarding shared-memory OpenCL kernel distribution are presented. This thesis also describes the design and validation of an x86 CPU driver in Chapter 4, a critical component of m2sOpenCL. Next, in Chapter 5 DistCL is discussed, and results for distributed memory OpenCL kernels are presented. Finally, Chapter 6 concludes the thesis. There are also two appendices to this thesis. Appendix A describes our attempts to validate the performance of Multi2Sim’s x86 CPU. Appendix B is a reproduction of the manual we included in our release of DistCL.
Chapter 2

Background

This thesis uses OpenCL for heterogeneous computing. This chapter looks at the OpenCL model and how it applies to the works described in this thesis. It also looks at other commercial and academic methods that perform heterogeneous computing across one and many devices.

2.1 OpenCL

OpenCL is a popular framework used for programming multi-core processors such as CPUs and GPUs. Typically, a programmer will use OpenCL to execute part of a program on a GPU. To do this, the programmer writes a function called a kernel in a subset of C99 (the version of the C programming language published in 1999) that includes extensions for parallelism. This kernel is compiled at runtime for the devices the programmer wishes to target. The programmer also writes a conventional program that makes calls to an OpenCL runtime. The runtime allows the program to submit the kernel to a compute device (such as a GPU) on the computer. A program that interacts with OpenCL is called a host program because it runs on the host computer. When the host program launches a kernel on the compute device (such as a GPU), it typically requests the creation of thousands of threads. Each thread runs its own instance of the kernel function, but has an ID unique within the kernel invocation. The way these threads are organized and executed is part of the OpenCL standard. We will refer to this organization as the device model.

To pass data between the host and devices, OpenCL uses the concept of buffers. The host program allocates, writes to, and reads from buffers using OpenCL calls. This allows the program to transfer data to and from kernels, which execute on the devices. Host programs cannot necessarily directly access device memory, because many compute devices have their own memory systems. Buffers allow OpenCL to marshal data between the host and devices.

2.1.1 Device Model

The threads of an OpenCL kernel invocation execute in an abstract grid called an NDRrange. An NDRrange can be one-dimensional (linear), two-dimensional (rectangular), or three-dimensional (rectangular prism-shaped). An n-dimensional NDRrange is specified by n positive integers, each being the NDRrange’s size in one dimension. The size and dimensionality of the NDRrange is decided by the host program when it enqueues the kernel to a device. At the finest granularity, this NDRrange contains
unit-sized work-items. These work-items are the threads of an OpenCL kernel invocation and each run their own instance of the kernel function. Each work-item has a unique \( n \)-dimensional global ID which are its coordinates in the NDRange. Similarly, the size of the NDRange is the invocation’s global size. The NDRange is also divided up into equally-sized \( n \)-dimensional regions called work-groups. Each work-item belongs to exactly one work-group, and has an \( n \)-dimensional local ID based on its position in the work-group. Similarly, the size of the work-groups is the invocation’s local size. Figure 2.1 shows an example of a two-dimensional NDRange with \( N \times M \) work-groups and \( n \times m \) work-items per work-group.

**Device Structure**

Each core of a compute device is called a compute-unit. Particularly in GPUs, compute-units contain multiple processing-elements where a processing element executes a single thread. Figure 2.2 shows OpenCL’s model for a compute device.

The compute-units of a compute-device are assigned work at the work-group granularity and compute-
units execute work-groups one at a time. Work-groups are further divided into fixed-sized collections of work-items called *wavefronts*. At any one time, only work-items from a single wavefront will be executing on the processing elements of the compute unit. Thus, the wavefront size is always a multiple of the number of processing elements per compute unit. GPU hardware interleaves the execution of work-items on processing elements to hide memory access latency. For instance, the GPU hardware used in this thesis interleaves the execution of 4 work-items on the same processing element. Despite this hardware interleaving, from a programming point of view, all the work-items of a wavefront are executing simultaneously, and a compute unit executes a work-group by interleaving the execution of its wavefronts.

**Work-Item Functions**

Work-items can retrieve information about the geometry of the kernel invocation they are a part of (the geometry of the NDRange), and their particular place in it. Table 2.1 shows the work-item functions that OpenCL supports, and what they return. Most functions in this table have a parameter - the dimension index. As NDRanges may have up to three dimensions, the value of this parameter (either 0, 1, or 2) selects the dimension to consider. Work-items can use the values returned by those functions to coordinate how they perform computations.
Table 2.1: OpenCL Work-Item Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint get_work_dim()</code></td>
<td>Number of dimensions in the NDRange</td>
</tr>
<tr>
<td><code>size_t get_global_size(uint dimidx)</code></td>
<td>Global size in a particular dimension</td>
</tr>
<tr>
<td><code>size_t get_global_id(uint dimidx)</code></td>
<td>Position in the NDRange in a particular dimension</td>
</tr>
<tr>
<td><code>size_t get_local_size(uint dimidx)</code></td>
<td>Local size in a particular dimension</td>
</tr>
<tr>
<td><code>size_t get_local_id(uint dimidx)</code></td>
<td>Position in the work-group in a particular dimension</td>
</tr>
<tr>
<td><code>size_t get_num_groups(uint dimidx)</code></td>
<td>Number of groups in the NDRange in a particular dimension</td>
</tr>
<tr>
<td><code>size_t get_group_id(uint dimidx)</code></td>
<td>Position of the group in a particular dimension</td>
</tr>
<tr>
<td><code>size_t get_global_offset(uint dimidx)</code></td>
<td>Value of an offset added to the global IDs in a particular dimension</td>
</tr>
</tbody>
</table>

Types of Memory

OpenCL supports four types of device memory: private memory, local memory, global memory and constant memory. Private memory is private to an individual work-item, and generally consists of the work-item’s copy of the parameters and any automatic variables. Similarly, each work-group has its own instance of local memory. Local memory can be allocated at runtime and passed to the kernel as a parameter, or allocated statically at compile-time. A separate instance of local memory exists for each running work-group; it is impossible for a work-item in one work-group to access the local memory of another work-group. On GPUs, local and private data are typically located in small, fast on-chip memories. Global memory is accessible by all work-groups. It is the only mutable memory that persists between kernel invocations and hence, is the only memory that can be read from and written to by the host. The host allocates regions of global memory as buffers, and passes them to kernels as parameters. Constant memory is like global memory, but can be cached more aggressively and can only be modified from the host.

2.1.2 Memory Coherence and Consistency

OpenCL does not require coherence between work-groups. In fact, devices can schedule work-groups in any order, so a work-group is not guaranteed to see the effects of another work-group’s execution. In contrast, the work-items within a work-group are guaranteed to be able to synchronize their execution, using the `barrier()` function, which causes a work-group wide barrier that all work-items must reach before any go past. Intra work-group synchronization is possible because a work-group executes on a single compute-unit. The lack of coherence between work-groups of the same kernel invocation is the main enabling factor that allows for distribution of a kernel invocation between multiple devices. Hence, when a kernel invocation is distributed, the distribution occurs at the work-group level of granularity.

2.1.3 Host Model

The host program, which runs in a conventional computing environment, interacts with OpenCL devices through a host interface comprised of C functions with names that start with `cl`. This interface is provided by a library which we refer to as an OpenCL Runtime or OpenCL Implementation. Different
Table 2.2: Zero-Copy Memory Creation Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_MEM_ALLOC_HOST_PTR</td>
<td>Place memory on host.</td>
</tr>
<tr>
<td>CL_MEM_USE_PERSISTENT_MEM_AMD</td>
<td>Place memory on device.</td>
</tr>
<tr>
<td>CL_MEM_USE_HOST_PTR</td>
<td>Use pre-existing memory (host passes pointer to this memory as a parameter to clCreateBuffer).</td>
</tr>
</tbody>
</table>

OpenCL implementations may expose devices with differing capabilities, but they are all accessible through the same host interface. This interface allows the host program to enumerate compute devices, place data onto these devices, and execute kernels on the devices.

The host interface returns objects such as devices, contexts, programs, kernels, command queues, and buffers to the host program. These objects allow the host program to manipulate OpenCL resources. Device objects represent an OpenCL device, like a CPU or GPU. Contexts are organizational objects. Objects (like devices, kernels, and buffers) that belong to the same context can interact, whereas objects from different contexts cannot. Programs contain OpenCL source code which can be compiled for various devices. Kernel objects are compiled OpenCL kernels, acquired from compiled programs. Kernels can have their parameters set, and they can be scheduled onto devices through command queues. Buffers represent regions of device-accessible memory. They can be passed to kernels as parameters, and can be read from and written to by the host program.

Buffers

To organize device memory, OpenCL uses the concept of buffers. These buffers appear to OpenCL kernels as arrays that can be accessed through C pointers. With OpenCL, the host program can access buffers two ways. The first way uses `clEnqueueReadBuffer` and `clEnqueueWriteBuffer`. These functions allow the host program to read or write a subset of an OpenCL buffer by transferring data between that part of the buffer and a normal host-allocated array. Second, OpenCL also allows the host program to ‘map’ a subset of a device-resident buffer into the host’s address space, and ‘unmap’ the buffer when the host is done accessing it. This method uses the `clEnqueueMapBuffer` and `clEnqueueUnmapMemObject` functions.

By default, buffers are allocated on the same device that accesses them; however, the host program can control the placement of buffers with the flags it passes to `clCreateBuffer`. The buffer can be placed either in host memory or on the device. If the CPU is the device, there is no difference between host and device memory; however, GPUs typically have their own memory, separate from the host’s. The placement of memory is important because it affects the performance with which it can be accessed by the CPU and GPU.

When the host reads buffers through mapping, the OpenCL implementation can make the buffer available in two ways. The implementation can either copy the device memory into the host program’s address space, or it can expose the memory to the host for direct access. This second approach is called ‘zero-copy’ access, because it does not involve copies. In order to enable zero-copy buffers for the GPU, the host program must pass flags into the `clCreateBuffer` call used to create OpenCL buffers. Table 2.2 shows the flags used to create zero-copy buffers. Depending on the flag used, the zero-copy buffer will be placed either in host or device memory. On systems with discrete GPUs, CPU access to
device memory access happens across the PCIe bus. Zero-copy memory is critical for fused execution because DualCL requires that both devices access the same physical memory during kernel execution.

### 2.2 Other Heterogeneous Programming Models

Besides OpenCL, there are other programming models for targeting GPU hardware.

#### 2.2.1 CUDA

Compute Unified Device Architecture (CUDA) [32], which pre-dates OpenCL, is a programming model developed by NVIDIA for programming their GPUs. CUDA follows a programming model very similar to that of OpenCL, with an essentially identical device programming model.

CUDA and OpenCL differ the most in their host interfaces. Whereas CUDA and OpenCL both have a standard C interface that can be linked to by any compiler, CUDA also has an additional compiler that supports a more convenient way of launching kernels with a special syntax.

#### 2.2.2 BOLT

BOLT [4] is a library released by AMD for heterogeneous computing. It allows algorithms that follow common high-order patterns (like reductions, or element-wise operations) to be easily expressed using C++ templates. BOLT hides device selection, scheduling, and memory placement from the programmer. This allows programmers to target devices without needing to learn a new programming model. In addition, BOLT programs contain less boiler plate code than OpenCL programs, however BOLT is not as flexible as OpenCL.

### 2.3 The AMD Fusion APU

The AMD Fusion APU [11] is a processor that was released by AMD in 2011. It is called the Fusion APU (Accelerated Processing Unit) because it contains both a multi-core x86 CPU and a programmable GPU on die. Despite having both devices on-die, depending on how memory is allocated, it still favours accesses either by the CPU or GPU [10]. Fusion still partitions memory into three regions: GPU memory, CPU memory, and ‘uncached’ memory, which is very similar to GPU memory. CPU reads to GPU memory are still slow because they cannot be accelerated by the CPU’s cache hierarchy and the Fusion only supports one pending read to GPU memory at a time. Similarly, GPU reads to CPU memory have lower peak bandwidth than GPU reads to GPU memory, because they must go through the CPU’s cache hierarchy, instead of only going through the GPU’s memory system which is specially tuned for memory throughput [10]. Note that while AMD renamed its heterogeneous processor line HSA (Heterogeneous System Architecture), the model used for this research is called Fusion, so we continue to use the original name.

### 2.4 Multi2Sim

Multi2Sim [43] is a cycle-accurate simulator for several CPU and GPU architectures. This work uses Multi2Sim’s ability to simulate multicore x86 CPUs and AMD Southern Island GPUs. While Multi2Sim
supports x86 simulation, it does not, by default, model a particular type of x86 CPU. Instead, it allows the user to specify detailed configuration files that define everything from the core count and operating frequency of the CPU to the number of various functional units. For the x86 CPU, Multi2Sim runs 32-bit Linux binaries directly, emulating system calls, and forwarding file I/O operations to the host. Multi2Sim also directly implements thread scheduling because it does not simulate a guest operating system. Multi2Sim uses additional system calls as a custom interface for interacting with the GPU hardware that it simulates. Thus, the GPU OpenCL driver interacts with the hardware through these system calls, whereas the CPU OpenCL driver runs completely on the x86 CPU and is fully compatible with real hardware. Multi2Sim is particularly good for exploring execution across multiple devices because it implements a new cache coherence protocol called NMOESI [42] that allows for the CPU and GPU to efficiently use the same cache hierarchy.

2.5 Related Work

We divide the related work into four categories:

1. frameworks that distribute kernels across multiple devices but use their own programming model,
2. frameworks that distribute kernels across multiple devices and use OpenCL,
3. frameworks that make remote devices seem like they are local,
4. characterization and simulation of heterogeneous systems.

2.5.1 Distribution with Custom Interfaces

HASS [39] takes a signature-based approach for determining which core to run a task on in a single-ISA heterogeneous system (which contains faster and slower cores of the same type). In HASS’s case, these signatures are based on the frequency of memory accesses generated by the tasks, and the paper argues why this is a good statistic to measure; however, the signature approach could be taken with other statistics as well. Qilin [30] introduces a new programming model that allows heterogeneous execution across a CPU and GPU system. It uses profiling to measure the performance of tasks as they execute on cores and determine how to divide up work. Both Qilin and our work can distribute a task between a CPU and GPU, but Quilin also requires its own programming model, which allows it to track memory. CUDASA [41] extends the CUDA [32] programming model to include network and bus levels on top of the pre-existing kernel, block and thread levels. Although CUDASA does provide a mechanism for the programmer to distribute a kernel between computers, it requires them to make explicit function calls to move memory between computers. The kernel code must also be modified in order to be distributed.

Some of these works also focus on particular classes of problems. One work [33] considers splitting 2D FFT operations between the CPU and GPU of a heterogeneous computer. 2D FFT operations contain many steps that can be done in parallel, so they are a good candidate for heterogeneous execution. In the FFT work, the performance of each device is profiled for each step of the 2D FFT and for various sizes of input. Because this work only does 2D FFTs, this profiling only needs to be done once. Another work [40] distributes linear algebra operations across GPU clusters and heterogeneous processors in both shared memory and distributed environments. It works by dividing the input matrices into blocks, and assigning them to different nodes. Within each node, the blocks are further divided between devices. A
graph is dynamically built based on knowledge of the algorithm to detect any dependencies and transfer between nodes when necessary. Its authors claim near optimality with regards to memory transfers. The linear algebra runtime can be seen as a domain-specific, optimized version of DistCL, with meta-functions and partitioning algorithms particularly suited for linear algebra.

### 2.5.2 Distribution of OpenCL Kernels

Work by Kim et al. [23] transparently distributes OpenCL kernels between multiple GPUs, with distinct memories, on the same PCIe bus. Instead of using meta-functions, it uses compiler analysis and sample runs of certain work-items to determine the memory accesses that will be performed by a subset of a kernel. Since the kernel is only divided among local GPUs there is no need to worry about network delays or manage multiple processes.

Work [26] concurrent with this thesis is similar to the work by Kim et al. in how it distributes a task across the devices of a heterogeneous computer. It uses a compiler to create a distributable version of an OpenCL kernel, analysing the data access patterns of the kernel, to determine whether the kernel accesses buffers in a way that allows distribution. Artificial intelligence techniques are used to try to partition OpenCL kernels on a system by running training sets that characterize the performance of the system and kernels that are to be run on it. DualCL and Multi2Sim’s OpenCL implementation do not require such detailed compiler analysis because they use devices that can access a single memory. In the future, DistCL could use similar compiler techniques to eliminate the need for meta-functions, enabling completely automatic distribution across clusters, as well as multi-device computers. Also, this thesis focuses on kernel partitioning based on runtime data only.

A similar work [20] focuses on predicting OpenCL kernel performance using compiler analysis to detect features that affect performance, and determine how to best divide the kernel between an CPU and GPU with known performance characteristics. The compiler analysis uses no runtime information.

### 2.5.3 Making Remote Devices Appear Local

SnuCL [24] is a framework that distributes OpenCL kernels across a cluster. SnuCL can create the illusion that all the OpenCL devices in a cluster belong to a local context, and can automatically copy buffers between nodes based on the programmer’s placement of kernels. SnuCL also executes on CPUs as well as GPUs. To do this, SnuCL made their own CPU OpenCL implementation because the NVIDIA platform SnuCL runs on does not support CPU execution. When two kernels on two different devices read the same buffer, SnuCL replicates the buffer. When two kernels write to the same buffer, SnuCL serializes the kernels. Because of this, what would be one buffer for a single-device application is typically many buffers for an application that uses SnuCL. This pattern of dividing buffers into regions is also seen in DistCL, but with DistCL’s approach, buffers are automatically divided based on a kernel’s memory access pattern.

Other prior work also proposes to dispatch work to remote GPUs in a cluster environment as if they were local. rCUDA [16] provides this functionality for NVIDIA devices. MGP [7] also provides OpenMP-style directives for parallel dispatch of many OpenCL kernels across a cluster. Hybrid OpenCL [5], dOpenCL [22], and Distributed OpenCL [17] provide this functionality cross vendor and cross platform. In cOpenCL [1] each remote node is represented locally as a separate platform. All these works, as well as SnuCL, require the programmer to create a separate set of buffers and kernel invocations for
each device and rely on explicit commands from the host program for communication between peers. LibWater [19] is also similar to the above works, but takes advantage of task dependencies inferred using the OpenCL event model to automatically schedule tasks efficiently. These works differ from our contributions because we focus on distributing a single kernel, not just marshalling kernels between devices.

2.5.4 Characterizing and Simulating Heterogeneous Execution

Two works [45] [8] examine the effects of memory transfer overheads between the CPU and GPU. Both conclude that sometimes, and unsurprisingly, the cost of transfer overheads outweighs the benefit of doing computations on a faster device.

Delorme [14] describes a CPU and GPU radix sort algorithm for the Fusion APU. Versions of radix sort that use disjoint device memories and share memories are developed and compared. The technique that Delorme uses to allow the CPU and GPU to share the same OpenCL buffer was used as part of DualCL’s buffer allocation scheme. Delorme only used OpenCL for the GPU, using standard windows threads for the portion of the sort that runs on the CPU.

There are also other heterogeneous system simulators. FusionSim [48] is a simulator that combines GPGPU-Sim [6] (which simulates NVIDIA GPUs) and PLTSim [47] (which simulates an x86 CPU) into a heterogeneous system simulator. FusionSim does not support OpenCL, nor does it currently have a runtime capable of automatically scheduling tasks across the CPU and GPU simultaneously.

Another simulator-based work [46] looks at a theoretical fused architecture, where the CPU and GPU share the same last-level cache. To do this, it combines MARSSx86 [34] with GPGPU-Sim. The combined simulator also modelled one way in which memory could be shared between the CPU and GPU, using the Linux ioremap function. The simulator revealed that the CPU could be used to prefetch data into the cache before it is used by the GPU, though the work did not investigate dividing work between the CPU and GPU.

Two works [44] [37] combine GPGPU-Sim with gem5 [9] to create a simulator of single-chip heterogeneous CPU-GPU systems. That work maps GPU memory to a part of host memory which is not kept coherent with the CPU. Its authors use this simulator to evaluate running OpenCL programs across both cores, while imposing whole-system power constraints by using DVFS (Dynamic Voltage and Frequency Scaling) and DCS (Dynamic Core Scaling) to control the power consumption of the cores. They create an algorithm that allocates work to GPU and CPU while controlling the number of active cores, the ratio of work assigned to the CPU and GPU, and the frequency and voltage of those cores. The work evaluates this algorithm using the simulator and an AMD Trinity APU, which contains both a CPU and GPU. As kernels are executed, their algorithm converges to a close-to optimal configuration. That work is similar to ours in that it divides work between the CPU and GPU, and it includes experiments from simulations and a real heterogeneous system. Conversely, that work focuses on power and core performance, whereas our work focuses more on memory and invocation overheads. Further, the algorithm presented in that work only makes one partitioning decision per kernel invocation, and takes several kernel invocations to converge to an optimal solution, whereas our algorithms try to converge during a single kernel invocation.
Chapter 3

Co-operative OpenCL Execution

This chapter takes the approach of dynamically dividing OpenCL kernels between the CPU and GPU of a heterogeneous chip without any before-hand knowledge regarding the kernels or the chip. This approach allows us to evaluate how well-suited a heterogeneous system is to execution across both of its devices while exposing the strengths and weaknesses of the system itself. In particular, this chapter focuses on distributed execution where both devices can access each other’s memory. We term this type of distribution co-operative execution. Co-operative execution has two major challenges:

- Allocating memory so that it is effectively used by both devices. This is particularly difficult in current hardware, where each device prefers to access its own memory.
- Scheduling work efficiently. This is difficult in current hardware where GPU invocation overheads are significant compared to kernel run-times. Execution overheads are the single most important factor for dynamic execution, as they affect not only overall system performance, but the performance estimates some of the scheduling algorithms make as well.

This chapter introduces two new OpenCL implementations: DualCL and m2sOpenCL. DualCL works on real hardware, and m2sOpenCL runs in the Multi2Sim simulator, but they both distribute OpenCL kernels between a CPU and GPU that can access the same physical memory - and they both cause these two devices to appear as a single device to their client programs. In order to operate on real hardware, DualCL uses a pre-existing OpenCL implementation to target the CPU and GPU devices. It allocates memory so that it stays accessible to both devices simultaneously, and transforms the source code of OpenCL kernels so that they remain correct when their NDRanges are broken up into smaller parts. M2sOpenCL uses Multi2Sim’s unified memory model to allocate memory. M2sOpenCL itself contains drivers for the CPU and GPU devices it supports, so it does not require any kernel transformations as it can control kernel execution directly. M2sOpenCL is the OpenCL implementation for Multi2Sim, and supports both single-device and co-operative kernel execution.

In this chapter, experiments were performed on three systems:

1. A system with a discrete GPU, where CPU-GPU communication happens over the PCIe bus and each device has a physically separate memory.
2. A system with an AMD Fusion APU, where communication is intra-chip and one physical memory is divided into separate CPU and GPU regions.
Table 3.1: Contributions to Multi2Sim’s Co-operative Execution

<table>
<thead>
<tr>
<th>Component</th>
<th>Contributor</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 OpenCL Driver</td>
<td>Author</td>
</tr>
<tr>
<td>Southern Islands GPU Driver</td>
<td>Multi2Sim Developers</td>
</tr>
<tr>
<td>Co-operative Execution Driver</td>
<td>Author</td>
</tr>
<tr>
<td>NMOESI Protocol</td>
<td>Multi2Sim Developers</td>
</tr>
<tr>
<td>Rodinia Benchmark Analysis</td>
<td>Author</td>
</tr>
<tr>
<td>Some Co-operative Execution Optimizations</td>
<td>Multi2Sim Developers</td>
</tr>
<tr>
<td>Non-Device Portions of OpenCL Runtime</td>
<td>Author</td>
</tr>
<tr>
<td>CPU and GPU Simulation</td>
<td>Multi2Sim Developers</td>
</tr>
</tbody>
</table>

3. A simulated system where communication is intra-chip and memory is fully unified.

With these systems, this chapter make the following findings:

- Co-operative performance on single-chip and discrete hardware is similar despite the tighter integration on chip.

- Co-operative scheduling algorithms fail to use a heterogeneous system’s full compute throughput because of compromises made with data placement, and high GPU invocation overhead, but shows a speedup when these compromises are factored out.

We did not write all of m2sOpenCL. In particular, the portion of m2sOpenCL that targets the simulated GPU was contributed by Multi2Sim’s developers. Our contributions include the runtime’s over-all design and the CPU runtime. Table 3.1 clarifies what each party did.

3.1 Outline

First, this Section 3.2 describes how the work of a kernel is represented and divided by DualCL and m2sOpenCL. Next, Section 3.3 describes the architecture of DualCL: how DualCL allocates buffers so that both the CPU and GPU can use them; modifies the kernels to run correctly across multiple devices; and schedules work onto the CPU and GPU. Then, Section 3.4 describes m2sOpenCL, focusing on the co-operative device driver. After that, Section 3.5 introduces the partitioning algorithms that DualCL and m2sOpenCL use to divide work between the devices. Then, the benchmarks are introduced, and the experimental study in Section 3.6, with DualCL running on discrete and Fusion hardware, and m2sOpenCL running on Multi2Sim, is described. The effect of memory placement is explored. In Section 3.7, the overall performance of the partitioning algorithms on both real-hardware systems is shown. For the Fusion system, the effect of overheads from the CPU invocation, GPU invocation, and DualCL’s operation are examined. Then, the performance of specific benchmarks is looked at in detail. Finally, the performance of specific benchmarks on the simulated system are investigated well.

3.2 Dividing Kernel Execution

Conventionally, the entire NDRange of a kernel invocation is executed on a single compute device. This situation is illustrated in Figure 3.1a. With co-operative execution, more than one device is used to
execute the same kernel invocation. This is done by dividing up the work-groups of the invocation between the compute devices, as illustrated in Figure 3.1b. Note that in both figures, there is only one memory.

### 3.3 DualCL

DualCL is an OpenCL implementation that combines two OpenCL devices, which are provided by an underlying OpenCL implementation, into a single OpenCL device. To do this, DualCL uses memory that both devices can access. Because the underlying OpenCL implementation cannot be made aware that a kernel is being scheduled across multiple devices, DualCL must modify the source of the kernel so that it executes correctly. DualCL works on the AMD Fusion APU, as well as systems with discrete AMD GPUs.

A primary goal of DualCL is ease of use. For that reason, DualCL runs completely unmodified OpenCL kernels and host programs. While creating two versions of each kernel (one for the CPU and one for the GPU) may result in increased performance, it is also an extra burden on the programmer. We do not claim that DualCL approaches the performance of hand tuning. Instead, we assert that, for kernels with relatively few global accesses, DualCL sees a speedup with no effort on the programmer’s behalf.

Certain simple code transformations are sometimes applied to GPU-centric kernels to make them execute more efficiently on the CPU. For instance, a for-loop can be added around the kernel body, causing each CPU thread to do the work of many GPU threads. Though DualCL does perform some basic source-code transformations to ensure correctness, it does not apply this technique. Indeed, automatic compiler techniques such as work-item coalescing [27] already perform similar transformations, so DualCL leaves it to the underlying OpenCL implementation to run kernels on the CPU efficiently.

DualCL takes a dynamic approach to scheduling kernel work, making several scheduling decisions per kernel. While many current GPU workloads can be served well with static scheduling approaches, this trend is starting to change. New features like dynamic parallelism [31] already allow a kernel to launch arbitrary amounts of additional work as a part of its computation. Further, DualCL is not tied to a particular architecture. Any system that supports zero-copy memory and has two OpenCL devices should be able to use DualCL with minimal modification. Though this means that DualCL does not take advantage of any architecture-specific optimizations, it allows DualCL to remain a viable research
3.3.1 Memory Allocation

DualCL must allocate global memory buffers that are accessible from the CPU and GPU simultaneously. There are two types of memory that DualCL can use for buffers: CPU memory or host memory: the memory used by the host program; and GPU memory, or device memory. Note that while both the CPU and GPU are OpenCL devices, device memory always refers to the GPU’s memory in this chapter.

To allocate a buffer, DualCL first allocates a buffer to the GPU (using either host or device memory) and then maps the buffer to the host’s address space. Because the CPU device and host program exist in the same process, any memory accessible by the host is also accessible by the CPU device. This allows DualCL to create a buffer object for the CPU that refers to the same memory as the GPU buffer object. Though the OpenCL specification does not define the behaviour of memory that is simultaneously-writeable from more than one device, using a technique inspired by Delorme [14], DualCL always allocates zero-copy memory, ensuring that only one copy of the buffer is ever created by the runtime. By using zero-copy memory, DualCL relies on the hardware coherence and consistency mechanisms for correctness.

3.3.2 Partitioning Kernels

Work-items can use work-item functions, such as get_global_size() and get_global_id() to get geometric information about their NDRange. When DualCL divides the NDRange of a kernel up into smaller NDRanges, it must ensure that these work-item functions continue to refer to the original NDRange otherwise the kernel will not execute correctly. In other words, if an NDRange is divided into many NDRanges for co-operative execution, the geometric information provided to the kernel should not reflect the division and the work-items should continue to think they are part of one big kernel. Figure 3.2b shows what happens if the NDRange in Figure 3.2a is divided in half naively. In this case, the size of the NDRange and IDs of some of the work-items no longer correspond to the original NDRange. Particularly, the global size is wrong (8 instead of the original 16), and the work-item and work-group IDs are wrong for the work-items in the second NDRange. Figure 3.2c shows how they must be adjusted to ensure correct execution.

To perform these adjustments, DualCL modifies kernels to accept three additional parameters, which contain information about the original NDRange. In addition, DualCL replaces the calls to work-item functions with versions that use the parameters in conjunction with geometric information returned by the underlying (unadjusted) work-item functions. The parameters, and the information they contain, are shown in Table 3.2 and the new functions are shown in Table 3.3. Note that get_global_id() does not need to be modified in this way because DualCL uses a pre-existing OpenCL feature to achieve the same effect. DistCL, which is discussed in Chapter 5 also uses this technique for adjusting the values of work-item functions when it splits up NDRanges.

3.4 m2sOpenCL

Multi2Sim’s OpenCL implementation, m2sOpenCL, consists of two main parts: a device-independent layer that implements the OpenCL interface, and a series of device drivers that expose the capabilities of
### Table 3.2: Parameters Added to Kernel Functions by DualCL

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>ulong4 dualcl_num_groups</td>
<td>The total number of work-groups, in each dimension, in the original NDRange</td>
</tr>
<tr>
<td>ulong4 dualcl_group_start</td>
<td>The starting point of this smaller NDRange in work-groups, in each dimension</td>
</tr>
<tr>
<td>ulong4 dualcl_offset</td>
<td>The starting point of this smaller NDRange in work-items, in each dimension</td>
</tr>
</tbody>
</table>

### Table 3.3: Substitutions Performed by DualCL

<table>
<thead>
<tr>
<th>Work-Item Function</th>
<th>Replacement (using parameters above)</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_global_size(x)</td>
<td>(dualcl_num_groups.sx * get_local_size(x))</td>
</tr>
<tr>
<td>get_group_id(x)</td>
<td>(dualcl_group_start.sx + get_group_id(x))</td>
</tr>
<tr>
<td>get_num_groups(x)</td>
<td>(dualcl_num_groups.sx)</td>
</tr>
<tr>
<td>get_global_offset(x)</td>
<td>(dualcl_offset.sx)</td>
</tr>
</tbody>
</table>

![Diagram](image1.png)

(a) An NDRange with 4 Work-groups

![Diagram](image2.png)

(b) The NDRange has been divided into two separate NDRanges naively

![Diagram](image3.png)

(c) The NDRange has been divided into two NDRanges and the geometric information has been adjusted

Figure 3.2: Dividing an NDRange into Smaller NDRanges
devices to the device-independent layer. Currently there are three drivers: an x86 CPU driver, Southern Islands GPU driver, and co-operative execution driver, which seamlessly divides work between the CPU and GPU drivers. Figure 3.3 shows how the runtime and device drivers of m2sOpenCL interact.

3.5 Partitioning Algorithms

Partitioning algorithms tell DualCL and m2sOpenCL how to divide the NDRange between the devices. To run the partitioning algorithms, each device has a thread associated with it that asks for work, and submits that work to its device. The threads do not ask for more work until their device has completed the work it has already been assigned, nor do threads ever wait to ask for more work once their device becomes ready.

To ensure correctness, partitioning algorithms never break-apart work-groups. Also, for simplicity, when partitioning multidimensional NDRangeS, our partitioning algorithms partition only along the dimension with the most work-groups. Though the shape of the parts a partitioning algorithm creates can affect the performance of the kernel, an exploration of this effect is left to future work.

3.5.1 Static Partitioning

The static partitioning algorithm simply divides the NDRange into two parts – one for the CPU and one for the GPU. It can be configured (through an environmental variable) to divide the NDRange at any proportion. This allows us to determine the division of work that is optimal for a given kernel and system. For programs that execute more than one kernel, if each kernel has a different optimal point, static partitioning may fail to produce the lowest execution time because our implementation of static partitioning does not vary the proportion between kernels of the same program. This situation is explored in detail in the results section.
3.5.2 Block by Block (block)

Block divides the NDRange into 16 equally-sized blocks; though, due to rounding, sometimes more blocks are created. The AMD OpenCL manual recommends dividing work between two OpenCL devices only if, for that particular work, the slower device is at least 10% as fast as the faster device. For this reason, a block size of a sixteenth of the work was chosen because it is a power of two, and because dividing the work further would only be useful in situations where one of the devices was less than 10% the speed of the other.

Initially, each device is assigned a block. Whenever a device completes, that device is assigned another block until there are no more blocks left and execution is complete. The faster a device is, the more often it can ask for blocks; hence, faster devices should execute more of the kernel than slower ones. Block uses a work-queue approach, where pre-determined units of work are dequeued and executed on each device.

3.5.3 NUMA (numa)

Numa was based on a paper about loop parallelization for NUMA machines [28]. It attempts to strike a balance between load balancing and preserving locality. From a single work queue, numa makes large assignments at the beginning, to preserve locality, but makes smaller ones as the kernel nears completion, to reduce the risk of one device becoming a straggler. Numa sizes assignments proportional to the amount of work left divided by the number of devices. In this case, the constant of proportionality is $\frac{1}{2}$, so with two devices, it assigns a quarter of the remaining work whenever a device becomes ready.

3.5.4 First-Past-The-Post (first)

First executes $\frac{3}{16}$ of the problem on each device. This number was selected empirically based on the results from the static partitioning. Then, whichever device completes first is assigned the rest of the problem. For kernels that strongly favour either the CPU or GPU, first will approximate an optimal solution. For short kernels, invocation overhead can mislead first into picking the wrong device.

3.5.5 Same Latency (same)

Same tries to divide work between the CPU and GPU so that they complete at the same time. Initially, same assigns $\frac{1}{8}$ of the work to each device. Same assigns $\frac{1}{8}$ of the work instead of block’s $\frac{1}{16}$ to better amortize the invocation overhead, reducing the effect it has on same’s calculations. After the initial assignment, whenever a device needs more work, same uses the performance data it collected to try and divide up the remaining work between the two devices. Same bases this division on the rate at which each device is executing work-groups and an estimation of how far along the other device is in its last assignment. In this sense, same is similar to guided self-scheduling [36], which assigns work to processors so that they all finish at the same time, taking into account the time at which each processor started. If the other device has not completed its initial assignment, same assumes it is on the verge of completion. Because the estimations are updated every time a device completes, same can make a varying number of assignments if the performance estimations change.
3.6 Experimental Study

To investigate co-operative execution, DualCL was used to run benchmarks from the Rodinia 2.3 [12] benchmark suite on two systems: a fusion system, containing an AMD Fusion APU, and a discrete system, containing a discrete AMD GPU. The specifications of the fusion and discrete system are displayed in Table 3.4. In the first experiment, the benchmarks were run using a single device. For consistency, these runs were still performed using DualCL. For each device, two runs were performed: one that used host memory and one that used device memory. Next, using host memory, the benchmarks were run using both devices with the static and dynamic partitioning algorithms. Device memory was not used for any other experiments because it sometimes produced incorrect results and has very poor performance when accessed by the CPU.

We believe that the inaccurate results with device memory occur when the CPU reads stale data from its write-combining buffer. Instead of using the cache hierarchy, each CPU core uses its write-combining buffer to write to device memory. Similarly, the GPU has its own write-combining buffers. Write-combining buffers allow writes to adjacent memory accesses to be ‘combined’ into one memory transaction, improving performance. When the CPU reads device memory, it goes through its write combining buffer as a first step [10]. This is presumably so that the CPU can see writes it has performed even if they haven’t been flushed to DRAM.

Given this setup, consider a situation where the CPU and GPU write different values to the same location in device memory. If either device immediately reads the data it wrote, it will get the value from its write-combining buffer. In this situation, the CPU and GPU do not have a consistent view of memory.

In the bfs benchmark, the host program writes a one-byte flag, invokes a kernel, and then reads the flag to check for a change. For kernel invocations with small numbers of work-groups, it is possible that the CPU core running the host program never performs any work and hence never flushes its write-combining buffer. Thus, when it reads the value of the flag after the kernel, it reads back a stale value. This issue only arises when zero-copy memory is mapped on both the CPU and GPU simultaneously, as is the case with DualCL.

We did not investigate the Fusion’s ‘uncached’ CPU memory because it has similar characteristics to GPU memory for CPU reads, saturating at less than 1 GB/s, but with degraded performance for the GPU reads [10]. DualCL was written with a single, unified memory in mind, as we view the many types of memory present on current heterogeneous processors as a symptom of an intermediate level of integration between the CPU and GPU.

The static partitioning algorithm was run 17 times for each benchmark, varying the distribution of work from all on the CPU to all on the GPU in increments of a 16th of the total work. Using OpenCL’s profiling capabilities, information about the invocation overheads incurred with partitioning algorithm run was collected. In addition, DualCL’s overhead was calculated too. Each run was performed three times, and the run with the median runtime was taken. Though three runs is not very many, variance was controlled mainly by increasing the problem size instead of increasing the number of runs. Variance between runs is mainly due to variation in GPU invocation overhead, which depends on how state is cached on both the GPU and host process, as well as contention for the CPU.

For the simulated results, the static and dynamic algorithms were run in the same fashion - except that one run was done for every benchmark, as the variance in runtime on the simulation is less than on real hardware. The simulated hardware only has one type of memory, so different memory placement was
Table 3.4: Test Systems

<table>
<thead>
<tr>
<th>Category</th>
<th>Fusion System</th>
<th>Discrete System</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>AMD Fusion A3650 APU</td>
<td>Intel Core 2 Duo E6600</td>
</tr>
<tr>
<td>Clock</td>
<td>2.6 GHz</td>
<td>2.4 GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>L1 Data</td>
<td>4 × 64KB</td>
<td>2 × 32KB</td>
</tr>
<tr>
<td>L1 Instruction</td>
<td>4 × 64KB</td>
<td>2 × 32KB</td>
</tr>
<tr>
<td>L2</td>
<td>4 × 1MB</td>
<td>4 MB</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Radeon HD 6530D</td>
<td>FirePro 2270</td>
</tr>
<tr>
<td>Architecture</td>
<td>AMD Evergreen</td>
<td>AMD Cedar</td>
</tr>
<tr>
<td>Clock</td>
<td>444 MHz</td>
<td>600 MHz</td>
</tr>
<tr>
<td>Compute Units</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Processing Elements Per CU</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>Connection to CPU</td>
<td>On-Chip</td>
<td>PCIe</td>
</tr>
<tr>
<td><strong>RAM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speed</td>
<td>1866 MHz</td>
<td>667 MHz (CPU) 600 MHz (GPU)</td>
</tr>
<tr>
<td>Amount</td>
<td>8GB</td>
<td>4 GB (CPU) 512 MB (GPU)</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operating System</td>
<td>Windows 7 Professional (32-bit)</td>
<td></td>
</tr>
<tr>
<td>AMD APP SDK Version</td>
<td>2.8</td>
<td></td>
</tr>
</tbody>
</table>

not explored. Also, while Appendix A summarizes our attempt to get Multi2Sim to model the Fusion’s CPU, we were unsuccessful because Multi2Sim’s model is not yet accurate enough, and underestimates the performance of x86 CPUs. Also, Multi2Sim only supports Southern Islands-architecture GPUs with the m2sOpenCL driver, and the Fusion contains an Evergreen-architecture GPU. Though the Southern Islands and Evergreen architectures are similar, they are not identical.

Instead, we used Multi2Sim’s default CPU and GPU configurations, modifying only the core count to match the Fusion system’s configuration. The CPU’s clock was left at Multi2Sim’s default, 1 GHz, but the GPU was set to 10 MHz, to compensate for the pessimistic CPU model. The effect of the issues of the CPU model make it difficult to simulate any real-world CPU; hence, the 10 MHz value for the GPU was chosen so that the CPU and GPU have similar performance. Had the GPU operated at a more realistic frequency, it would have been far too powerful in comparison to the CPU for co-operative execution to be effective. The value of 10 in particular was chosen because it was a round number. We did not attempt to match CPU and GPU performance exactly, as it varies by benchmark. For all results, both simulated and real, the runtimes presented reflect only kernel execution. Because zero-copy memory was used, memory transfer times were not considered.

3.6.1 Benchmarks

Though Rodinia is a UNIX-based benchmark suite, we got nine benchmarks built and running correctly for our Windows-based runtime. Windows was used instead of Linux because AMD only supports zero-copy memory on Evergreen in Windows. The discrete card used in this evaluation supports a maximum of 128 work-items per work-group - half that of the Fusion, so six benchmarks ran correctly on the
discrete system. Six of the benchmarks could not run in Mult2Sim because the Southern Islands GPU simulation does not yet support the entire Southern Islands instruction set. The nine benchmarks used are described below.

**Back Propagation (backprop)**

Back propagation is a method for training a neural network. The training happens in two kernels. The first activates the input neurons and measures the result at the output. The second uses the error observed at the outputs to propagate changes in the weights of the neural network back toward the input neurons. For real hardware, we passed backprop an input size of 102400 elements. This benchmark runs 2 kernels.

**Breadth-First Search (bfs)**

Breadth-first search performs a search in a graph. We passed bfs a 1 million element graph. As configured, this benchmark runs 24 kernels.

**Computational Fluid Dynamics (cfd)**

This computation models a compressible fluid using Euler’s equations. As configured, this benchmark runs 14004 kernels.

**Gaussian Elimination (gaussian)**

This is the standard matrix operation. Each pair of kernels solves a row of the matrix based on the row above it. We passed this kernel a 1024 by 1024 element matrix. As configured, this benchmark runs 2046 kernels.

**Hotspot Chip Simulation (hotspot)**

Hotspot simulates how heat spreads through a chip by dividing the chip into elements and modelling the interaction between neighbouring elements. We passed this kernel a 1024 by 1024 element chip and simulated it for 8 time steps. As configured, this benchmark runs 2 kernels.

**K-Means Clustering (kmeans)**

K-Means is the standard artificial intelligence clustering algorithm. As configured, this benchmark runs 38 kernels.

**Nearest Neighbour (nn)**

Nearest neighbour calculates the distance between a reference point and an array of points that represent the position of hurricanes. After this parallel calculation, the host sorts the array and reports the 10 closest hurricanes. We increased the number of ‘hurricanes’ used in this benchmark to about 4 million. This benchmark runs one kernel.
Needleman-Wunsch (nw)
Needleman-Wunsch is a matrix operation used to find matches in DNA sequences. As configured, this benchmark runs 511 kernels.

Path Finder (pathfinder)
Path Finder finds the least-weighted route through a grid. Each kernel analyses a row of the grid at a time. As configured, this benchmark runs 5 kernels.

3.7 Results and Discussion
First, this section presents the overall performance of co-operative execution versus single-device execution for each real system. Next, the performance of each of the dynamic partitioning algorithms is examined and compared to the fastest static partitioning run. To gain insight into the results, we examine some key properties of the dynamic partition algorithms’ behaviour. The most important property is the fraction of the total work that the algorithms assign to each device. This section refers to this division of work as the work ratio and represents the ratio numerically as the proportion of work-groups assigned to the CPU. The amount of invocation overhead incurred by the dynamic scheduling algorithms is also examined. Finally, detailed results for some of the benchmarks are displayed, showing all of the runs of the static sweep and dynamic algorithms.

3.7.1 Memory Placement
Figure 3.4 shows the runtimes of the benchmarks when being executed by the CPU and GPU on host and device memory. In addition, the runtime of the fastest dynamic partitioning algorithm and the optimal static partitioning is displayed for each benchmark. Figure 3.5 shows the same results for the discrete machine. For each benchmark, the results are normalized to the performance of the GPU using device memory - the fastest device. The y-axis of the graph has been plotted with a logarithmic scale, to lend equal weight to speedups and slowdowns. For the same reason, we used the geometric mean to average the benchmarks, as shown in the rightmost cluster of the graph.

On average, for the Fusion system, while the fastest static algorithm beat the GPU and the fastest dynamic algorithm did not, both beat the CPU using host memory and GPU when using host memory. This shows that the partitioning algorithms speed up execution, but that the speed-up they provide cannot totally compensate for the compromise of always using host memory. The GPU benefits from using device memory, which is optimized for large coalesced memory accesses at the price of latency. However, for nn and nw, the GPU performs better with host memory. For nw this is not so surprising, as the CPU is better at this algorithm which suggests that nw is better suited to the CPU memory hierarchy. On the other hand, nn involves large contiguous memory accesses, for which the GPU memory hierarchy is better. This trend likely has to do with how OpenCL caches buffers in host memory before their first use. Because nn executes only one kernel per run, it is particularly sensitive to this effect. To test this theory on the discrete system, the nn benchmark was modified to execute the kernel multiple times. In this situation, kernel performance exceeded performance with host memory by about 10%.

Hotspot and pathfinder, which use local memory, share a clear pattern. Memory placement for these benchmarks does not matter, as the time taken to transfer the data to and from local memory is
Chapter 3. Co-operative OpenCL Execution

Figure 3.4: Different Device and Memory Combinations on Fusion

...amortized by the computation which uses the local, on-chip memory.

Overall, the CPU using device memory was over an order of magnitude slower than baseline, making it a poor choice for co-operative execution. This, combined with issues we experienced trying to get device memory to work correctly when accessed by both devices, meant that we did not use it for any further experiments.

The fastest dynamic partitioning algorithms were only faster than the GPU using device memory on the discrete system, but this is mainly because cfd and hotspot, for which dynamic partitioning performed poorly, did not run on the discrete system. That said, looking at only host memory averages, the fastest dynamic partitioning outperformed the CPU and GPU on both systems.

3.7.2 Partitioning Algorithms

Figure 3.6 and Figure 3.7 show the performance of the individual dynamic partitioning algorithms versus static partitioning for the Fusion and discrete system, respectively. As before, these results have been normalized to the runtime of the fastest device using its own memory.

For the Fusion system, static partitioning performed the best - or at least close to the best - with the exception of backprop. This is because backprop contains two kernels (per run) that differ in the ratio they should be divided between the CPU and GPU for optimal performance. The dynamic partitioning algorithms are free to vary the ratio they assign from kernel to kernel, so they can accommodate this difference whereas static partitioning cannot.

The numa, block, and same partitioning algorithms performed similarly, on average, with both systems. First, on the other hand, was the best algorithm on the Fusion system and the worst on the discrete system. This is because first chose to run most of the work on the CPU device for nn on the
Figure 3.5: Different Device and Memory Combinations on Discrete System

discrete system even though the CPU is the slower device for that benchmark.

Figure 3.8 shows what proportion of work each scheduling algorithm assigned to the CPU (with the remainder being assigned to the GPU). On average, the optimal static partitioning scheduled about 20% less work on the CPU than the scheduling algorithms did. This is due to the increased GPU invocation overhead when compared to the CPU. As expected, first scheduled either 3/16 or 13/16 of the work onto the CPU for bfs, hotspot, nn, and pathfinder. For other benchmarks, first made different scheduling decisions for each kernel in a run, so the average is somewhere in between. For bfs, the fastest static algorithm scheduled everything on the CPU, though all the scheduling algorithms initially assign both devices work, similarly, for nn the fastest static partitioning scheduled everything on the GPU. The insight that most scheduling algorithms scheduled too much work on the CPU explains why first did well on the Fusion machine: because it scheduled the least amount of work on the CPU.

Figure 3.9 and Figure 3.10 show the amount of invocation overhead incurred by the CPU and GPU, respectively, as a fraction of total execution time. This overhead was calculated by subtracting the ‘queued’ time from the ‘start’ time of the kernels, as reported by the underlying AMD OpenCL runtime. CPU invocation overhead is relatively low. Several benchmarks have CPU invocation overheads under 1% for the CPU. On the other hand, GPU invocation overhead is much higher. Also, while hotspot and nn incurred very little CPU overhead, they incurred very high GPU overhead, implying that GPU overhead and CPU overhead arise from different effects. The high GPU invocation for nn also helps to explain how a scheduling algorithm like first could incorrectly identify the faster device for the benchmark (though for the result considered, first chose correctly).

The high (and variable) invocation overhead of the GPU explains why, on average, the scheduling algorithms put more work on the CPU than was optimal - because the GPU’s overhead made it seem slower. GPU invocation overhead is hard to predict because it depends on how GPU data structures are
Figure 3.6: Partition Algorithms on Fusion System

Figure 3.7: Partition Algorithms on Discrete System
cached in both the host process and on hardware, and is affected by CPU utilization as well. Of course, even static partitioning incurs the latency of one GPU invocation - but the other scheduling algorithms incur the latency many times, or make future decisions about scheduling taking the latency into account.

Though these experiments did not investigate situations where the GPU was ‘warmed up’ by an initial kernel invocation before doing actual work, if the first invocation accounted for most of the overhead, block and numa would have performed better because their first runs would have acted as initializations.

Figure 3.11 shows the overhead of the dynamic scheduling system itself. This overhead was determined by calculating how much time each device spent doing work (including overheads) and subtracting the larger of those two numbers from the total runtime. DualCL’s own overhead trends closely with CPU overhead, which makes sense because DualCL and the OpenCL CPU device both contend for the same CPU hardware. We tried to alleviate this contention by forcing the OpenCL CPU device to leave one core unused, but this failed to address the main performance issue - invocation overhead - and hence had a minimal impact on the overall results.

### 3.7.3 Individual Benchmarks

Many of the benchmarks reveal interesting information when looked at in detail. This thesis represents the results of a particular benchmark with a graph that displays the results of all the static and dynamic runs. The x-axis of these graphs is the proportion of the work each partitioning algorithm scheduled on the CPU and on the y-axis is execution time in ms. On each graph, all 17 of the static partition runs are displayed, with each one assigning 1/16 more work onto the CPU than the previous one. The static partitioning lines are connected with a line to show how the execution time trends as the work ratio is varied. The dynamic partitioning algorithms are each graphed as a single point, representing...
Figure 3.9: CPU Kernel Invocation Overhead

Figure 3.10: GPU Kernel Invocation Overhead
the overall work ratio and execution time that the algorithm attained. When a dynamic algorithm lies above the static curve, it means that the algorithm operated less efficiently than the static algorithm for the same work ratio. The overall performance of a dynamic partitioning algorithm can be determined by comparing both its work ratio and runtime to the lowest point on the static partitioning curve.

Figure 3.12 and Figure 3.13 show the results for the kmeans benchmark on the Fusion and discrete systems, respectively. For both systems, the static partitioning curve has a minimum that is neither at the CPU nor GPU, which implies that maximum performance is attained when the devices share the problem. For kmeans, all of the scheduling algorithms found the optimal point. In particular, for the discrete machine, the scheduling algorithms actually beat the static algorithm slightly. This is because the kmeans algorithm is actually comprised of two kernels, each with their own optimum. Though, with kmeans one kernel is run much more frequently than the other, so the effect is minimal. Thus, we save our discussion of this effect in detail for our discussion of backprop where its impact is much greater.

Figure 3.14 shows nearest neighbour on the Fusion system and Figure 3.15 shows it on the discrete system. Unlike the kmeans results, the static partitioning line does not have a minimum in the middle, signifying that no heterogeneous speedup is possible with DualCL. For both systems, the GPU was the faster device. For the Fusion system, first was the best dynamic partitioning algorithm. All of the other dynamic algorithms divided the work more or less evenly between the CPU and the GPU. In particular, though numa scheduled far more pieces of work onto the GPU than the CPU, the two pieces of work it scheduled on the CPU represented a big enough proportion of the work that the eventual distribution was nowhere near the optimal. On the discrete device, numa, block and same do better. This time, numa scheduled only the initial quarter of work on the CPU. On the other hand, first assigned most of the work to the CPU, resulting in a much higher runtime than the other scheduling algorithms.

Figure 3.16 shows pathfinder on the Fusion system and Figure 3.17 shows it on the discrete system.
Chapter 3. Co-operative OpenCL Execution

Figure 3.12: Kmeans on Fusion System

Figure 3.13: Kmeans on Discrete System
Figure 3.14: Nearest Neighbour on Fusion System

Figure 3.15: Nearest Neighbour on Discrete System
On both the discrete and Fusion systems, pathfinder’s static curve shows that there is some benefit to using both devices; however, first is the only scheduling algorithm able to attain any speedup. Nevertheless, the rest of the scheduling algorithms still come close to finding the ideal proportion of work. In particular, on the discrete system, they are only 1/8th away from the optimal.

Figure 3.18 shows the execution of the back propagation algorithm on the Fusion system. Here, the CPU and GPU have similar performance and the scheduling algorithms all perform well. In particular, they outperform the static sweep. This is because the back propagation benchmark is a combination of two kernels - each with their own sweet spot. Figure 3.19 and Figure 3.20 show the runtime of each kernel individually. In these figures, the dynamic scheduling algorithms do not out-perform the static partitioning on either of the kernels, but because the static partitioning is varied across benchmark executions, and not within benchmark executions, it is can be beaten by scheduling algorithms that can adapt on a per-kernel basis when kernels of the same benchmark have different characteristics.

Figure 3.19 and Figure 3.20 show the results for the first and second kernel invocations of backprop separately. At the individual kernel level, scheduling algorithms, as expected, lie on or just above the static partitioning curve. Also, they partition the first kernel with more work on the GPU than the second kernel, allowing them to beat the static partitioning algorithm. Of course, this difference in performance reflects the nature of our experiment more than it does a fundamental limitation in static partitioning, as each kernel could have been statically partitioned differently as well.
Figure 3.17: Pathfinder on Discrete System

Figure 3.18: Back Propagation on Fusion System
Figure 3.19: First Kernel of Back Propagation on Fusion System

Figure 3.20: Second Kernel of Back Propagation on Fusion System
3.7.4 Simulated Results

We ran three of the Rodinia benchmarks in Multi2Sim. Although several more are supported by the m2sOpenCL runtime, Multi2Sim’s Southern Islands GPU implementation is incomplete, and can only run four benchmarks. One of these, kmeans, runs in functional simulation mode, but not detailed (cycle-accurate) simulation mode. The limited support for these benchmarks in Multi2Sim is because support for the NMOESI protocol, Southern Islands GPU, and shared memory hierarchy are all still being actively developed by Multi2Sim’s developers.

Figures 3.21, 3.23 and 3.24 show the result of running the static and dynamic scheduling algorithms in Multi2Sim. Note that the execution time on the graphs for the simulations is in simulated time. For gaussian and nn, the CPU and GPU are relatively evenly matched, and the curves produced by the static partitioning algorithms are far more linear than in the real hardware results, particularly for gaussian. This is particularly interesting because the simulated problem sizes are much smaller than the real-world problem sizes, so any invocation overheads would have a much larger impact on overall performance. In fact, Multi2Sim does not consider PCIe bus latency or system call overheads at all when modelling GPU performance. CPU invocation overhead is also reduced because Multi2Sim’s OpenCL implementation is optimized for fast invocation, and Multi2Sim does not model resource contention with an operating system kernel. Multi2Sim’s NMOESI protocol also relaxed any communication bottlenecks that exist in real hardware.

The gaussian simulation results show all three scheduling algorithms achieving speedup versus the faster CPU device (as evidenced by the fact that the algorithms’ runtimes are below that of the static
partition run that was all on the CPU). The static partitioning is also an ideal check mark curve, showing almost ideal speedup; however, the scheduling algorithms do not lie on the static partitioning curve. We suspect this is because of cache locality. Gaussian is a very compute-intensive algorithm, so small data-sets were used for simulation. In fact, the results in Figure 3.21 are from a $64 \times 64$ matrix.

When block and numa partition a kernel into very small parts, these parts operate on a very small regions of the matrix that fit entirely in the cache. The next part of the kernel operates on some of the same data and thus, can benefit from the cache locality if it is assigned to the same device, although, sometimes it is not. To test this theory, we reduced the size of the problem to a $32 \times 32$ matrix to see if the difference in performance would increase. As shown in Figure 3.22, it is considerably more-pronounced. To investigate this phenomenon further, we created a version of DualCL that assigned the CPU and GPU work from opposite ends of the NDRange instead of interleaving CPU and GPU allocations; however, this had a negligible effect on the larger problem sizes of real hardware so we did not use it in simulation.

Back-propagation favoured the CPU significantly, taking only $731 \ \mu s$ on the CPU to execute. It is interesting to see that the scheduling algorithms here each favoured the CPU as much as they could. The best-performing algorithm is block, which always execute 1/16th of the work on each device to start. Next is first that execute 3/16th of the work on each device to start; and finally is numa which allocates 1/4 of the work to the first device to request it which, for this benchmark, was the GPU.

In simulation, nearest neighbour also shows a speedup with both the static partitioning and scheduling algorithms over the fastest device. For this benchmark, the CPU got numa’s first allocation, so its second allocation, which was 1/4 of the 3/4 of the remaining work (or 3/16 of the work), matched that of firsts,
causing them both to allocate $3/16$ of the work to the GPU. Interestingly, `block` did even worse, allocating only $1/8$th of the work to the GPU, whereas an ideal divide would have been closer to $3/8$.

### 3.8 Conclusions

DualCL and m2sOpenCL are two OpenCL implementations that allow co-operative execution of kernels. DualCL works on real hardware, accessing the CPU and GPU devices using an underlying OpenCL implementation. DualCL allocates buffers in host memory that is accessible to both devices, and applies source code transformations to kernels to allow them to be partitioned. Using partitioning algorithms that dynamically divide the NDRange of a kernel up based on earlier performance, on average, DualCL sees a speedup over a baseline of GPU execution with host memory. M2sOpenCL is Multi2Sim’s OpenCL implementation which allows for direct co-operative execution without the need for code transformations. M2sOpenCL exploits Multi2Sim’s NMOESI cache consistency protocol to allow for data exchange between the CPU and GPU.

Host memory performs better than device memory for co-operative execution because the relatively small GPU in the Fusion APU incurs less of a penalty accessing host memory than the CPU does accessing GPU memory. That said, the similar performance trends seen in the discrete system and the Fusion system shows that current single-chip heterogeneous processors are still very similar to discrete systems, and offer very little advantage to co-operative execution, exhibiting similar invocation overheads and memory placement trade-offs. Nevertheless, DualCL’s dynamic algorithms did outperform either
device using host memory, the type of memory chosen for these experiments.

Results from m2sOpenCL show that future hardware can improve heterogeneous performance by reducing invocation overheads. In particular, the partitioning algorithms’ performance was much closer to their best theoretical performance; though the accuracy of Multi2Sim itself as an experimental platform is still improving.
Chapter 4

CPU Execution for Multi2Sim

Multi2Sim [43] is a cycle-accurate simulator capable of simulating devices such as x86 CPUs and AMD Southern Islands GPUs. For this reason, amidst all the excitement over heterogeneous systems, Multi2Sim positions itself as a valuable tool for the exploration of systems that contain both a CPU and GPU. To enable the simulation of heterogeneous programs, Multi2Sim allows programs to run computations on its GPUs using OpenCL; however, until our work, Multi2Sim was unable to run OpenCL kernels on its CPUs. This is because it was missing an x86 CPU driver for its OpenCL implementation.

This chapter describes the CPU driver we wrote to enable OpenCL execution on Multi2Sim’s CPU. OpenCL CPU execution is particularly exciting because it allows for the automatic distribution of work between the CPU and GPU. The simulation results presented in Chapter 3 used this driver to run OpenCL on the CPU.

This chapter describes the high level details and goals of the CPU driver, then it switches to the driver’s implementation. It explains how the CPU driver executes individual work-items, then work-groups, and finally entire kernel invocations. In particular, it focuses on how barriers were implemented, as they have a significant impact on performance. Finally, this chapter evaluates the performance of the CPU driver as our driver is not useful if it is too much slower than the real x86 AMD runtime. We find that the driver’s performance is within 6% of AMD’s implementation, when running AMD benchmarks.

4.1 Overall Design

For consistency with Multi2Sim’s GPU OpenCL driver, the CPU driver runs binaries that were compiled with the AMD APP SDK version 2.7 OpenCL compiler. Because AMD does not publish any documentation explaining the format of their binaries, we had to reverse engineer the binaries to create the driver. Further, the CPU driver runs both in Multi2Sim and on real hardware. This allows us to compare the CPU driver’s performance to that of other software that does not easily run in simulation, like AMD’s x86 runtime. Finally, we designed the CPU driver to support co-operative execution with other devices such as Multi2Sim’s CPU. To facilitate this goal, the CPU driver has been contributed to the Multi2Sim project, and is now part of all Multi2Sim releases.
4.1.1 The OpenCL Device Model for CPUs

Our implementation executes binaries in the same way as the AMD driver with which it has binary compatibility. In both implementations, each core in a CPU becomes a compute unit with one processing element. This means that a core only runs one work-group at a time. Whereas GPUs typically interleave the execution of work-items within a work-group, our CPU driver does not. A call to \texttt{barrier()} made by the work-items of a work group will cause them to synchronize, but without barriers, work-items within a work-group just run to completion, one after another. Also, the private, local, global and constant memories are all implemented with normal, cacheable memory on CPUs.

4.2 Behaviour of a Work-Item

An AMD-compiled kernel binary contains machine code that represents the behaviour of a single work-item. This is in contrast with other techniques, like work-item coalescing [27], that generate code representing an entire work-group. Because of this, a work-item’s calls to \texttt{barrier} require the current work-item to yield the CPU to another work-item from the work-group. Hence, each work-item is a co-operatively scheduled thread with its own stack, which it uses to store all private memory and its own copy of the parameters passed to the kernel.

4.2.1 Work-Item Stack Layout

At the ABI (Application Binary Interface) level, work-items are C functions that use the standard C calling convention. This means that the stack grows toward lower memory addresses and parameters to the kernel are pushed onto the stack from right to left. The return address that the work-item should jump to upon completion is pushed onto the stack last. Contrary to 32-bit conventions, work-items expect the first 4 floating point parameters to be passed using SSE registers XMM0 through XMM3.

Work-item functions like \texttt{get\_global\_id()} are implemented as lookups into a special stack-resident data structure. Table 4.1 shows the information stored at each offset. This data-structure is located in the highest-addressed part of the stack, just above the parameters passed to the work-item. The data-structure is always aligned to end just below a multiple of 8kB, allowing a work-item to round its stack pointer up to the nearest multiple of 8k to locate the data-structure. For this to work, the size of a work-item’s stack (and hence the amount of private data it can store) is limited to just under 8kB. Figure 4.1 shows work-item’s entire stack, including the data-structure, kernel parameters, and any private memory. Note that each work-item has its own ID, and hence, has its own, private, data-structure in its stack.

4.3 Fibres

For efficiency’s sake, though each work-item has its own stack, all the work-items in a work-group execute in the same operating system thread. Instead of being scheduled pre-emptively by the OS, they are scheduled co-operatively by the CPU driver. Co-operatively scheduled contexts executed in this fashion are often called ‘green threads’, ‘user threads’, ‘co-routines’ or ‘fibres’. This thesis uses the term ‘fibres’.
Table 4.1: Work-Item Information Data-Structure.

<table>
<thead>
<tr>
<th>Contents</th>
<th>Offset (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>padding</td>
<td>-0x4</td>
</tr>
<tr>
<td>get_global_id(2)</td>
<td>-0x8</td>
</tr>
<tr>
<td>get_global_id(1)</td>
<td>-0xC</td>
</tr>
<tr>
<td>get_global_id(0)</td>
<td>-0x10</td>
</tr>
<tr>
<td>padding</td>
<td>-0x14</td>
</tr>
<tr>
<td>get_group_id(2)</td>
<td>-0x18</td>
</tr>
<tr>
<td>get_group_id(1)</td>
<td>-0x1C</td>
</tr>
<tr>
<td>get_group_id(0)</td>
<td>-0x20</td>
</tr>
<tr>
<td>padding</td>
<td>-0x24</td>
</tr>
<tr>
<td>get_local_size(2)</td>
<td>-0x28</td>
</tr>
<tr>
<td>get_local_size(1)</td>
<td>-0x2C</td>
</tr>
<tr>
<td>get_local_size(0)</td>
<td>-0x30</td>
</tr>
<tr>
<td>padding</td>
<td>-0x34</td>
</tr>
<tr>
<td>get_global_size(2)</td>
<td>-0x38</td>
</tr>
<tr>
<td>get_global_size(1)</td>
<td>-0x3C</td>
</tr>
<tr>
<td>get_global_size(0)</td>
<td>-0x40</td>
</tr>
<tr>
<td>padding</td>
<td>-0x44</td>
</tr>
<tr>
<td>group lowest get_global_id(2)</td>
<td>-0x48</td>
</tr>
<tr>
<td>group lowest get_global_id(1)</td>
<td>-0x4C</td>
</tr>
<tr>
<td>group lowest get_global_id(0)</td>
<td>-0x50</td>
</tr>
<tr>
<td>get_work_dim</td>
<td>-0x54</td>
</tr>
<tr>
<td>pointer to static local memory</td>
<td>-0x58</td>
</tr>
<tr>
<td>pointer to barrier function pointer</td>
<td>-0x5C</td>
</tr>
<tr>
<td>padding</td>
<td>-0x60</td>
</tr>
</tbody>
</table>
Though many fibre implementations already exist, for performance reasons, we implemented our own for the CPU driver. The AMD OpenCL compiler never uses the legacy x87 floating point and segment registers, so those they do not need to be saved upon a context switch. In addition, some fibre implementations re-route UNIX signal masks, which is unnecessary for OpenCL.

As fibres cannot pre-empt each other, they need to call a fibre switch method in order to change which fibre is running on the CPU. In OpenCL, that method is barrier(). To switch between two fibres, first, general purpose registers are backed up on the suspending fibre’s stack and the program counter and stack pointer of the suspending fibre are saved. Then, the resuming fibre is resumed by jumping to it’s program counter, which restores the registers that were backed-up and continues execution beyond the fibre switch call. Switching between fibres does not involve the operating system, making them more efficient than pre-emptively scheduled threads. Further, user-mode processes can allocate the stacks of fibres directly, which allows the OpenCL driver to place work-item stacks back-to-back, improving cache locality when executing kernels.

4.4 Scheduling and Allocating Work-Items

In OpenCL, work-items yield to each other using the barrier() call. In fact, work-group-wide barriers are the only form of in-kernel synchronization that OpenCL supports. Unlike get_global_id() (which is a lookup into the work-item information data structure), a barrier call is a conventional call which runs the barrier function present in the CPU driver.
Chapter 4. CPU Execution for Multi2Sim

4.4.1 Stack Allocation Optimization

When a kernel involves calls to \texttt{barrier()}, many work-items are in flight at once, and each work-item needs its own stack to retain its state; however, if there are no calls to \texttt{barrier()}, the CPU driver may execute the work-items one after another, re-using the same region of memory for all the work-items' stacks. Reusing the same memory increases the performance of the driver because it reduces the working set of the kernel invocation, increasing cache locality. We noticed significant speedups for barrierless kernels when we implemented this optimization.

For this optimization to work, the driver runs the first work-item. If the work-item terminates without making any calls to \texttt{barrier()}, the driver re-uses the work-item's stack, re-initializing it for the second work-item. Alternately, if the first work-item calls \texttt{barrier()}, before the driver runs the
Table 4.2: Test Configuration for OpenCL CPU Driver Comparison

<table>
<thead>
<tr>
<th>Processor Name</th>
<th>Intel Core 2 Duo E6600</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Clock</td>
<td>2.4 GHz</td>
</tr>
<tr>
<td>Processor Cores</td>
<td>2</td>
</tr>
<tr>
<td>Processor L2 Cache</td>
<td>4 MB Shared</td>
</tr>
<tr>
<td>RAM</td>
<td>4 GB</td>
</tr>
<tr>
<td>Operating System</td>
<td>Ubuntu 13.04 (32-bit)</td>
</tr>
<tr>
<td>AMD APP SDK Version</td>
<td>2.7</td>
</tr>
</tbody>
</table>

second work-item, it initializes its stack. This strategy avoids initializing all the stacks for kernels that do not call barrier(), but complicates the implementation of the barrier() function. Note that, in OpenCL, all the work-items in a work-group must call barrier() the same number of times, so the first work-item not calling barrier() guarantees that no other work-item will either.

4.5 Whole-Kernel Execution and Scheduling

The driver runs a thread on each core of the CPU. For an $n$-core CPU, $(n-1)$ of those threads come from the driver, and the remaining thread comes from the command queue, which has its own thread so that it can run operations asynchronously.

To run a kernel, each thread dequeues work-groups and runs them. Because the work-groups are numbered, the dequeuing operation is implemented as an atomic decrement. When this counter reaches zero, and all the threads finish, the kernel is complete, and the command queue advances to the next operation.

To reduce the latency associated with dispatching a kernel, the $(n-1)$ worker threads owned by the driver busy wait for another kernel. Kernel invocation overhead is particularly important for fused execution, when a single kernel can involve scheduling multiple parts onto each device.

4.6 Performance Validation

We evaluated our driver's performance with two types of tests: synthetic micro-benchmarks that stress a particular aspect of the driver and more conventional benchmarks from AMD's development kit. The configuration of the computer used for the test is shown in Table 4.2. We did not use the fusion machine itself because it has a newer version of the AMD APP SDK installed.

Figure 4.3 shows the results of the synthetic micro-benchmark tests. Each test was run three times, and the median was taken. The float test launches a work-group with 256 work-items that add four floating-point numbers together and write the sum to an array element. This test does not include any barriers, it just tests the basic functionality of the driver. The next test is local. In this test, work-items in a single work-group write one element of a local memory array, synchronize with a barrier, then read the value in the next element in the local memory array and add that value to a global memory buffer. This tests both the performance and correctness of local memory accesses. The next test, multi barrier is similar to the local test, except that there are multiple work-groups. The next test, multi also launches multiple work-groups, each with 256 work-items, but each work-item only writes to a global array, so
non-barrier-related overheads of each driver are invoked. Finally, work group is similar to float except that it does no calculations at all - it only involves runtime overheads.

If the ratio of our execution time and AMD’s execution time is taken on a per benchmark basis, the geometric mean of those ratios shows that our runtime is 1.27 times slower than AMDs - within 30%. Although we lose out with the work group test, we beat AMD with the float test, suggesting our mechanism for passing floating point parameters to work-items is more efficient than AMD’s. The main area where we lost out is on the tests that involved dispatching many work-groups. We were slower for multi and multi barrier. This might be because AMD’s runtime dequeues work-groups many-at-a time, reducing contention. Finally, our barrier implementation seems to be faster than AMD’s, as seen by our local test.

We ran benchmarks from the AMD APP SDK samples on our OpenCL runtime and with AMD’s OpenCL runtime targeting the CPU device. For all benchmarks, default parameters were passed, and a verification feature of the benchmarks (which runs a non-OpenCL version algorithm and compares the results) confirmed that our runtime ran the binaries properly. Figure 4.4 shows the ratio in execution time of our execution time to AMD execution time for a subset of the AMD APPSDK benchmarks. A ratio above one means that AMDs runtime was faster whereas a ratio below one means that ours was faster. These ratios are shown on a log graph so that relative speedups and slowdowns are represented equally. For each benchmark, three runs were performed, and the fastest of the three runs was chosen, again to eliminate the effects of background operations. With the APPSDK samples, our runtime has a geometric mean 1.055 times that of AMD’s. Our runtime was faster with the benchmarks even though it was slower in the micro-benchmarks. This is because some micro-benchmarks stressed operations for
which our implementation is slower, like dispatching work-groups, more than the APPSDK benchmarks do.

4.7 Conclusion

This section presented a x86 OpenCL driver that is compatible with the binaries produced by the AMD OpenCL Runtime. In order to create this runtime, we reverse engineered the AMD binaries and created a runtime that could execute these non-standard binaries in a way that meets the OpenCL specification. We used fibres which allowed for efficient execution of work-items, and employed an optimization to reduce the memory footprint of kernels that do not contain barriers. Our kernel dispatching is designed to minimize kernel invocation latency and work-group dispatching involves a single operation. Overall, our driver has similar performance to that of version 2.7 of the AMD runtime. This driver allows us to explore heterogeneous execution in the Multi2Sim simulator.
Chapter 5

Distributed OpenCL Execution

5.1 Introduction

Recently, there has been significant interest in using GPUs for general purpose and high performance computing. Significant speedups have been demonstrated when porting applications to a GPU [6] [38]; however, additional speedups are still possible beyond the computational capabilities afforded by a single GPU. Therefore, it is no surprise that modern computing clusters are starting to include multiple GPUs [25]. However, distributing a data-parallel task across a cluster still involves the following steps:

1. partitioning work into smaller parts,
2. scheduling these parts across the network,
3. partitioning memory so that each part of memory is written to by at most one device, and
4. tracking and transferring these parts of memory.

This chapter introduces DistCL, a framework for the distribution of OpenCL kernels across an MPI-based multi-node cluster. To simplify this task, DistCL takes advantage of three insights:

1) OpenCL tasks (called kernels) contain threads (called work-items) that are organized into small groups (called work-groups). Work-items from different work-groups cannot communicate during a kernel invocation. Therefore, work-groups only require that the memory they read be up-to-date as of the beginning of the kernel invocation. So, DistCL must know what memory a work-group reads, to ensure that the memory is up-to-date on the device that runs the work-group. DistCL must also know what memory each work-group writes, so that future reads can be satisfied. But no intra-kernel synchronization is required.

2) Most OpenCL kernels make only data-independent memory accesses; the addresses they access can be predicted using only the immediate values they are passed and the geometry they are invoked with. This means that their accesses can be efficiently determined before they run. DistCL requires that kernel writes be data-independent.

3) Kernel memory accesses are often contiguous. This is because contiguous accesses fully harness the wide memory buses of GPUs [38]. DistCL does not require contiguous accesses for correctness, but they improve distributed performance because contiguous accesses made in the same work-group can be treated like a single, large access when tracking writes and transferring data between devices.
DistCL introduces the concept of meta-functions: simple functions that describe the memory access patterns of an OpenCL kernel. Meta-functions are programmer-written kernel-specific functions that relate a range of work-groups to the parts of a buffer that those work-groups will access. When a meta-function is passed a range of work-groups and a buffer to consider, it divides the buffer into intervals, marking each interval either as accessed or not accessed by the work-groups. DistCL takes advantage of kernels with sequential access patterns, which have fewer (larger) intervals, because it can satisfy their memory accesses with fewer I/O operations. By dividing up buffers, meta-functions allow DistCL to distribute an unmodified kernel across a cluster. To our knowledge, DistCL is the first framework to do so.

In addition to describing DistCL, this chapter evaluates the effectiveness of kernel distribution across a cluster based on the kernels’ memory access patterns and their compute-to-transfer ratio. It also examines how the performance of various OpenCL and network operations affect the distribution of kernels.

This chapter makes the following primary contributions:

- The introduction of DistCL and its concept of meta-functions, which allow for the distribution of unmodified kernels.
- An evaluation of how the properties of different kernels affect their performance when distributed.

This chapter describes DistCL using vector addition as an example, in particular looking at how DistCL handles each step involved with distribution (Section 5.2). The benchmarks are introduced and grouped into three categories: linear runtime benchmarks, compute intensive benchmarks, and benchmarks that involve inter-node communication (Section 5.3). Results for these benchmarks are presented (Section 5.4). A comparison with SnuCL [24] (described in Section 2.5.3) is also provided. Finally, conclusions are drawn about how the properties of an OpenCL kernel and cluster hardware impact distributed performance (Section 5.5).

5.2 DistCL

DistCL executes on a cluster of networked computers. OpenCL host programs use DistCL by interacting with one compute device that represents the aggregate of all the devices in the cluster. When a program is run with DistCL, identical processes are launched on every node. When the OpenCL context is created, one of those nodes becomes the master. The master is the only node that is allowed to continue executing the host program. All other nodes, called peers, enter an event loop that services requests from the master. Nodes communicate in two ways: messages to and from the master, and raw data transfers that can happen between any pair of nodes.

To run a kernel, DistCL divides its NDRange into smaller grids called subranges. These subranges are analogous to the parts created by the partitioning algorithms described in Section 3.5. Kernel execution gets distributed because these subranges run on different peers. DistCL must know what memory a subrange will access in order to distribute the kernel correctly. This knowledge is provided to DistCL with meta-functions. Meta-functions are programmer-written, kernel-specific callbacks that DistCL uses to determine what memory a subrange will access. DistCL uses meta-functions to divide buffers into arbitrarily-sized intervals. Each interval of a buffer is either accessed or not. DistCL stores the intervals calculated by meta-functions in objects called access-sets. Once all the access-sets have been calculated,
Listing 5.1: OpenCL kernel for vector addition.

```c
__kernel void vector(__global int *a, __global int *b, __global int *out) {
    int i = get_global_id(0);
    out[i] = a[i] + b[i];
}
```

Figure 5.1: Vector’s 1-dimensional NDRange is partitioned into 4 subranges.

DistCL can initiate the necessary transfers needed to allow the peers to run the subranges they have been assigned. Note the important distinction between subranges which contain threads and intervals which contain data. The remainder of this section describes the execution process in more detail, illustrating each step with a vector addition example, whose kernel source code is given in Listing 5.1.

### 5.2.1 Partitioning

Partitioning divides the NDRange of a kernel execution into smaller grids called subranges. DistCL never fragments work-groups, as that would violate OpenCL’s execution model and could lead to incorrect kernel execution. For linear (1D) NDRangeS, if the number of work-groups is a multiple of the number of peers, each subrange will be equal in size. Otherwise, some subranges will be one work-group larger than others. DistCL partitions a multidimensional NDRange along its highest dimension first, in the same way it would partition a linear NDRange. If the subrange count is less than the peer count, DistCL will continue to partition lower dimensions. This strategy assumes that the GPUs in the cluster have similar performance in contrast to the partitioning algorithms used in DualCL, which attempts to balance performance differences.

Multidimensional arrays are often organized in row-major order, so highest-dimension-first partitioning frequently results in subranges accessing contiguous regions of memory. Transferring fragmented regions of memory requires multiple I/O operations to avoid transferring unnecessary regions, whereas large contiguous regions can be sent all at once.

Our vector addition example has a one-dimensional NDRange. Assume it runs with 1M = 2^{20} work-items on a cluster with 4 peers. Assuming 1 subrange per peer, the NDRange will be partitioned into 4 subranges, each with a size of 256k work-items, as shown in Figure 5.1.

### 5.2.2 Dependencies

DistCL tracks global memory buffers when they are passed to a kernel. For example, the three parameters, a, b, and out in Listing 5.1 are buffers.

DistCL must know what parts of each buffer a subrange will access in order to create the illusion of many compute devices with separate memories sharing a single memory. The set of addresses in a buffer that a subrange reads and writes are called its read-set and write-set, respectively. DistCL represents these access-sets with concrete data-structures and calculates them using meta-functions. Access-sets are
calculated every kernel invocation, for every subrange-buffer combination, because the access patterns of a subrange depend on the invocation’s parameters, partitioning, and NDRange. In our vector addition example with 4 subranges and 3 buffers, 24 access-sets will be calculated: 12 read-sets and 12 write sets.

An access-set is a list of intervals within a buffer. DistCL represents addresses in buffers as offsets from the beginning of the buffer; thus an interval is represented with a low and high offset into the buffer. These intervals are half open; low offsets are part of the intervals, but high offsets are not.

For instance, subrange 1 in Figure 5.1 contains global IDs from the interval [256k, 512k). As seen in Listing 5.1, each work-item produces a 4-byte (sizeof (int)) integer, so subrange 1 produces the data for interval [1 MB, 2 MB) of out. Subrange 1 will also read the same 1 MB region from buffers a and b to produce this data. The intervals [0 MB, 1 MB) and [2 MB, 4 MB) of a, b and out are not accessed by subrange 1.

Calculating Dependencies

To determine the access-sets of a subrange, DistCL uses programmer-written, kernel-specific meta-functions. Each kernel has a read meta-function to calculate read-sets and a write meta-function to calculate write-sets.

DistCL passes meta-functions information regarding the kernel invocation’s geometry. This includes the invocation’s global size (global in Listing 5.2), the current subrange’s size (subrange), and the local size (local), although local is not used in this particular meta-function. DistCL also passes the immediate parameters of the kernel (params) to the meta-function. The subrange being considered is indicated by its starting offset in the NDRANGE (subrange_offset) and the buffer being considered is indicated by its zero-indexed position in the kernel’s parameter list (param_num).

DistCL builds access-sets one interval at a time, progressing through the intervals in order, from the beginning of the buffer to the end. Each call to the meta-function generates a new interval. If and only if the meta-function indicates that this interval is accessed, DistCL includes it in the access-set.

To call a meta-function, DistCL passes the low offset of the current interval through start and the meta-function sets next_start to its end. The meta-function’s return value specifies whether the interval is accessed. Initially setting start to zero, DistCL advances through the buffer by setting the start of subsequent calls to the previous value of next_start. When the meta-function sets next_start to the size of the buffer, the buffer has been fully explored and the access-set is complete.

Rectangular Regions

Many OpenCL kernels structure multidimensional arrays into linear buffers using row-major order. When these kernels run, their subranges typically access one or more linear, rectangular, or prism-shaped areas of the array. Though these areas are contiguous in multidimensional space, they are typically made up of many disjoint intervals in the linear buffer. Recognizing this, DistCL has a helper function, called require_region, that meta-functions can use to identify which linear intervals of a buffer constitute any such area.

require_region, whose prototype is shown in Listing 5.3, operates over a hypothetical dim-dimensional grid. Typically, each element of this grid represents an element in a DistCL buffer. The size of this grid in each dimension is specified by the dim-element array total_size. require_region considers a rectangular region of that grid whose size and offset into the grid are specified by the dim-element arrays required_size and required_start, respectively. Given this, require_region
Listing 5.2: Example read meta-function for vector.

```c
int is_buffer_range_read_vector(
    const void **params,
    const size_t *global,
    const size_t *subrange,
    const size_t *local,
    const size_t *subrange_offset,
    unsigned int param_num,
    size_t start, size_t *next_start)
{
    int ret = 0;
    *next_start = sizeof(int) * global[0];
    if (param_num != 2) {
        start /= sizeof(int);
        ret = require_region(
            1,
            global, subrange_offset, subrange, start,
            next_start);
        *next_start *= sizeof(int);
    }
    return ret;
}
```

Listing 5.3: require_region helper function.

```c
int require_region(
    int dim,
    const size_t *total_size,
    const size_t *required_start,
    const size_t *required_size,
    size_t start,
    size_t *next_start);
```

calculates the linear intervals that correspond to that region if the elements of this dimension-dimensional grid were arranged linearly, in row-major-order. Because there may be many such intervals, the return value, `start` parameter and `next_start` parameter of `require_region` work the same way as in a meta-function, allowing the caller to move linearly through the intervals, one at a time. If a kernel does not access memory in rectangular regions, it does not have to use the helper function.

Even though vector has a one-dimensional NDRange, `require_region` is still used for its read meta-function in Listing 5.2. This is because `require_region` not only identifies the interval that will be used, but also identifies the intervals on either side that will not be used. `require_region` is passed `global` as the hypothetical grid’s size, making each grid element correspond to an integer, the datatype changed by a single work-item. Therefore, lines 10 and 12 of Listing 5.2 translate between elements and bytes, which differ by a factor of `sizeof(int)`.

Figure 5.2 shows what actually happens when the meta-function is called on buffer `a` for subrange 1. In Figure 5.2a, the first time the meta-function is called, DistCL passes in 0 as the start of the interval and the meta-function calculates that the current interval is not in the read set, and that the next interval starts at an offset of 1MB. Next, in Figure 5.2b, DistCL passes in 1MB as the start of the interval. The
meta-function calculates that this interval is in the read-set and that the next interval starts at 2MB. Finally, in Figure 5.2c, DistCL passes in 2MB as the start of the interval. The meta-function calculates that this interval is not in the read-set and that it extends to the end of the buffer which has a size of 4 MB.

5.2.3 Scheduling Work

The scheduler is responsible for deciding when to run subranges and on which peer to run them. The scheduler runs on the master and broadcasts messages to the peers when it assigns work. DistCL uses a simple scheme for determining where to run subranges. If the number of subranges equals the number of peers, each peer gets one subrange; however, if the number of subranges is fewer, some peers are never assigned work.

5.2.4 Transferring Buffers

When DistCL executes a kernel, the data produced by the kernel is distributed across the peers in the cluster. The way this data is distributed depends on how the kernel was partitioned into subranges, how these subranges were scheduled, and the write-sets of these subranges. DistCL must keep track of how the data in a buffer is distributed, so that it knows when it needs to transfer data between nodes to satisfy subsequent reads - which may not occur on the same peer that originally produced the data.

DistCL represents the distribution of a buffer in a similar way to how it represents dependency information. The buffer is again divided into a set of intervals, but this time each interval is associated with the ID of the node that has last written to it. This node is referred to as the owner of that interval.

Buffers

Every time the host program creates a buffer, the master allocates a region of host (not device) memory, equal in size to the buffer, which DistCL uses to cache writes that the host program makes to the buffer. Whether the host program initializes the buffer or not, the buffer’s dependency information specifies the master as the sole owner of the buffer. Additionally, each peer allocates, but does not initialize, an OpenCL buffer of the specified size. Generally, most peers will never initialize the entire contents of their buffers because each subrange only accesses a limited portion of each buffer.
Satisfying Dependencies

When a subrange is assigned to a peer, before the subrange can execute, DistCL must ensure that the peer has an up-to-date copy of all the memory in the subrange's read-set. For every buffer, DistCL compares the ownership information to the subrange's read-set. If data in the read-set is owned by another node, DistCL initiates a transfer between that node and the assigned node. Once all the transfers have completed, the assigned peer can execute the subrange. When the kernel completes, DistCL also updates the ownership information to reflect the fact that the assigned peer now has the up-to-date copy of the data in the subrange's write-set. DistCL also implements host-enqueued buffer reads and writes using this mechanism.

Transfer Mechanisms

Peer-to-peer data transfers involve both intra-peer and inter-peer operations. For memory reads, data must first be transferred from the GPU into a host buffer. Then, a network operation can transfer that host buffer. For writes, the host buffer is copied back to the GPU. DistCL uses an OpenCL mechanism called mapping to transfer between the host and GPU.

5.3 Experimental Setup

Eleven applications, from the Rodinia benchmark suite v2.3 [12] [13], AMD APPSDK [2], and GNU libgcrypt [18], were used to evaluate our framework. Each benchmark was run three times, and the median time was taken. This time starts when the host initializes the first buffer and ends when it reads back the last buffer containing the results, thereby including all buffer transfers and computations required to make it seem as if the cluster were one GPU with a single memory. The time for each benchmark is normalized against a standard OpenCL implementation using the same kernel running on a single GPU, including all transfers between the host and device. We group the benchmarks into three categories:

1. Linear compute and memory characteristics: nearest neighbor, hash, Mandelbrot;
2. Compute-intensive: binomial option, Monte Carlo;
3. Inter-node communication: n-body, bitonic sort, back propagation, HotSpot, k-means, LU decomposition.

These benchmarks were chosen to represent a wide range of data-parallel applications. They will provide insight into what type of workloads benefit from distributed execution. The three categories of problems give a spread of asymptotic complexities. This allows the effect of distribution, which primarily affects memory transfers, to be studied with tasks of varying compute-to-transfer ratios. Note that the first two categories consist solely of embarrassingly parallel benchmarks and the last category contains no such benchmarks. The important characteristics of the benchmarks are summarized in Table 5.1, and each is described below. We excluded Rodinia benchmarks that required image support, contained data-dependent writes, or contained too few work-groups. From the remaining benchmarks, five were chosen. For the Rodinia benchmarks, many of the problem sizes are quite small, but they were all run with the largest possible problem size, given the input data distributed with the suite. The worst relative
standard deviation in runtime for any benchmark is 11%, with the Rodinia benchmarks’ runtime varying the most due to their smaller problem sizes. For the non Rodinia benchmarks, it was usually under 1%.

Table 5.1: Benchmark description

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Source</th>
<th>Inputs</th>
<th>Complexity</th>
<th>Work-Items per Kernel</th>
<th>Kernels</th>
<th>Problem Size</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nearest neighbour</td>
<td>Nearest neighbor search</td>
<td>Rodinia</td>
<td>42764 locations(n)</td>
<td>O(n)</td>
<td>n</td>
<td>1</td>
<td>12n</td>
<td></td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>Mandelbrot set calculation</td>
<td>AMD</td>
<td>24M points(n) x: 0 to 0.5, y: -0.25 to 0.25 max iterations(k) 1000</td>
<td>O(kn)</td>
<td>n</td>
<td>1</td>
<td>4n</td>
<td></td>
</tr>
<tr>
<td>Hash</td>
<td>sha-256 cracking</td>
<td>Libgcrypt</td>
<td>24M hashes(n)</td>
<td>O(n)</td>
<td>n</td>
<td>1</td>
<td>32 + n</td>
<td></td>
</tr>
<tr>
<td>Bitonic</td>
<td>Parallel sort</td>
<td>AMD</td>
<td>32M elem.(n)</td>
<td>O(n lg^2 n)</td>
<td>lg^2 n</td>
<td>1</td>
<td>4n</td>
<td></td>
</tr>
<tr>
<td>Binomial option</td>
<td>American option pricing</td>
<td>AMD</td>
<td>786432 samp.(k) 767 iterations(n)</td>
<td>O(knm^2)</td>
<td>k(n + 1)</td>
<td></td>
<td>2m^2(2n + 1)</td>
<td></td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>Asian option pricing</td>
<td>AMD</td>
<td>4k sums(n) 1536 samp.(m), 10 steps(k)</td>
<td>O(kn^2)</td>
<td>k</td>
<td></td>
<td>16n</td>
<td></td>
</tr>
<tr>
<td>n-body</td>
<td>n-body simulation</td>
<td>AMD</td>
<td>568k bodies(n), 1,8 iter.(k)</td>
<td>O(kn^2)</td>
<td>n</td>
<td>k</td>
<td>8nk, 4(2nk + c)</td>
<td></td>
</tr>
<tr>
<td>k-means</td>
<td>clustering</td>
<td>Rodinia</td>
<td>819200 points(n) 34 features(k), 5 clusters(c)</td>
<td>kern. 1 O(nk), kern. 2 O(nck)</td>
<td>n</td>
<td>1, variable</td>
<td>8nk, 4(2nk + c)</td>
<td></td>
</tr>
<tr>
<td>Back propagation</td>
<td>neural network training</td>
<td>Rodinia</td>
<td>4M input nodes(n) 16 hidden nodes(k)</td>
<td>kern. 1 O(nk), kern. 2 O(nk)</td>
<td>nk</td>
<td>2</td>
<td>4(kn + 3n + 2k + 4), 4(kn + 2n + k + 3)</td>
<td></td>
</tr>
<tr>
<td>HotSpot</td>
<td>Heat transfer</td>
<td>Rodinia</td>
<td>chip dim.(n) 1k time-steps.</td>
<td>O(n^2) per kernel(s)</td>
<td>δ, total(k) 60</td>
<td></td>
<td>4((n π/2π)^2 + 4n^2)</td>
<td></td>
</tr>
<tr>
<td>LUD</td>
<td>LU-decomposition</td>
<td>Rodinia</td>
<td>matrix dim.(n) 2k, 2n - 1 iterations (k) current iter. denoted(i)</td>
<td>kern. 1 O(n) kern. 2 O(n^2)</td>
<td>16</td>
<td>k + 1</td>
<td>4n^2</td>
<td></td>
</tr>
</tbody>
</table>

5.3.1 Cluster

DistCL was evaluated using a cluster with an Infiniband interconnect [29]. The configurations and theoretical performance are summarized in Table 5.2. The cluster consists of 49 nodes. Though there are two GPUs per node, we use only one to focus on distribution between machines. We present results for 1, 2, 4, 8, 16 and 32 nodes.

We use three microbenchmarks to test the cluster and to aid in understanding the overall performance of our framework. The results of the microbenchmarks are reported in Table 5.3. We first test the performance of memcpy(), by copying a 64 MB buffer between two points in host memory. We initialize both arrays to insure that all the memory was paged-in before running the timed portion of the code. The measured memory bandwidth was 20.3 Gbps.

To test OpenCL map performance, a program was written that allocates a buffer, executes a GPU kernel that increments each element of that buffer, and then reads that buffer back with a map. The program executes a kernel to ensure that the GPU is the only device with an up-to-date version of the buffer. Every time the host program maps a portion of the buffer back, it reads that portion, to force it
Table 5.2: Cluster specifications

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>49</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPUs Per Node</td>
<td>2 (1 used)</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Tesla M2090</td>
</tr>
<tr>
<td>GPU memory</td>
<td>6 GB</td>
</tr>
<tr>
<td>Shader / Memory clock</td>
<td>1301 / 1848 MHz</td>
</tr>
<tr>
<td>Compute units</td>
<td>16</td>
</tr>
<tr>
<td>Processing elements</td>
<td>512</td>
</tr>
<tr>
<td>Network</td>
<td>4 × QDR Infiniband (4 × 10 Gbps)</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel E5-2620</td>
</tr>
<tr>
<td>CPU clock</td>
<td>2.0 GHz</td>
</tr>
<tr>
<td>System memory</td>
<td>32 GB</td>
</tr>
</tbody>
</table>

Table 5.3: Measured cluster performance

<table>
<thead>
<tr>
<th>Transfer type</th>
<th>Test</th>
<th>64MB Latency (Gbps)</th>
<th>8B Latency (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-memory</td>
<td>Single thread memcpy()</td>
<td>26.5 (20.3)</td>
<td>0.0030 (21)</td>
</tr>
<tr>
<td>Inter-device</td>
<td>OpenCL map for reading</td>
<td>36.1 (14.9)</td>
<td>0.62 (0.10)</td>
</tr>
<tr>
<td>Inter-node</td>
<td>Infiniband round trip</td>
<td>102 (10.5)</td>
<td>0.086 (3.0)</td>
</tr>
</tbody>
</table>

to be paged into host memory. The program reports the total time it took to map and read the updated buffer. To test the throughput of the map operation, the mapping program reads a 64MB buffer with a single map operation. Only the portion of the program after the kernel execution completes gets timed. We measured 14.9 Gbps of bandwidth between the host and the GPU. The performance of an 8-byte map was measured to determine its overhead. An 8-byte map takes 0.62 ms, equivalent to 100 kbps. This shows that small fragmented maps lower DistCL’s performance.

The third program tests network performance. It sends a 64MB message from one node to another and back. The round trip time for Infiniband took 102 ms and each one-way trip took only 51 ms on average, yielding a transfer rate of 10.5 Gbps. Since Infiniband uses 8b/10b encoding this corresponds to a signalling rate of 13.1 Gbps. This still fall short of the maximum signalling rate of 40 Gbps. Even using a high-performance Infiniband, network transfers are slower than maps and memory copies. For this reason it is important to keep network communication to a minimum to achieve good performance when distributing work. Infiniband is designed to be low-latency, and as such its invocation overhead is lower than that of maps.

5.4 Results and Discussion

Figure 5.3 shows the speedups obtained by distributing the benchmarks using DistCL, compared to using normal OpenCL on a single node. Compute-intensive benchmarks see significant benefit from being distributed, with binomial achieving a speedup of over 29x when run on 32 peers. The more compute-intensive linear benchmarks, hash and Mandelbrot, also see speedup when distributed. Of the inter-node communication benchmarks, only n-body benefits from distribution, but it does see almost perfect scaling from 1-8 peers and speedup of just under 15x on 32 peers. For the above benchmarks, we see better scaling when the number of peers is low. While the amount of data transferred remains
constant, the amount of work per peer decreases, so communication begins to dominate the runtime.

The remaining inter-node communication and linear benchmarks actually run slower when distributed versus using a single machine. These benchmarks all have very low compute-to-transfer ratios, so they are not good candidates for distribution. For the Rodinia benchmarks in particular, the problem sizes are very small. Aside from LU decomposition, they took less than three seconds to run. Thus, there is not enough work to amortize the overheads.

Figure 5.4 shows the speedups attained, by running the benchmarks on DistCL and SnuCL, relative to normal OpenCL run on a single node, for the benchmarks that we were able to port to SnuCL. Overall, the speedups are comparable, especially for the more compute-intensive benchmarks, which saw the most speed up. Thus, when DistCL and SnuCL are both capable of distributing workloads, those that distribute well see good performance using either framework. SnuCL distributes a task across a cluster by requiring the programmer to divide what would remain one buffer in DistCL into many buffers, as SnuCL tracks dependencies at the buffer granularity whereas DistCL uses intervals. The change to multiple buffers also requires the programmer to modify their OpenCL kernels so that each kernel can compute part of the output instead of always producing everything. When SnuCL runs faster than DistCL, it is because SnuCL forces the task of tracking buffers onto the programmer, who must now manage additional buffers, whereas DistCL uses extra synchronization in its dependency tracking algorithms.

Figure 5.5 show a run-time breakdown of the benchmarks for the 8 peer case. Each run is broken down into five parts: buffer, the time taken by host program buffer reads and writes; execution, the time during which there was at least one subrange execution but no inter-node transfers; transfer, the time during which there was at least one inter-node transfer but no subrange executions; overlapped
transfer/execution, the time during which both subrange execution and memory transfers took place; and other/sync, the average time the master waited for other nodes to update their dependency information.

The benchmarks which saw the most speedup in Figure 5.3 also have the highest proportion of time spent in execution. The breakdowns for binominal, Monte Carlo, and n-body are dominated by execution time; whereas, the breakdowns for nearest neighbor, back propagation and LU decomposition are dominated by transfers and buffer operations, which is why they did not see a speedup. One might wonder why Mandelbrot sees a speedup, but bitonic and k-means do not, despite the proportion of time they spent in execution being similar. This is because Mandelbrot and hash are dominated by host buffer operations, which also account for a significant portion of execution with a single GPU. In contrast, Bitonic and k-means have higher proportions of inter-node communication, which map to much faster intra-device communication on a single GPU.

Table 5.4 shows the amount of time spent managing dependencies. This includes running meta-functions, building access-sets and updating buffer information. The time it took to manage dependencies is expressed in terms of total time per benchmark, time per kernel invocation, and as a proportion of the total runtime. Thus, if a benchmark contains only one kernel, the first two columns of Table 5.4 will be the same. Benchmarks that have fewer buffers like Mandelbrot and Bitonic Sort spend less time applying dependency information per kernel invocation than benchmarks with more buffers. LU decomposition has the most complex access pattern of any benchmark. Its kernels operate over non-coalescable regions that constantly change shape. Further, the fact that none of LU decomposition’s kernels update the whole array means that ownership information from previous kernels is passed forward, forcing the ownership information to become more fragmented, and take longer to process. With the exception of
LU decomposition, the time spent managing dependencies is low, demonstrating that the meta-function based approach is intrinsically efficient.

An interesting characteristic of HotSpot is that the compute-to-transfer ratio can be altered by changing the pyramid height. Pyramid height refers to the number of iterations done per kernel, as each iteration operates on a square one element smaller on each side, forming a step-pyramid-like pattern. The taller the pyramid the higher the compute-to-transfer ratio. However, this comes at the price of doing more computation than necessary. Figure 5.6 shows the speedup of HotSpot run with a pyramid height of 1, 2, 3, 4, 5, and 6. The distributed results are for 8 peers. Single GPU results were acquired using conventional OpenCL. In both cases, the speedups are relative to that framework’s performance using a pyramid height of 1. The number of time-steps used was 60 to ensure that each height was a divisor of the number of time-steps. We can see that for a single GPU, the preferred pyramid height is 2. However, when distributed the preferred size is 5. This is because with 8 peers we have more compute available but the cost of memory transfers is much greater, which shifts the sweet spot toward a configuration that does less transfer per computation.

Benchmarks like Hotspot and LU decomposition that write to rectangular areas of two-dimensional arrays need special attention when being distributed. While the rectangular regions appear contiguous in two-dimensional space, in a linear buffer, a square region is, in general, not a single interval. This means that multiple OpenCL map and network operations need to be performed every time one of these areas is transferred.

We modified the DistCL scheduler to divide work along the y-axis to fragment the buffer regions transferred between peers. This results in performance that is 204× slower on average across all pyramid heights, for 8 peers. This demonstrates that the overhead of invoking I/O operations on a cluster is a
significant performance consideration.

In summary, like SnuCL, DistCL can distribute OpenCL kernels across a cluster, but it is easier to use. DistCL also exposed important characteristics regarding distributed OpenCL execution. Distribution amplifies the performance characteristics of GPUs. Global memory reads become even more expensive compared to computation, and the aggregate compute power is increased. Further, the performance gain seen by coalesced accesses is not only realized in the GPU’s bus, but across the network as well. Synchronization - now a whole-cluster operation - becomes even higher latency. There are also aspects of distributed programming not seen with a single GPU. At a certain point, it is better to transfer extra data with few transfers than it is to transfer less data with many transfers.

5.5 Conclusion

DistCL is a framework for distributing the execution of an OpenCL kernel across a cluster, causing that cluster to appear as if it were a single OpenCL device. DistCL shows that it is possible to efficiently run kernels across a cluster while preserving the OpenCL execution model. To do this, DistCL uses meta-functions that abstract away the details of the cluster and allow the programmer to focus on the algorithm being distributed. Speedups of up to 29 on 32 peers are demonstrated.

With a cluster, transfers take longer than they do with a single GPU, so more compute-intense approaches perform better. Also, certain access patterns generate fragmented memory accesses. The overhead of doing many fragmented I/O operations is profound and can be impacted by partitioning.

By introducing meta-functions, DistCL opens the door to distributing unmodified OpenCL kernels.
Table 5.4: Execution time spent managing dependencies

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Time (µs)</th>
<th>Per Kernel Invocation Time (µs)</th>
<th>Percent of Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mandelbrot</td>
<td>109</td>
<td>109</td>
<td>0.097</td>
</tr>
<tr>
<td>Hash</td>
<td>112</td>
<td>112</td>
<td>0.15</td>
</tr>
<tr>
<td>Nearest neighbor</td>
<td>120</td>
<td>120</td>
<td>0.62</td>
</tr>
<tr>
<td>Binomial</td>
<td>126</td>
<td>126</td>
<td>0.0043</td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>1500</td>
<td>150</td>
<td>0.028</td>
</tr>
<tr>
<td>n-body</td>
<td>166</td>
<td>166</td>
<td>0.00045</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td>29900</td>
<td>91.9</td>
<td>2.9</td>
</tr>
<tr>
<td>k-means</td>
<td>30400</td>
<td>1450</td>
<td>4.6</td>
</tr>
<tr>
<td>Back propagation</td>
<td>434</td>
<td>217</td>
<td>0.017</td>
</tr>
<tr>
<td>HotSpot</td>
<td>23500</td>
<td>981</td>
<td>5.9</td>
</tr>
<tr>
<td>LUD</td>
<td>$2.31 \times 10^7$</td>
<td>60400</td>
<td>30</td>
</tr>
</tbody>
</table>

DistCL allows a cluster with 32 GPUs to be accessed as if it were a single GPU. Using this novel framework, we gain insight into both the challenges and potential of unmodified kernel distribution. DistCL is a joint work between the author and Tahir Diop, where the author wrote DistCL itself, Tahir ported the benchmarks to SnuCL, and both the author and Tahir ported the benchmarks to DistCL. DistCL is available at http://www.eecg.toronto.edu/~enright/downloads.html and was described in our 2013 MASCOTS publication [15].
Chapter 6

Conclusion

Distribution of OpenCL Kernels across multiple devices enables faster processing in both supercomputing and commodity systems. This thesis introduces three new OpenCL implementations: DualCL, which distributes kernels across a CPU and GPU that can share the same physical memory, m2sOpenCL, which distributes kernels across unified-memory hardware simulated by Multi2Sim, and DistCL, which distributes memory across a cluster of networked GPUs with distinct memories.

DualCL demonstrates that for compute-intense workloads and workloads that leverage local memory, automatic scheduling of work across heterogeneous devices can result in speedups without any programmer effort. DualCL also exposes limitations of current hardware, showing that memory must be further unified and GPU overheads reduced for programmers to automatically harness the full parallelism of a heterogeneous chip using DualCL’s approach. The fact that DualCL’s performance was similar on the discrete and single-chip Fusion system is evidence for this claim.

That said, even on current hardware, DualCL’s dynamic partitioning algorithms outperformed the CPU and GPU when only host memory was used. Much of this speedup was due to the fact that DualCL identified the faster device, and for some benchmarks, such as pathfinder and back propagation, DualCL found a near-optimal operating point. DualCL shows that a completely automatic approach to co-operative execution is viable even with today’s level of device and memory integration. Further, we believe that as memories unify and workloads become more complex, DualCL’s performance will improve. DualCL’s weaknesses, invocation overhead and memory compromises, will be mitigated. Its strength, dynamic scheduling, will become more useful.

DistCL is a framework for distributing the execution of an OpenCL kernel across a cluster, causing that cluster to appear as if it were a single OpenCL device. DistCL shows that it is possible to efficiently run kernels across a cluster while preserving the OpenCL execution model. Unlike DualCL and m2sOpenCL, which are completely automatic, DistCL uses user-supplied meta-functions. This meta-function approach demonstrates that distribution across a cluster can be accomplished without any actual knowledge of the cluster on the programmer’s behalf. DistCL only requires the programmer to describe the memory access patterns of their kernel, something the programmer already knows. DistCL shows a speedup for compute-intense kernels, and gives us two interesting insights into distribution over networks: fragmented accesses have a big impact on distributed performance and the higher latency of global memory changes the optimal operating point of some algorithms. DistCL has been released as an open-source project.
6.1 Future Work

One source of GPU invocation overhead - CPU contention - can be controlled by DualCL to a degree. DualCL could be augmented to interrupt CPU OpenCL execution when launching work on the GPU. Also, some of the issues with device memory on the fusion system might be solved by introducing write-combining buffer flushes for certain host-side operations, even if the host still has device memory mapped. DualCL was also written to be extensible to more than two devices; thus, it could be used to study distribution over multiple GPUs as well.

M2sOpenCL will continue to become more valuable as the simulator it runs on, Multi2Sim, matures. Multi2Sim’s NMOESI protocol will allow for studying device interaction on the micro-architectural level, enabling the design of new hardware coherence mechanisms for CPU and GPU memories.

DistCL can be improved by adding kernels that re-arrange data in buffers to reduce the fragmentation in data transfers. Also, the compiler techniques used in contemporary work [26] for single-computer kernel distribution can be applied to DistCL to eliminate the need for meta-functions, though the compiler techniques would still be limited to data-independent kernels.

The three works in this thesis, DualCL, m2sOpenCL and DistCL, show the promise that heterogeneous processing can bring to parallel programming. At the same time, they also highlight the challenges that engineers face when they work to realize that promise.
Appendices
Appendix A

The Accuracy of the Multi2Sim CPU Device

To characterize the performance of our AMD Fusion CPU, we wrote five microbenchmarks in x86 assembly. Each microbenchmark repeats a certain type of operation, or pattern of memory accesses in an unrolled loop. They were run on the fusion system, and then in Multi2Sim, using the benchmarks as a way of validating our CPU models.

On real hardware, to measure time, our micro-benchmarks use the `CLOCK_MONOTONIC` timer exposed by the `clock_gettime()` function, which returns the number of nanoseconds that has elapsed since a certain undefined point in the passed. Multi2Sim’s authors enhanced Multi2Sim to support this function using the cycle count of the simulation and virtual ‘clock frequency’ of the CPU.

A.1 Microbenchmarks

Each of these five micro-benchmarks was written to look at specific architectural features of the CPU.

A.1.1 Register

This test involves moving a value to one register and then back again, over and over, creating a cycle of dependent register move operations. This test should take a cycle on most modern CPUs and is used as a baseline sanity check.

Listing A.1: Register Test

```
1  mov ecx, ebx
2  mov ebx, ecx
```

Listing A.1 shows a single iteration of this cycle. This iteration was unwound 16 times and put in a loop to create the final benchmark.

A.1.2 Dependent Add

This test is just like the register test, except that the move instructions have been replaced by add instructions, creating a chain of dependent add operations.
A.1.3 Independent Add

This benchmark tries to test how many in-flight add operations can be supported at once by the CPU. It does this by issuing 5 add operations at the same time. Though each add operation has one input register in common, the output register (on the left in Intel syntax), differs for each add, making them independent operations. The sequence of independent adds can be seen in Listing A.2.

Listing A.2: Independent Add Test

```
1  add ecx, ebx  
2  add edx, ebx  
3  add edi, ebx  
4  add esi, ebx  
5  add ebp, ebx  
```

So that each loop iteration has 16 operations, just like the previous two benchmarks, 3 of these independent add blocks and an extra `add ecx, ebx` are included in the loop of the benchmark.

A.1.4 Memory Throughput

This benchmark is designed to stress the memory system of the processor. It involves a series of sequential memory accesses with addresses that can all be calculated independently.

Listing A.3: Memory Throughput Test

```
1  mov edx, [edi + esi]  
2  mov [edi + esi], edx  
3  mov edx, [edi + esi + 4]  
4  mov [edi + esi + 4], edx  
5  mov edx, [edi + esi + 8]  
6  mov [edi + esi + 8], edx  
7  mov edx, [edi + esi + 12]  
8  mov [edi + esi + 12], edx  
9  mov edx, [edi + esi + 16]  
10  mov [edi + esi + 16], edx  
11  mov edx, [edi + esi + 20]  
12  mov [edi + esi + 20], edx  
13  mov edx, [edi + esi + 24]  
14  mov [edi + esi + 24], edx  
15  mov edx, [edi + esi + 28]  
16  mov [edi + esi + 28], edx  
```

Listing A.3 shows the entire loop for the memory throughput test. Each pair of instructions is a read and write to a sequential memory address. Though these instructions to have an interdependence on `edx`, this can be resolved by the CPU after the memory accesses. This benchmark not only supports varying the number of operations performed, but the size of the memory region that these operations access, allowing the effect of data locality to be explored.
Appendix A. The Accuracy of the Multi2Sim CPU Device

Table A.1: Results from Multi2Sim Configuration

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Problem Size</th>
<th>Measured Cycles / Op</th>
<th>Simulated Cycles / Op</th>
<th>Percent Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.30</td>
</tr>
<tr>
<td>add_dep</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.30</td>
</tr>
<tr>
<td>add_indep</td>
<td>0.38</td>
<td>0.37</td>
<td>0.37</td>
<td>0.99</td>
</tr>
<tr>
<td>mem_latency</td>
<td>32 kB</td>
<td>3.00</td>
<td>3.99</td>
<td>32.95</td>
</tr>
<tr>
<td>mem_latency</td>
<td>64 kB</td>
<td>3.01</td>
<td>5.91</td>
<td>96.05</td>
</tr>
<tr>
<td>mem_latency</td>
<td>512 kB</td>
<td>15.05</td>
<td>198.20</td>
<td>1217.35</td>
</tr>
<tr>
<td>mem_latency</td>
<td>1 MB</td>
<td>27.98</td>
<td>210.73</td>
<td>653.21</td>
</tr>
<tr>
<td>mem_latency</td>
<td>8 MB</td>
<td>218.95</td>
<td>221.57</td>
<td>1.2</td>
</tr>
<tr>
<td>mem_throughput</td>
<td>32 kB</td>
<td>1.00</td>
<td>3.50</td>
<td>248.91</td>
</tr>
<tr>
<td>mem_throughput</td>
<td>64 kB</td>
<td>1.01</td>
<td>4.17</td>
<td>314.68</td>
</tr>
<tr>
<td>mem_throughput</td>
<td>512 kB</td>
<td>1.34</td>
<td>17.65</td>
<td>1215.23</td>
</tr>
<tr>
<td>mem_throughput</td>
<td>1 MB</td>
<td>1.81</td>
<td>17.64</td>
<td>876.89</td>
</tr>
<tr>
<td>mem_throughput</td>
<td>8 MB</td>
<td>3.14</td>
<td>17.66</td>
<td>462.59</td>
</tr>
</tbody>
</table>

A.1.5 Memory Latency

The memory latency test tests the latency of memory operations by chasing a circular linked list in memory. The design of this linked list is important. Ideally, elements that are adjacent in the list should be far apart in memory. If they are in the same cache line, the CPU may hit on the second element, making the memory latency seem low than it actually is. Secondly, the whole linked list should be confined to a certain region of memory, so that the effects of locality can be accurately determined.

To accomplish this, first, a linked list where each element is visited in order is created in memory. Then, to destroy locality, millions of swaps are performed to scramble the order that the elements are visited in. To make these tests reproducible, and to avoid having to perform these swaps in simulation, scrambled linked lists are pre-generated and stored in files so they can be loaded into the benchmark. The benchmark’s loop just consists of eight \texttt{mov edx, [edx]} instructions.

A.2 Issues and Best Configuration

Table A.1 shows the results of running a simulation where the L1 and L2 cache latencies were each set to 1 cycle. Despite the first three register-based tests, and smaller memory based tests being accurate, the simulated CPU was much slower for larger memories. This is a problem in particular because changing the access latency of main memory also affects the simulation of the GPU, which has different performance characteristics. We believe this to be a bug with Multi2Sim’s memory hierarchy simulation, and we have notified Multi2Sim’s developers about the issue, although it has not been resolved.
Appendix B

The DistCL and Metafunction Tool Manual

B.1 Introduction

DistCL is a OpenCL implementation that automatically distributes OpenCL kernels across a cluster of GPU-equipped computers. Unlike other distributed OpenCL implementations, that simply make it easier to schedule work on remote devices, DistCL actually makes all the GPUs in the cluster appear as though they are one device. DistCL does this to increase the compute parallelism available for kernels without manually dividing them up. In particular, DistCL does not focus on memory. DistCL’s operation was described at the IEEE MASCOTS 2013 conference. The paper submitted to that conference is available at http://www.eecg.toronto.edu/~enright/MASCOTS13.pdf. In addition, any updated versions of DistCL and this document are available at http://www.eecg.toronto.edu/~enright/downloads.html.

This document describes how to build DistCL, link it to an OpenCL host program, and use DistCL to distribute that program’s kernels across a pre-existing MPI-based GPU cluster. It also gives insights into how to write metafunctions, which DistCL needs in addition to the host program. When DistCL distributes an OpenCL kernel, it partition’s that kernel’s NDRange into smaller pieces called Subranges. DistCL will pass a meta-function the position and size of a subrange, and that metafunction must identify which regions of each buffer that subrange will access. This information allows DistCL to correctly distribute data across the cluster. DistCL comes with a metafunction writing tool that helps programmers write metafunctions.

DistCL is released under the GNU LGPL version 3. Our hope is that you enhance and build upon this framework. As such, this document includes a section explaining how to include new kernel partitioning and scheduling algorithms into DistCL. That said, DistCL is released as-is. Its authors will not be supporting it.

B.1.1 Intended Audience

This document assumes the reader is familiar with OpenCL and the build process on POSIX systems - including concepts such as building and linking to static and dynamic libraries. It also assumes familiarity
B.2 Building DistCL

DistCL builds in Linux. In particular, DistCL requires the following to build:

- GCC g++ 4.5 or higher (a version that supports C++11 lambda expressions)
- Google protocol buffers (tested with version 2.4.1)
- OpenMPI (tested with version 1.4.3)
- GNU Portable Threads version 2.0.7
- An Underlying OpenCL Implementation

DistCL does not use autotools to build. Instead, the Makefile that comes with this package uses `pkg-config`. DistCL assumes the following packages will be available:

- `protobuf` for Google Protocol Buffers
- `pth` for GNU Portable Threads

In particular, DistCL does not need to be configured with the underlying OpenCL implementation at compile time. Instead, it loads the OpenCL library dynamically at run time based on the configuration file specified in section B.4.1. This package includes the OpenCL 1.2 headers, so DistCL should be able to link to any OpenCL implementation without issue.

To build DistCL, go to the root of this archive and type `make`. This should build the DistCL static library and the vector benchmark.

B.3 Linking DistCL to a Host Program

This package produces a static library called `libDistCL.a` which includes the compiled DistCL sources. In order to build a program that uses DistCL, it must link to this static library, `rt` (the Linux real time library) and `m` (the math library) as well as the aforementioned `protobuf` and `pth` libraries.

B.3.1 Linking Metafunctions

DistCL was designed to accept the source code for metafunctions and compile them at runtime, in parallel with the normal OpenCL kernel compilation work flow. To accept Metafunction sources, DistCL has an addition OpenCL call, `clLoadMetafunctionSource`. DistCL accomplishes this compilation by forking an instance of gcc with `-x c -fPIC -g -Wall -Werror -I <path to metafunction.h> -shared` as flags; however, some versions of MPI forbid MPI processes from forking, so this call has been disabled. Instead, to allow DistCL to link against metafunctions, those metafunctions must be compiled into a shared library called `meta.so` in the same directory as the binary, and it must be accessible to binaries running on every node in the network.
B.4 Running DistCL

DistCL uses MPI to run. It relies on MPI for both communication between peers, and for the launching of the DistCL program on each of the nodes involved in execution. The host program will only run on one of the nodes which DistCL calls the master. No kernels will execute on the master. Instead, DistCL distributes the kernels amongst the remainder of the nodes in the MPI job, which DistCL calls peers. Because of MPI’s programming model, identical processes are launched on each node, however when a peer makes a call to clCreateContext, it enters an event loop and stays there until the master forces it to terminate when the host program calls clReleaseContext.

To run a DistCL program, because of the master, the -np parameter passed to mpirun (or the number of peers submitted to the cluster’s system) must be one greater than the number of peers. For instance, to run the vector addition example from this package on 4 peers over a $2^{16}$-element array with verification enabled, in the bin directory, type:

```
$ mpirun -np 5 ./vector 65536 v
```

One process becomes the master and the rest become peers. Note that DistCL’s strategy of launching peers means that any work done before the call to clCreateContext is done on every node. This can be avoided by moving the call earlier in the program and does not affect DistCL’s potency as a research tool.

B.4.1 distcl.conf

DistCL needs to be configured with various runtime parameters in order to run. These parameters must be present in key-value pairs in a text file named distcl.conf placed in the working directory of the program. Each key value pair is on a separate line and has the key, a space, and the value. Neither keys nor values every have spaces in distcl.conf.
### B.5 Writing Metafunctions

Once DistCL has partitioned an NDRange into Subranges, it calls programmer-supplied meta-functions to determine what regions of memory each subrange will access. Metafunctions do this by relating the size and position of a subrange to the size and position of the parts of each buffer that the kernel will access. There are two main metafunctions for each kernel: a read metafunction, used for reading and a write metafunction, used for writing. There is also a third metafunction that simply tells DistCL whether each parameter is a buffer or not. This is required because the `clGetKernelArgInfo` call that reveals information about a parameter is not available in older OpenCL implementations. Metafunctions are associated with their corresponding OpenCL functions based on their name. The table below specifies the naming convention used where `kernel-name` is the name of the kernel.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Naming Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Metafunction</td>
<td><code>is_buffer_range_read_kernel-name</code></td>
</tr>
<tr>
<td>Write Metafunction</td>
<td><code>is_buffer_range_write_kernel-name</code></td>
</tr>
<tr>
<td>Buffer Identification</td>
<td><code>is_param_buffer_kernel-name</code></td>
</tr>
</tbody>
</table>

The read and write meta-functions are passed the same set of parameters. They are summarized in the table below, in the order they should appear in a function. In addition to the table, the example DistCL programs also have metafunctions that can serve as templates.
Appendix B. The DistCL and Metafunction Tool Manual

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>const void **params</td>
<td>An array of pointers to the immediate (non-buffer) arguments passed to the kernel.</td>
</tr>
<tr>
<td>const size_t *global</td>
<td>The global size of the NDRange. The metafunction is assumed to know how many dimensions there are, and thus, the number of elements in global.</td>
</tr>
<tr>
<td>const size_t *subrange</td>
<td>The size of the current subrange in work-items.</td>
</tr>
<tr>
<td>const size_t *local</td>
<td>The local size in work-items.</td>
</tr>
<tr>
<td>const size_t *subrange_offset</td>
<td>The start of the subrange in work-items.</td>
</tr>
<tr>
<td>unsigned int param_num</td>
<td>The argument index of the buffer to consider.</td>
</tr>
<tr>
<td>size_t start</td>
<td>The starting position of the buffer to consider.</td>
</tr>
<tr>
<td>size_t *next_start</td>
<td>The next part in the buffer that should be considered.</td>
</tr>
</tbody>
</table>

In addition to accepting these parameters, the read and write metafunction return an integer which specifies whether the memory between `start` and `*next_start` is accessed or not. For each buffer-subrange pair, DistCL iteratively calls both the read and write metafunctions. Initially, DistCL calls the metafunction with `start` set to zero. The metafunction must determine whether the first byte of the buffer (at index zero) is accessed and, in addition, set `*next_start` to the point in the buffer where this condition changes. DistCL will then call the metafunction again with `start` now set to the value returned in `*next_start`. It is ok if the metafunction’s return value remains the same, but returning the same value twice does imply that `*next_start` could have been set to the second value originally. This iterative process continues until the meta-function sets `*next_start` to the size of the buffer, in bytes. In this approach the metafunction and DistCL walk through the buffer, at a pace controlled by the metafunction, and mark sections of it as either accessed or not accessed.

### B.5.1 Using `require_region`

It can be cumbersome to try to sort out regions of buffers manually, particularly when the buffers hold multidimensional arrays. For linear arrays and arrays organized in row-major-order, DistCL provides a helper function that automates the iterative process of identifying accessed and not-accessed regions. This function is called `require_region`. To use `require_region`, a metafunction must include the header `metafunction.h` which is included in this package. The parameters passed to `require_region` are analogous to those passed to the read and write metafunctions. They are summarized in the table below.
### Appendix B. The DistCL and Metafunction Tool Manual

#### C Declaration

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int dim</td>
<td>The dimensionality of the array.</td>
</tr>
<tr>
<td>const size_t *total_size</td>
<td>The total size of the array, in elements, in each dimension.</td>
</tr>
<tr>
<td>const size_t *required_start</td>
<td>The start of the required region, in elements, in each dimension.</td>
</tr>
<tr>
<td>const size_t *required_size</td>
<td>The size of the required region, in elements, in each dimension.</td>
</tr>
<tr>
<td>size_t start</td>
<td>The current linear address being compared against the required region.</td>
</tr>
<tr>
<td>size_t *next_start</td>
<td>The next linear address to compare.</td>
</tr>
</tbody>
</table>

#### B.5.2 The Tool

Writing metafunctions involves capturing the relationship between the parameters a work-item is passed, the work-item’s IDs and the memory addresses it accesses. Therefore, writing metafunctions for unfamiliar kernels involves discovering the memory access patterns of that kernel. To help this process along, and to help validate the correctness of metafunctions, this package includes a tool, separate from the DistCL code base, that analyses the access patterns of OpenCL kernels and compares them to metafunctions, with minimal programmer intervention. This tool makes writing metafunctions faster and less error-prone.

This tool acts as a host program to an arbitrary OpenCL kernel. As such, it reads a JSON-format configuration file that describes to the tool how many buffers to allocate, what size they should be, what immediate parameters to pass to the kernel, the geometry to invoke the kernel with, and the information to pass to the metafunction. The tool executes the kernel, passing it zeroed buffers, and compares the changes in each buffer to the regions that the metafunction said would be accessed. If the metafunction did not accurately predict the changed regions, the tool prints a summary of the missing or extraneous regions of each miss-matching buffer.

When using the tool to detect writes, the user should modify the source of the kernel to prevent false negatives that would occur if the kernel were to write a value of zero back to the buffer. The easiest way to do this is to edit the kernel so that each write just writes a value of 1 instead of whatever computed value it would usually write. Similarly, the tool cannot detect reads, so for validating read metafunctions, the user should disable all writes and change all reads into writes of a value of 1. Keep in mind that increment operations like `++` and `+=` are both reads and writes, and hence, must be accounted for in both versions of the kernels.

The JSON file passed to the tool contains a JSON object with 5 fields. A description of each field follows

#### Constants

The constants section is an array of constants. Each constant is an object with a name and value field. The name can be any string that is at least two characters in length that does not start with a number. The value is an integer. It may be an integer literal or an RPN expression. RPN expressions are arrays that contain integer literals, `+`, `-`, `*`, `/` and `%` operators, and previously-declared constants.
For instance, the listing below shows a constants section that defines three constants. ARRAY_COUNT is 128, ELEMENT_SIZE is 4, and ARRAY_SIZE is their product, 512. Constants and RPN expressions, which can be used throughout the rest of the configuration file, can be used to ensure that important relationships between buffer sizes and NDRanges stay intact while testing the kernel over various input sizes, and it can make it easier to change which subrange is being tested.

```json
"constants": [
  {"name": "ARRAY_COUNT", "value": 128},
  {"name": "ELEMENT_SIZE", "value": 4},
  {"name": "ARRAY_SIZE", "value":
    ["ARRAY_COUNT", "ARRAY_SIZE", "*"]
}]
```

Buffers

The buffers section is an array of buffer declaration objects. These buffers can be passed to the kernel and considered by the metafunction. Each buffer declaration has a name field, a string, which can be used to refer to it later, and a buffer_size field that specifies the buffer's size in bytes. Buffer declarations have a gran field, which specifies the granularity, in bytes, at which changes should be tracked. For instance, if the buffer contains 4-byte integers, the granularity should be set to 4, so that the tool knows that a change in any one of four consecutive bytes represents a change in all of them, otherwise, the tool might mistakenly conclude that a metafunction predicted extra changes because some bytes within a single element stayed the same.

Finally, buffer declarations also have a geometry field. This field is an array of integer expressions (integer literals, constants or RPN expressions). The geometry field represents the logical organization of the array, particularly if it is a multidimensional array. This affects how the tool reports missing and extra regions of change back to the user. For instance, different work-groups within a kernel that does a matrix operation may operate on rectangular regions of the matrix. The geometry field allows the tool to report back in terms of rectangular regions instead of just sections of linear buffer. The numbers in the geometry field are in terms of array elements, so product of the numbers in the geometry field and the gran field should equal the buffer_size field.

For instance, the buffers section below declares two buffers. The first has 128 4-byte elements which are logically accessed as a linear array. The second has 256 4-byte elements, which are accessed as a 16 × 16 element array.

```json
"buffers": [
  {"name": "linear_buffer",
   "gran": 4, "size": "ARRAY_SIZE",
   "geometry": ["ARRAY_COUNT"]},
  {"name": "square_buffer",
   "gran": 4, "size": 1024,
   "geometry": [16, 16]}
]
```
### Kernel

The kernel section specifies what kernel to run. It is an object with two fields: `file`, the source file for the kernel and `name` the name of the kernel. For instance, below, the kernel section specifies that a kernel named `test_kernel` in `kernel.cl` be run.

```json
"kernel": {
  "file": "kernel.cl",
  "name": "test_kernel"
}
```

### Arguments

The arguments section describes the arguments that should be passed to the kernel. It is an array of argument objects. The order that the argument objects appear corresponds to the order that they will be passed to the OpenCL kernel. Each argument object has a `type` field which has three four valid values: `int` (for immediate integers), `float` (for immediate floating point values), `local` (for local memory), and `buffer` for global memory. If the type is `buffer`, the only other value in the argument object is `name` which should be set to the name of the buffer as declared in the buffer section. If the type is `local`, the only other value is `size` which is an integer expression representing the size of the array in bytes. Finally, if the type is `int` or `float`, the only other field is `value` an expression representing the value of the immediate parameter.

The argument section below specifies three arguments. The kernel will be passed the `linear_buffer` buffer declared above as its first parameter. Its second parameter is a local memory buffer 4 bytes in size, and its third parameter is the number 42.

```json
"arguments": [
  {"type": "buffer", "name": "linear_buffer"},
  {"type": "local", "size": "ELEMENT_SIZE"},
  {"type": "int", "value": 42}
]
```

### Invocation

The invocation section specifies the parameters sent to `clEnqueueNDRangeKernel`. It is an object with three fields: `offset`, `global` and `local`, corresponding to the `global_work_offset`, `global_work_size` and `local_work_size` parameters of `clEnqueueNDRangeKernel`, respectively. Each of these three field is an array of integer expressions. Note that in order to simulate the execution of different subranges, the user must tweak the value of `global` and `offset` manually. Depending on the work-item functions the kernel calls, the user might also have to edit the source of the kernel so that calls like `get_global_size()` and `get_group_id()` work as expected. DistCL applies code transformations to this effect automatically, but the tool does not modify the kernel at all.

The invocation section below specifies a one-dimensional kernel invocation of 128 work-items grouped into work-groups of 8.

```json
"invocation": {
```
"offset": [0],
"global": ["ARRAY_COUNT"],
"local": [8]}

Metafunction

The metafunction section is an object that contains information about what metafunction to run and what information to pass it. Unlike DistCL, the tool only evaluates a single subrange every time it is run. To test every subrange in a kernel, the tool must be called several times, with different values in the invocation and metafunction sections.

The metafunction section has a file field which contains the name of the shared library that has the metafunction it in, a name filed with the name of the metafunction, and global, subrange_offset, subrange, and local fields, all arrays of integer expressions, that should be passed to the metafunction. Note that, in general, the subrange value in the metafunction section corresponds to the global field in the invocation section and the subrange_offset field to the global work_offset field. Again, this is because the tool does not divide the kernel up into subranges, but metafunctions operate on subranges exclusively.

The metafunction section below specifies name and location of the metafunction, that it has a global size of ARRAY_COUNT^2, and that the subrange currently being tested has an offset of 0, a size of ARRAY_COUNT, and a local size of 8 - values which correspond to the invocation section.

"metafunction": {
  "file": ".\\file\meta.so",
  "name": ".buffer-range-read-test-kernel",
  "global": [["ARRAY_COUNT", "ARRAY_COUNT" "*"]],
  "subrange_offset": [0],
  "subrange": ["ARRAY_COUNT"],
  "local": [8]
}

When a configuration file with all these sections present is passed to the tool, the tool will run the kernel and metafunctions and compare their output. For each buffer, it will print the whether there were any omitted or extra regions specified by the metafunction. These regions will be in terms of the geometry specified for that buffer.

B.5.3 Building the Tool

The tool is in the tool directory of this archive. To build it, type make. The tool requires a library called cJSON in order to parse its JSON configuration file. cJSON can be downloaded from http://sourceforge.net/projects/cjson/. The Makefile assumes a pkg-config package called cJSON exists for the cJSON library. Unlike DistCL, cJSON does link to the underlying OpenCL implementation at compile-time; therefore, the variables OPENCL_INC and OPENCL_LIB need to be set in the Makefile accordingly.

With this tool, it is easy for programmers to quickly write and verify metafunctions.
B.6 Enhancing DistCL

DistCL was written to be extensible. Some of the algorithms used in DistCL, particularly those that could be interesting for future research, have been strategized so they can be easily exchanged with other algorithms.

B.6.1 Partition Strategies

All partition strategies are descendents of PartitionStrategy which is defined in partition-strategy.h. The partition method of a PartitionStrategy is given a target number of subranges, and the group count in each dimension, and it must populate a vector of objects, where each object holds the start and offset of a subrange in work-groups. To change the partition strategy used, change the constructor being called on line 55 of kernel-execution.cpp.

B.7 Limitations

This section contains a few things the reader should know about when using DistCL.

B.7.1 With AMD 5000 Series GPUs

As of AMD APP SDK 2.7, at least in 32-bit Linux, there is a bug involving how AMD’s OpenCL implementation handles multiple mapping operations enqueued to an AMD 5000 Series GPU. As a result, these GPUs do not work with DistCL. The AMD CPU device, on the other hand, works perfectly.

B.7.2 Completeness of OpenCL Implementation

According to its documentation, OpenMPI does not robustly support the MPI_THREAD_MULTIPLE programming model; therefore, each DistCL process runs in a single OS thread, using user-schedulable contexts to enable multiple IO operations to run concurrently. This means that all operations in DistCL are blocking, and there is no event model. Also, not all OpenCL calls are implemented. The file mastercl.cpp contains all of the implemented OpenCL calls.
Bibliography


