Cache Memory Access Patterns in the GPU Architecture

Yash Nimkar
ypn4262@rit.edu
Cache Memory Access Patterns in the GPU Architecture

Yash Nimkar
Cache Memory Access Patterns in the GPU Architecture
YASH nimkar

Committee Approval:

Dr. Sonia Lopez Alarcon
Thesis Advisor
Department of Computer Engineering
Rochester Institute of Technology

Dr. Amlan Ganguly
Department of Computer Engineering
Rochester Institute of Technology

Dr. Roy Melton
Department of Computer Engineering
Rochester Institute of Technology
Acknowledgments

I would like to thank my Thesis Advisor, Dr. Sonia Lopez Alarcon for all her help and support. This has been a continual learning experience for me and my thesis would not have been the same without Dr. Sonia Lopez’s guidance and direction. I would also like to thank Dr. Amlan Ganguly and Dr. Roy Melton for being on my Master’s thesis committee.

I would like to thank Richard Tolleson and Richard Flegal from the Department of Computer Engineering at RIT for all IT related help especially while setting up my research VMs and computers.

I would like to thank Paul Mezzanini, Sidney Pendelberry and Emilio Del Plato from the Research Computing department at RIT for all their help in relation to the research computing clusters and VMs that were initially used for my research.

I would like to thank Xun Gong who is a Multi2Sim developer at Northeastern University for all his help with the Multi2Sim simulator and for setting up and customizing the NVIDIA Kepler model on the Multi2Sim.

I would like to thank our labs ex-students or alumni, Ben Johnstone and Harshdeep Chawla, for their help as I used their research as a base to understand the GPU architectures and setup the GPU simulator.

Lastly, I would like to thank my family members and close friends for their continued support along the way. This would not have been possible without their support and encouragement.
Abstract

Data exchange between a Central Processing Unit (CPU) and a Graphic Processing Unit (GPU) can be very expensive in terms of performance. The characterization of data and cache memory access patterns differ between a CPU and a GPU. The motivation of this research is to analyze the cache memory access patterns of GPU architectures and to potentially improve data exchange between a CPU and GPU. The methodology of this work uses Multi2Sim GPU simulator for AMD Radeon and NVIDIA Kepler GPU architectures. This simulator, used to emulate the GPU architecture in software, enables certain code modifications for the L1 and L2 cache memory blocks. Multi2Sim was configured to run multiple benchmarks to analyze and record how the benchmarks access GPU cache memory. The recorded results were used to study three main metrics: (1) Most Recently Used (MRU) and Least Recently Used (LRU) accesses for L1 and L2 caches, (2) Inter-warp and Intra-warp cache memory accesses in the GPU architecture for different sets of workloads, and (3) To record and compare the GPU cache access patterns for certain machine learning benchmarks with its general purpose counterparts.
# Contents

Signature Sheet i

Acknowledgments ii

Dedication iii

Abstract iv

Table of Contents v

List of Figures vii

List of Tables ix

1 Introduction 1
   1.1 Motivation ................................................. 2
   1.2 Objective .................................................. 4

2 Background 6
   2.1 Simulators ..................................................... 6
   2.2 Related Work .................................................. 7

3 Multi2Sim 12
   3.1 Setting up the simulator on an Ubuntu OS .................. 13
   3.2 AMD Southern Islands (Radeon) GPU Architecture .......... 14
   3.3 NVIDIA Kepler GPU Architecture .......................... 19

4 Analyzing Cache Memory Access Patterns 24
   4.1 L1 and L2 Cache Hit Ratios ................................ 25
      4.1.1 AMD Southern Islands .................................. 25
      4.1.2 NVIDIA Kepler .......................................... 26
   4.2 MRU and LRU Cache Counters for Temporal Locality ....... 27
   4.3 Inter-warp and Intra-warp Cache Locality .................. 33

5 Results 39
   5.1 Cache Hit Ratios .............................................. 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2 MRU and LRU Temporal Locality Results</td>
<td>41</td>
</tr>
<tr>
<td>5.3 Inter-warp and intra-warp Locality Results</td>
<td>48</td>
</tr>
<tr>
<td>5.4 Comparing the CPU and GPU results</td>
<td>54</td>
</tr>
<tr>
<td>6 Machine Learning Benchmarks using CUDA</td>
<td>58</td>
</tr>
<tr>
<td>6.1 Design Methodology</td>
<td>58</td>
</tr>
<tr>
<td>6.2 Results</td>
<td>61</td>
</tr>
<tr>
<td>6.2.1 L1 and L2 cache hit ratios</td>
<td>61</td>
</tr>
<tr>
<td>6.2.2 MRU and LRU Temporal Locality Results</td>
<td>62</td>
</tr>
<tr>
<td>6.2.3 Inter-warp and Intra-warp Locality Results</td>
<td>65</td>
</tr>
<tr>
<td>7 Conclusion</td>
<td>70</td>
</tr>
<tr>
<td>7.1 Future Work</td>
<td>72</td>
</tr>
<tr>
<td>7.1.1 Unified Memory Model</td>
<td>72</td>
</tr>
<tr>
<td>7.1.2 Machine Learning Benchmarks on the GPU</td>
<td>72</td>
</tr>
<tr>
<td>A Code Listings</td>
<td>78</td>
</tr>
</tbody>
</table>
## List of Figures

3.1 Block diagram of the AMD 7970 architecture [14] .................................. 16
3.2 Block diagram of the Compute Unit of the AMD 7970 architecture [14] 17
3.3 Block diagram of the NVIDIA Kepler architecture [1] ....................... 19
3.4 Kepler GPU Memory Architecture [1] .............................................. 20
3.5 NVIDIA Kepler Streaming Multi-processor (SM) architecture [1] .... 21


5.1 Cache Memory Hit Ratios for the OpenCL benchmarks ..................... 40
5.2 Cache Memory Hit Ratios for the CUDA benchmarks ....................... 40
5.3 OpenCL benchmark results for the AMD SI architecture for L1 cache 42
5.4 CUDA benchmark MRU Results for L1 Cache of the NVIDIA Kepler architecture .......................................................... 43
5.5 OpenCL benchmark results for the AMD SI architecture for L2 cache 45
5.6 CUDA benchmark MRU Results for L2 Cache of the NVIDIA Kepler architecture .......................................................... 46
5.7 OpenCL L1 vector and L2 cache intra-warp and inter-warp access percentages .......................................................... 49
5.8 CUDA L1 vector and L2 cache intra-warp and inter-warp access percentages .......................................................... 50
5.9 OpenCL L1 vector intra-warp and inter-warp hits and misses .......... 52
5.10 OpenCL L2 intra-warp and inter-warp hits and misses ..................... 52
5.11 CUDA L1 intra-warp and inter-warp hits and misses ...................... 53
5.12 CUDA L2 intra-warp and inter-warp hits and misses ...................... 53

6.1 Machine Learning benchmark results for the NVIDIA Kepler architecture for L1 cache .......................................................... 63
6.2 Machine Learning benchmark results for the NVIDIA Kepler architecture for L2 cache .......................................................... 64
6.3 L1 and L2 cache inter-warp and intra-warp access percentages for Machine Learning benchmarks .......................................................... 66
6.4 L1 intra-warp and inter-warp hits and misses for Machine Learning benchmarks .......................................................... 67
LIST OF FIGURES

6.5 L2 intra-warp and inter-warp hits and misses for Machine Learning benchmarks ............................................. 67

7.1 NVIDIA’s Unified Memory Model [27] ............................... 73
List of Tables

3.1 Dependency list for the Multi2Sim simulator ................................. 13
3.2 Difference in terminology between OpenCL and CUDA ..................... 15
3.3 AMD Southern Islands emulated GPU configuration on Multi2Sim [10, 14] ................................................................. 18
3.4 NVIDIA Kepler emulated GPU configuration on Multi2Sim [1, 14] ..... 22
5.1 OpenCL Benchmark MRU results for L1 Cache (per module) of the AMD SI architecture ......................................................... 42
5.2 CUDA Benchmark MRU results for L1 Cache of the NVIDIA Kepler architecture ................................................................. 43
5.3 OpenCL Benchmark MRU results for L2 Cache of the AMD SI architecture ................................................................. 45
5.4 CUDA Benchmark MRU results for L2 Cache of the NVIDIA Kepler architecture ................................................................. 46
5.5 Inter-warp and Intra-warp Access Percentages for L1 and L2 cache for OpenCL benchmarks ......................................................... 49
5.6 Inter-warp and Intra-warp Access Percentages for L1 and L2 cache for CUDA benchmarks ......................................................... 50
5.7 MRU Results for the CPU [11] .................................................... 54
5.8 Comparison of MRU results for the CPU and the GPU [11] .............. 55
5.9 Cache Hit Rates for the CPU [10] ............................................... 56
5.10 Comparison of Cache Hit Ratios for the CPU and the GPU ............. 57
6.1 Defined Machine Learning algorithms in CUDA ............................... 59
6.2 L1 and L2 Hit Ratios for Machine Learning Benchmarks .................. 61
6.3 MRU Results for the Machine Learning benchmarks on the NVIDIA Kepler architecture for L1 cache ..................................... 62
6.4 MRU Results for the Machine Learning benchmarks on the NVIDIA Kepler architecture for L2 cache ..................................... 63
6.5 Inter-warp and Intra-warp Access Percentages for L1 and L2 cache ... 66
6.6 Comparison of the warp locality results for the two sets of CUDA benchmarks ................................................................. 68
Graphic Processing Units (GPUs) are regarded as one of the standards when it comes to high processing power and parallel execution. As the world advances in technology, the applications of GPUs range from machine learning and computer vision to general purpose computing and more. Although the use of GPUs is growing greatly in industry, there is very little knowledge and understanding of GPUs and their relationship with cache, as well as the communication between a CPU and a GPU.

Data level parallelism is achieved when multiple processors work on the same code to execute the results in parallel therefore improving the performance. GPUs enabled programmers to run multiple applications using data level parallelism by executing multiple threads in parallel. This was done using the defined hardware characteristics, such that each thread will execute the same kernel functionality. This execution model is one level higher than the Single Instruction Multiple Data (SIMD) model and is called Single Instruction Multiple Thread (SIMT). In 2007, NVIDIA launched their first GPU accelerators with supporting hardware to improve the performance of high end applications. GPU accelerators are mainly used in heterogeneous multi-processor systems, where the GPU works in tandem with the CPU to improve or accelerate performance.

In a heterogeneous multi-processor chip, the CPU and GPU are the two main processors. The CPU is the primary host while the GPU is the device that works
off the primary host (CPU). The CPU’s cache memory access patterns are very well
documented, but the same cannot be said about the GPU. There has not been enough
research done in the GPU field to understand how the GPU accesses its cache and to
understand the cache memory access patterns of the GPU.

1.1 Motivation

One of the primary objectives of this research is to understand the GPU cache memory
access patterns and in turn improve the communication between GPU and CPU in a
heterogeneous multi-processor chip. A GPU simulator called Multi2Sim was used to
simulate various benchmarks for two sets of GPU architectures — NVIDIA’s Kepler
and AMD’s Radeon.

The development of GPUs has enabled high data parallelism using multiple, easy
to use, and flexible programming models. Modern GPUs can be used for general
purpose processing and are part of a heterogeneous multi-processor chip along with
the CPU. Modern GPUs offer greater computational power than the CPU and are
used to run applications in parallel taking advantage of the application’s data level
parallelism. GPUs are Single Instruction Multiple Data (SIMD) architectures. This
enables them to offer greater performance with low latency and overhead.

In a heterogeneous multi-processor system, different tasks are assigned to and
handled by different processing cores based on the individual characteristics of the
processor and the task assigned to it. This improves performance and saves power,
effectively using the available resources on the system or chip with an even work load
for the processors.

Understanding the memory access patterns of the GPU can lead to improving the
communication and memory management between the CPU and the GPU. The CPU
cache memory definitions and access patterns have been studied and identified for
various releases of CPUs [11, 12, 13]. Savaldor Petit et al. [11] talk about percentage
of cache hits along the multiple lines for each set of the CPU’s cache. Various CPU benchmarks were run to identify the temporal locality of lines for each set of CPU cache. The first MRU line shows a very high percentage of cache hits between 85% to 95%, depending on the benchmarks being tested. The other 15% to 5% of hits were then distributed among the other MRU lines. The results showed a very high percentage of hits on the Most Recently Used (MRU) block or MRU0 line, and this value was recorded to be 92% on average. This served as a motivation to find the MRU cache access patterns of the GPU to see if the GPU showed similar results to those of the CPU.

Studying the GPU cache locality and cache memory access patterns can further help understand the working of the GPU for specific high-performance benchmarks like those used for machine learning. Most of the machine learning workloads are executed using pre-defined libraries or frameworks like Tensorflow. These machine learning workloads are heavily reliant on a GPU for high data level parallelism and high computational power. To the best of the author’s knowledge, machine learning algorithms or workloads have not been tested on a simulator level to understand how the GPU works with these different workloads and how this impacts its cache memory. By running some machine learning benchmarks on the Multi2Sim simulator for the GPU, the GPU cache behavior can be studied for different machine learning workloads.

This research can also pave the way for a new approach, currently being pursued by AMD and NVIDIA, called the Unified Memory Model[27] between the CPU and the GPU. Based on this model, the CPU and GPU share a dedicated unified memory block located in Dynamic Random Access Memory (DRAM), where the GPU can access the data directly from DRAM without involving or depending on the CPU.
1.2 Objective

The main objective of this research is to learn more about the cache memory access patterns of the GPU. These results will help identify specifics about GPU cache operations and can be used to improve data exchange between the GPU and the CPU in a heterogeneous multi-processor chip. Multi2Sim [6] was used to perform a wide range of simulations on two different GPU architectures. Multi2Sim emulates the AMD’s Radeon GPU architecture as well as NVIDIA’s Kepler GPU architecture. A set of OpenCL and CUDA benchmarks was used to test both GPU architectures for different workloads.

The first objective was to setup the Multi2Sim simulator on an Ubuntu VM with all the required dependencies. The simulator was compiled to run the OpenCL and CUDA benchmarks to test the working of the GPU. To modify the GPU emulator code defined in the Multi2Sim simulator, first there was a need to understand how the AMD and NVIDIA GPUs were defined in hardware and how they were emulated in code on the simulator. After understanding the architectural code and the memory code, three metrics were defined to analyze GPU cache behavior: cache hit ratios, MRU temporal locality and inter-warp/intra-warp locality.

The caching policy was studied to add specific cache counters to keep track of the most recently used (MRU) and least recently used (LRU) blocks in the GPU cache. A set of MRU and LRU counters were implemented to study the temporal locality of the GPU cache. These MRU counters determine the number of cache accesses made for each MRU line in a multi-way set associative cache. In the GPU architecture, L1 and L2 cache blocks were defined using different set associativity. This associativity determined how many ways or lines were used to split the cache. For example, if L1 cache was defined a 4-way set associative cache, then each set was defined using 4 ways or lines. MRU0 represents the most recently used line and
MRU4 would represent the least recently used line. The caching policy uses these multiple lines to add, replace, and remove blocks of memory for each set in cache. The MRU counters were updated during the GPU execution every time a block got accessed in cache at that very location on the MRU line. The respective MRU line counter was incremented. These MRU counters were used to record the temporal locality of the GPU in L1 and L2 cache. These GPU MRU results were compared to those of the CPU to see if the GPU showed similar values and access patterns to those of the CPU.

A set of inter-warp and intra-warp cache counters were implemented to understand the cache locality for different sets of workloads for the defined GPU architectures. The intra-warp access counters represent the accesses made within each warp while the inter-warp access counters represent the accesses made across various warps in the overall GPU architecture. This set of counters can help analyze the cache memory access patterns of a GPU. The L1 and L2 intra hits were defined as the number of L1 and L2 data cache hits that resulted from intra-warp locality, respectively. Contrarily, the L1 and L2 inter hits were the data cache hits that resulted from inter-warp locality. This helped understand the cache locality of the GPU architecture for different sets of workloads.

Furthermore, some machine learning benchmarks were defined in CUDA and tested on the GPU architecture to analyze how the GPU reacts to different machine learning workloads. These benchmarks were defined using math algorithms that are most commonly used by Convolution Neural Networks (CNN) and Deep Neural Networks (DNN). The cache hit ratios, MRU temporal locality and the inter-warp/ intra-warp locality were recorded and compared against the general purpose CUDA benchmarks.
Chapter 2

Background

2.1 Simulators

A large portion of the work in this thesis focused on setting up the right GPU simulator. GPGPU Sim [1, 2] was the most popular simulator a few years ago, and most of the research in the GPU field was performed using this simulator. The advantages of this simulator were that it was easy, convenient and had all the latest GPU architectures modeled to replicate GPUs in hardware, at that time. But over the years, this simulator has not been updated, and this results in many compatibility issues in terms of the latest versions of CUDA, OpenCL, GCC compilers and Python. This meant that this simulator could not be used to perform experiments for the latest GPU architectures, to study recent technology trends. This GPGPU simulator recently merged with Gem5 [3], but the some of the dependencies still have not been updated. Hence neither simulator (GPGPU Sim or Gem5-gpu) was chosen for these experiments.

Barra [1, 4] and Ocelot [1, 5] are some of the other GPU simulators that were considered for this research. The drawback of Barra is that it runs CUDA applications and benchmarks for NVIDIA GPUs only and does not have the capability to run any OpenCL applications and benchmarks for the AMD series of GPUs. In addition to that, Barra can be used to run CUDA benchmarks on an assembly level using an
CHAPTER 2. BACKGROUND

Instruction Set Architecture (ISA) and not on a Parallel Thread Execution (PTX) level. Unlike Barra, Ocelot supports both CUDA and OpenCL benchmark on a ISA and PTX level. Since Ocelot has not been maintained and updated to the latest SDK, GCC, and CUDA releases and hence was not chosen for this research.

Multi2Sim [1, 6] was chosen as the main simulator for this research as it was the most up-to-date simulator with the latest dependencies and accurate functionality. Multi2Sim is easy to setup and use, and it is very well documented and supported, being one of the newer open source simulators out there. Older versions of Multi2Sim supported only CPU architecture simulations. But recent simulator released have helped incorporate GPU architecture emulation and simulation for AMD’s Radeon Southern Islands series and for NVIDIA’s Kepler GPU series. This is another advantage as this simulator can be used to run both OpenCL and CUDA benchmarks to test the AMD Radeon and NVIDIA Kepler GPU architectures.

2.2 Related Work

In this new generation of high performance computing, the GPUs have taken up a new distinction in terms of processing power using parallel thread execution. The technology has come a long way since the initial release of the GPU in terms of its architecture and how it works communicates with the CPU.

Lee et al. [7] proposed a method to improve GPU performance by updating the warp scheduling policy within a GPU architecture. This method also used a simulator and tested the scheduling policies of the GPU, like the round-robin and greedy instruction issue scheduling policies and patterns. Most of the related research in this field had focused on static scheduling methods for all workloads, and this was the first time a dynamic or adaptive instruction issue scheduling policy was suggested. The proposed method was called instruction-issue pattern based adaptive warp scheduler (iPAWs). This scheduling policy switched between a greedy and
round-robin scheduler based on the type of workload being run on the GPU. Various
sets of benchmarks were run to simulate large diverse workloads on an NVIDIA
GPU using CUDA. The proposed adaptive warp scheduling method was observed to
be optimal than the static greedy or round-robin scheduling methods. This paper
accurately described the use of a GPU simulator for results and defined the GPU
modifications well in terms of the GPU architecture. This paper served as a baseline
for the use of a GPU simulator for a similar research in the GPU field, but this time
to analyze the memory access patterns of a GPU. This paper also presented GPU
results for warp locality mainly, inter-warp and intra-warp statistics in L1 and L2
cache. This helped in understanding the warp locality for different sets of workloads
for both the Greedy, Round-Robin scheduling policies and iPAWS (as suggested by
the paper).

Mei et al. [8] talk about the GPU memory hierarchy through micro-benchmarking.
This paper proposes a new micro-benchmarking approach to test the GPU cache
memory components for three NVIDIA GPU models: Kepler, Fermi and Maxwell.
The memory components studied are data cache, texture cache, and translation look-
aside buffer (TLB). This paper goes on to introduce the proposed micro-benchmarking
approach to study the memory access patterns of the GPU which are comprised of
cache memory units with the Least Recently used (LRU) caching policy. The memory
access patterns of the GPU and the memory statistics, like number of accesses, number
of hits, number of misses, hit-ratio, etc. were recorded for the new proposed method
of fine grained benchmarking method. The proposed method proved to enhance the
memory capability and capacity for all three generations of the NVIDIA GPUs, more
so for Kepler and Maxwell than Fermi. Furthermore, it enhanced the performance
of the GPU by reducing the latency for shared memory of the GPU caused due to
bank conflicts. Some of the drawbacks or limitations of this approach included low
utilization rate of GPU hardware resources and unbalanced bandwidth values.
Johnstone et al. [9] talk about the bandwidth requirements of the GPU cores to determine the appropriate choice of an interconnect between the GPU and CPU in a heterogeneous multi-core chip. This paper completely focused on running GPU simulations for different benchmarks and workloads using GPGPU simulator. The interconnect properties were found to be dependent on the performance of the GPU, and how the bandwidth affected the GPU performance. The GPU architectures and the use of GPGPU simulator were the main takeaways from this paper in relation to this research area.

Choo et al. [10] talk about analyzing and optimizing GPU cache memory performance for different computational workloads using Multi2Sim. This paper used the AMD Southern Islands (SI) GPU architecture defined on MultiSim to run various computational workloads and observe the L1 and L2 cache hit rates and behavior. The L1 and L2 cache hit ratios were compared for the CPU and the GPU. The CPU showed much higher cache hit ratios than the GPU as expected, as the CPU focuses more on the memory hierarchy during execution to increase performance. On the other hand, the GPU relies more on parallel execution to process larger workloads such that the same set of instructions can be executed across multiple threads to increase productivity and performance. Since this paper used Multi2Sim, it was used as a reference point for any AMD SI GPU simulations performed during this thesis. The cache hit ratios recorded from this thesis were also compared to the results recorded from this paper for reference. There was no research in terms of caching policies or identifying the most recently used blocks in cache memory. This paper completely focuses on improving the GPU memory performance by proposing two methods: shared L1 vector data cache and clustered work-group scheduling. Both of these methods were executed for different workloads, and the performance improvements were recorded.

Furthermore, the CPU cache memory definitions and access patterns have been
studied and identified for various releases of CPUs [11, 12, 13]. Savaldor Petit et al. [11] investigated the temporal locality of a multi-way set associative cache by recording the percentage of cache hits along the multiple lines for each set of the CPU’s cache. This helped analyze the power consumption and performance of the CPU for the different cache lines using the current caching policies. A new drowsy cache policy was proposed to demonstrate a good balance between performance and power consumption. The experiments were performed using the HotLeakage simulator and the Spec2000 benchmark suite for CPUs. This drowsy cache policy was then compared to two existing caching policies, Most Recently Used On (MRO) and Two Most Recently Used On (TMRO), to compare the performance and power statistics. This research involved going through each Most Recently Used (MRU) block or line in cache and the power values, and hit percentages were recorded for each line. Various CPU benchmarks were run to identify the temporal locality of lines for each set of CPU cache. The first MRU line shows a very high percentage of cache hits between 85% to 95%, depending on the benchmarks being tested. The other 15% to 5% of hits were then distributed among the other MRU lines. The results showed a very high percentage of hits on the Most Recently Used (MRU) block or MRU0 line, and this value was recorded to be 92% on average. This paper serves as a good reference point for the MRU cache access patterns for the CPU. This served as a motivation to find the MRU cache access patterns of the GPU to see if the GPU showed similar results to the CPU results recorded by this paper.

Kumar et al. [12] talk about the modern CPU cache memory hierarchy and perform cache analysis for various cache replacement policies in CPU cache memory. The latest generations of processors were compared to find the different sets of factors that affect the performance and utilization of cache memory. Furthermore, the cache replacement policies were studied and analyzed using performance analysis. Banday et al. [13] talk about the recent advances in cache memories for the latest processors.
CHAPTER 2. BACKGROUND

Most of the latest processors were compared based on cache hierarchy, organization, performance, cache access patterns and cache replacement policies. The CPU’s cache memory access patterns and functionality are known and have been studied for years, but the same cannot be said about the GPU.
Multi2Sim is a well known and documented research simulator. The simulator was recently updated to include certain GPU architectures. Furthermore, Multi2Sim is regarded as the most up to date open source simulator that works with the latest versions of CUDA, OpenCL, Python, and GCC.

As stated previously, the two main GPU architectures that Multi2Sim can simulate are AMD’s Radeon Southern Islands and NVIDIA’s Kepler GPU architectures. Both GPU architectures are approximately two generations behind the current industry standard, as is the case with most of the CPU and GPU simulators. However, they can be modified to add any current or new architectures at the simulation level using newer releases. Both AMD Radeon and NVIDIA Kepler represent really strong GPUs that are used for a wide variety of applications like parallel and high performance computing. They are commonly used in machine learning and computer vision applications, mainly using libraries and frameworks like Tensorflow. AMD’s SI GPU architecture used the OpenCL libraries and framework while NVIDIA’s Kepler GPU architecture used CUDA libraries.

Multi2Sim emulates GPU architectures in software and enables the user to run a variety of benchmarks on the CPU itself as though the experiments were being run on a GPU. This enables the user to make architectural changes to the defined GPU model without making any hardware changes. Every time a change is made
in the GPU emulator code, the Multi2Sim framework is recompiled to reflect those changes, and a new executable is generated to run simulations. Multi2Sim supports multiple CPU and GPU benchmarks that help test the architectures for different workloads. Spec2000 is the most commonly used CPU benchmark suite. Contrarily, AMD SDK 2.5 [21] CUDA SDK [22] are the commonly used GPU benchmark suites. Both OpenCL and CUDA benchmarks display two sections, one for the CPU and one for the GPU. Both the CPU and GPU sections are simulated on the respective CPU and GPU architectures implemented in software by Multi2Sim.

### 3.1 Setting up the simulator on an Ubuntu OS

The open source Multi2Sim simulator code was downloaded from the official M2Sim github repository and was setup on a 64 bit Ubuntu 14.04 operating system. Although Ubuntu is the primary OS supported by Multi2Sim, the 14.04 version is the optimal Ubuntu version as the simulator was tried and tested on this version during the developmental phase. This does not limit the Multi2Sim to only the 14.04 version. Systems with newer Ubuntu versions can also be used to setup this simulator.

Table 3.1 shows the major dependencies of the Multi2Sim simulator. These specific packages and versions had been tested during the developmental stages of the Multi2Sim simulator and hence were chosen and setup on the testing system.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Ubuntu (14.04 preferred)</td>
</tr>
<tr>
<td>CUDA</td>
<td>CUDA version 6.5 (preferred) or higher</td>
</tr>
<tr>
<td>Python</td>
<td>Python2 and Python3</td>
</tr>
<tr>
<td>GCC</td>
<td>GCC version 4.8.2 or higher</td>
</tr>
</tbody>
</table>
CHAPTER 3. MULTI2SIM

Helper tools were used to download and add the installation scripts for the simulator. After running the configure and make install commands, Multi2Sim was successfully built, and the m2s executable was generated in the dedicated Multi2Sim home directory. The m2s executable was used to run various simulations on the defined CPU and GPU architecture models. All the commands supported by the simulator were listed by running the command m2s –help. Listing 3.1 shows all the bash commands that were used to configure and install Multi2Sim.

```
# Installing any dependencies – if required
$ sudo yum install devtoolset-2
$ scl enable devtoolset-2 bash

# Bash Commands to install Multi2Sim
$ libtoolize
$ aclocal
$ autoconf
$ automake --add-missing
$ cd ~/Multi2Sim_Home_Directory
$ ./configure --prefix=/home/pyn4262/tools/multi2sim/5.0/

# Compiling all the simulator source files and generating the m2s executable
$ make -j 4
$ make install

# Using the help command to find all the available commands for Multi2Sim
$ m2s --help
$ ./tools/multi2sim/5.0/bin/m2s --help
```

Listing 3.1: Bash commands to setup Multi2Sim

3.2 AMD Southern Islands (Radeon) GPU Architecture

AMD’s Radeon series of GPUs are the closest competitors to NVIDIA’s GeForce series of GPUs. AMD’s Southern Islands family of HD GPUs consists of the Radeon HD 7000 series of GPUs. The Southern Islands architecture defined by Multi2Sim consists of Radeon HD 7770, 7850, 7870 and 7970 architectures. The default and most commonly used Southern Islands architecture is Radeon HD 7970. This architecture was analyzed to understand the GPU and its architectural design.
Table 3.2: Difference in terminology between OpenCL and CUDA

<table>
<thead>
<tr>
<th>OpenCL</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Unit (CU)</td>
<td>Streaming Multiprocessor (SM)</td>
</tr>
<tr>
<td>Compute Element</td>
<td>CUDA Core</td>
</tr>
<tr>
<td>Work-item</td>
<td>Thread</td>
</tr>
<tr>
<td>Wavefront (64 work-items)</td>
<td>Warp (32 threads)</td>
</tr>
<tr>
<td>Wavefront pool</td>
<td>Warp pool</td>
</tr>
<tr>
<td>Work-group</td>
<td>Thread block</td>
</tr>
<tr>
<td>Local memory</td>
<td>Shared memory</td>
</tr>
<tr>
<td>Private memory</td>
<td>Registers</td>
</tr>
</tbody>
</table>

The AMD SI GPU architecture used an OpenCL Application Programming Interface (API) platform which serves as a parallel to the CUDA platform of NVIDIA’s GPUs. In the OpenCL implementation, the GPU is defined with Compute Unit (CU) which serves as the CUDA equivalent of NVIDIA’s Streaming Multiprocessor (SM). Furthermore, OpenCL APIs use work items, wavefront, and wavefront pools which serve as the CUDA equivalent of NVIDIA’s threads, warps, and warp pools (or thread blocks) respectively. Table 3.2 shows the difference in terminology between OpenCL and CUDA.

Figure 3.1 shows a block diagram of the AMD Radeon 7970 architecture from the AMD Southern Islands GPU family. The Radeon 7970 architecture has three main parts, compute devices, compute units and SIMD lanes. Figure 3.1 (a) shows the block diagram of a compute device. This specific architecture is defined using a thread dispatcher and scheduler with 32 compute units. The ultra-thread dispatcher is responsible for scheduling work groups and assigning them to the available Compute Units. The Compute Units interact directly with global memory, which is comprised of cache memory and main memory. This global memory block can be accessed by the entire collection of Compute Units in the ND-Range [14].

Figure 3.1 (b) shows the block diagram of a compute unit. The individual compute
unit has 4 SIMD execution units, each having 16 SIMD lanes for parallel work-item execution. These SIMD units are free to interact directly with local memory. The SIMD lanes allow parallel work-item execution, such that the same set of instructions is executed across multiple work-items. A wavefront consisting of 64 work-items is created within each work group and is assigned to a specific SIMD unit for execution. Each of the 16 SIMD lanes of the Compute Unit is executes four work-items per wavefront. The local memory block allows the work-items to share information during execution [14].

Figure 3.1 (c) represents a single SIMD lane which shows how an ALU interacts with the register file. The ALU is comprised of functional units to process integer and floating-point values. The register file is responsible for each work-item’s private memory [14].

Figure 3.2 shows a block diagram of the Compute Unit of the AMD Radeon 7970 GPU architecture. The figure shows the modular structure of the Compute Unit comprising of a Scalar Unit, Vector Memory Unit, Branch Unit, Local Data Share Unit (LDS) unit, and 4 SIMD units for parallel execution. During execution, multiple work-groups are assigned to each compute unit. These work-groups are further split into wavefronts (comprised of 64 work-items), and the wavefronts are executed simultaneously for each instruction. The front-end of the Compute Unit fetches and reads the instructions from the instruction memory. These instructions are then passed
to the functional units for execution. The Scalar Unit executes scalar arithmetic and scalar memory instructions. The Vector Memory Unit handles all vector global memory instructions. The Local Data Share Unit (LDS) handles the local memory instructions. The Branch Unit (BRU) handles all the branch instructions for control flow. Each of the functional units (SIMD units) interacts with either global or local memory and executes vector arithmetic and logic instructions [14].

Table 3.3 shows the hardware configuration of the AMD Radeon 7970 GPU series that was replicated by the AMD SI GPU model in Multi2Sim [10, 14]. L1 cache and L2 cache were defined as 4-way and 16-way set associative caches, respectively. The L1 cache was split between L1 scalar cache and L1 vector cache. The L1 scalar cache was responsible for all the L1 scalar instructions that got fetched and executed only once for the entire wavefront. Most of the constant data values were stored here. The L1 vector cache was responsible for all the vector instructions that got fetched for the whole wavefront but got executed multiple times for each of the 64 work-items in that wavefront for parallel execution. The L1 vector cache was the primary focus for the L1 cache statistics. The L1 vector cache has 32 memory modules such that each
Table 3.3: AMD Southern Islands emulated GPU configuration on Multi2Sim [10, 14]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computational</strong></td>
<td></td>
</tr>
<tr>
<td>Frequency</td>
<td>1000 Hz</td>
</tr>
<tr>
<td>Number of Compute Units</td>
<td>32</td>
</tr>
<tr>
<td>Number of SIMD lanes</td>
<td>16</td>
</tr>
<tr>
<td>Max Number of Wavefront Pools</td>
<td>4</td>
</tr>
<tr>
<td>Max Wavefronts per Pool</td>
<td>10</td>
</tr>
<tr>
<td><strong>L1 Vector Cache</strong></td>
<td></td>
</tr>
<tr>
<td>Number of L1 Cache Modules</td>
<td>32</td>
</tr>
<tr>
<td>Associativity / Number of Ways</td>
<td>4</td>
</tr>
<tr>
<td>Number of Sets</td>
<td>64</td>
</tr>
<tr>
<td>L1 Block Size</td>
<td>64 B</td>
</tr>
<tr>
<td>Total L1 Cache Size</td>
<td>512 KB</td>
</tr>
<tr>
<td><strong>L1 Scalar Cache</strong></td>
<td></td>
</tr>
<tr>
<td>Number of L1 Cache Modules</td>
<td>8</td>
</tr>
<tr>
<td>Associativity / Number of Ways</td>
<td>4</td>
</tr>
<tr>
<td>Number of Sets</td>
<td>64</td>
</tr>
<tr>
<td>L1 Block Size</td>
<td>64 B</td>
</tr>
<tr>
<td>Total L1 Cache Size</td>
<td>128 KB</td>
</tr>
<tr>
<td><strong>L2 Cache</strong></td>
<td></td>
</tr>
<tr>
<td>Number of L2 Cache Modules</td>
<td>6</td>
</tr>
<tr>
<td>Associativity / Number of Ways</td>
<td>16</td>
</tr>
<tr>
<td>Number of Sets</td>
<td>128</td>
</tr>
<tr>
<td>L2 Block Size</td>
<td>64 B</td>
</tr>
<tr>
<td>Total L2 Cache Size</td>
<td>768 KB</td>
</tr>
</tbody>
</table>

of the 32 compute units is directly mapped to a single L1 cache module. L2 cache has 6 memory modules that are shared by all of the 32 compute units. The L2 cache had a higher set associativity than L1 cache and was larger than L1 cache in terms of cache size.
3.3 NVIDIA Kepler GPU Architecture

NVIDIA is the leader in the GPU market with its latest generation GeForce GPU series. The NVIDIA Kepler GPU architecture was the sole NVIDIA architecture available on Multi2Sim, but future updates may result in other GPU models. The Kepler architecture comprises of the GeForce 6000 series and the GeForce 7000 series GPUs.

Figure 3.3 shows a block diagram of the NVIDIA Kepler architecture from the Kepler series of GPUs. Part (a) shows the block diagram of the SIMD execution pipeline of the GPU. The SIMD pipeline consists of the front end, a register file, 32 SIMD lanes followed by an execution buffer. Part (b) shows a block diagram of the Branch Unit (BRU) in the GPU architecture. The BRU describes how the opcode is fetched from the dispatch buffer and sent to the execution unit. Moreover, the data are written to the register file using the write buffer. Part (c) shows the Load and Store Unit (LSU) of the GPU architecture. The LSU controls how the GPU interacts with its different memory units like cache, shared memory, global memory, and local memory. The LSU unit is used by the Streaming Multiprocessor (SM) to interact with the GPU’s memory units, and data are either written to or read from these memory units during execution.
Figure 3.4: Kepler GPU Memory Architecture [1]

Figure 3.4 shows the top level block diagram of the Kepler GPU architecture. The GigaThread Engine interacts with the SMs which in turn interact with global memory. There are a total of 14 SMs defined in the Kepler architecture on the Multi2Sim and each SM has a dedicated L1 cache module assigned to it. The L2 cache modules are shared between multiple SMs. Finally, the entire memory architecture is supported using interconnects and memory controllers. The GigaThread engine is responsible for storing and assigning the thread blocks to the defined SMs. The block and grid dimensions are defined in the GigaThread Engine. The engine is in charge of assigning thread blocks to available SMs for execution. The thread blocks then get passed on to the SM where they get executed for the fetched instruction opcode. Based on available resources, multiple thread blocks may be assigned to the same SM. The Kepler architecture is defined using CUDA, which specifies a thread as set of instructions to be executed. Multiple threads can be executed in parallel using the Single Instruction Multiple Data (SIMD) pipeline which enables high data level parallelism. In CUDA, a group of threads is defined as a thread block, where all the threads in that thread block execute the same set of instructions in parallel. A group of 32 threads is defined as a warp, and a thread block is further comprised of multiple warps. These warps get assigned to available SMs during execution. This
architecture defines 4 warp schedulers, thus enabling 4 warps to be scheduled and processed at the same time [8].

Figure 3.5 shows the block diagram of the Streaming Multiprocessor (SM) in the Kepler architecture. Each SM is comprised of various functional units like the Single Precision Unit (SPU), the Branch Unit (BRU), the Double Precision Unit (DPU), the Integer Math Unit (IMU), the Special Functional Unit (SFU), and the Load and Store Unit (LSU).

Table 3.4 shows the hardware configuration of the NVIDIA Tesla K20X GPU [17, 18] that was matched by the NVIDIA Kepler GPU architecture on Multi2Sim.

Similar to the AMD GPU architecture, NVIDIA’S Kepler architecture also defines L1 caches and L2 caches as 4-way and 16-way set associative cache blocks, respectively. L1 cache had 14 memory modules such that each of the 14 Streaming Multiprocessors (SM) is directly mapped to a single L1 cache module. L2 cache is divided into 6
Table 3.4: NVIDIA Kepler emulated GPU configuration on Multi2Sim [1, 14]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computational</td>
<td></td>
</tr>
<tr>
<td>Computational Frequency</td>
<td>732 Hz</td>
</tr>
<tr>
<td>Computational Number of SMs</td>
<td>14</td>
</tr>
<tr>
<td>Computational Number of SIMD lanes</td>
<td>32</td>
</tr>
<tr>
<td>Computational Warp Size</td>
<td>32</td>
</tr>
<tr>
<td>Computational Max Warps per SM</td>
<td>64</td>
</tr>
<tr>
<td>Computational Max Threads per SM</td>
<td>2048</td>
</tr>
<tr>
<td>L1 Cache</td>
<td></td>
</tr>
<tr>
<td>L1 Cache Number of L1 Cache Modules</td>
<td>14</td>
</tr>
<tr>
<td>L1 Cache Associativity / Number of Ways</td>
<td>4</td>
</tr>
<tr>
<td>L1 Cache Number of Sets</td>
<td>32</td>
</tr>
<tr>
<td>L1 Cache L1 Block Size</td>
<td>128 B</td>
</tr>
<tr>
<td>L1 Cache Total L1 Cache Size</td>
<td>16 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td></td>
</tr>
<tr>
<td>L2 Cache Number of L2 Cache Modules</td>
<td>6</td>
</tr>
<tr>
<td>L2 Cache Associativity / Number of Ways</td>
<td>16</td>
</tr>
<tr>
<td>L2 Cache Number of Sets</td>
<td>32</td>
</tr>
<tr>
<td>L2 Cache L2 Block Size</td>
<td>128 B</td>
</tr>
<tr>
<td>L2 Cache Total L2 Cache Size</td>
<td>1536 KB</td>
</tr>
</tbody>
</table>

memory modules which are shared by all of the 14 SMs.

The AMD SI and NVIDIA Kepler GPU architectures were both linked to Multi2Sim’s common memory model. Initially, the NVIDIA Kepler architecture was not connected to any memory module, as the common memory module defined in Multi2Sim was based on the AMD GPU’s memory hierarchy. After a discussion with the Multi2Sim Developers team, there was consensus that the memory model could be connected to the NVIDIA Kepler GPU architecture to show accurate functionality. Multi2Sim’s memory module was then connected to the NVIDIA Kepler GPU architecture in the LSU unit of the SM definition. The LSU unit was responsible for all memory accesses. Since both GPU architectures used a common memory model, the results were consistent and easy to compare.

Listings 3.2 and 3.3 show the UNIX commands used to compile and run OpenCL and CUDA benchmarks on Multi2Sim.
CHAPTER 3. MULTI2SIM

Listing 3.2: Compiling the OpenCL and CUDA benchmarks

Listing 3.3: Running the OpenCL and CUDA benchmarks on Multi2Sim
Chapter 4

Analyzing Cache Memory Access Patterns

The objective of this research is to analyze cache memory access patterns for GPU architectures. In particular, the three metrics include cache hit ratios, Most Recently Used (MRU) and Least Recently Used (LRU) cache access counters, and finally data cache accesses for intra-warp and inter-warp locality for different workloads.

The L1 and L2 cache statistics and hit ratios were compared for a variety of benchmarks for both AMD SI and NVIDIA Kepler GPU models. The data cache hit ratios varied based on the benchmark being tested and depended greatly on the benchmarks size. Each benchmark was run for multiple input sizes to find the optimal cache hit ratio. All cache hit ratios were compared for the all benchmarks to understand the cache performance of both AMD SI and NVIDIA Kepler GPU architectures for the tested workloads.

The MRU and LRU counters recorded the most and least recently used blocks in each set for L1 and L2 cache respectively. The results for each set were combined to generate cache access percentages for each MRU line in cache. The cache access percentages for different MRU lines were compared and analyzed to find GPU cache memory trends for a variety of benchmarks. The GPU results were compared against the CPU as a reference.

The L1 and L2 data cache accesses were studied for intra-warp and inter-warp localities. Inter-warp accesses represent the accesses made in data cache such that
the accessing threads belong to the same warp. Intra-warps accesses represent the accesses made in data cache where the accessing threads do not belong to the same warp, but belong to different warps within the warp-pool. The inter-warps and intra-warps results varied based on the workload characteristics. The cache statistics were recorded and analyzed for OpenCL and CUDA benchmarks for inter-warps and intra-warps localities.

4.1 L1 and L2 Cache Hit Ratios

4.1.1 AMD Southern Islands

The hardware configuration defined in Chapter 3 demonstrates the AMD Southern Islands GPU architecture model on Multi2Sim. This model replicates the AMD Radeon HD 7970 GPUs series. The AMD Southern Islands GPU model uses a 4-way set associative L1 cache and a 16-way set associative L2 cache. The L1 cache consists of two sections, L1 scalar cache and L1 vector cache. L1 scalar cache has a total of 8 defined memory modules, such that each module maps to four compute units. L1 vector cache has a total of 32 defined memory modules, such that each module maps to a single compute unit. Contrarily, L2 cache does not have any scalar or vector differentiation. L2 cache consists of 6 cache memory modules which are shared across all 32 compute units during execution.

Multi2Sim’s pre-defined memory counters were used to find the memory statistics for the entire memory model. These memory statistics were recorded for cache memory, global memory and local memory. The memory statistics included Accesses, Evictions, Retrieved Accesses, Hits, Misses, and Hit Ratios for each and every memory module defined in cache. All these statistics across various memory modules for the specific type of cache (L1 or L2) were compared. The same set of AMD SDK OpenCL benchmarks was used to test the behavior and cache hit ratios of L1 scalar
cache, L1 vector cache and L2 cache. All the results were tabulated, graphed and compared to find common trends of cache memory using the defined AMD SI GPU model on Multi2Sim.

4.1.2 NVIDIA Kepler

Similarly, the L1 and L2 cache statistics were recorded for the NVIDIA Kepler GPU model on Multi2Sim. Pertaining to the hardware configurations mentioned in Chapter 3 for the Kepler GPU model, the L1 and L2 cache definitions were noted. The NVIDIA Kepler GPU model did not differentiate the L1 cache as scalar or vector cache, as the AMD SI GPU model did. The Kepler model defined the L1 cache as unified cache modules. The Kepler GPU model used a 4-way set associative L1 cache and a 16-way set associative L2 cache, similar to the AMD SI model. L1 vector cache has a total of 14 defined memory modules, such that each L1 cache module maps to a single SM (as the Kepler architecture used 14 SMs). L2 cache consists of 6 cache memory modules, and these are shared across multiple SMs during execution.

The memory statistics for various CUDA SDK benchmarks were recorded for the NVIDIA Kepler GPU architecture. The L1/L2 cache hit ratios and memory statistics were tabulated, graphed and analyzed. The results from the CUDA benchmarks and OpenCL benchmarks were compared to understand which model worked better with the respective memory hierarchy.
4.2 MRU and LRU Cache Counters for Temporal Locality

The simulator, timing and memory code for both AMD SI and NVIDIA Kepler GPU architectures was studied to learn how the memory hierarchy was used in each GPU architecture. As explained in Chapter 3, the AMD GPU architecture used an OpenCL API platform while the NVIDIA Kepler GPU architecture used a CUDA API platform.

The Compute Unit (for OpenCL) or the Streaming Multiprocessor (for CUDA) is the main processing unit of the GPU that executes the defined kernels. Both GPU models use Multi2Sim’s memory model to load and store data. Specific memory units are defined in the GPU architectures on Multi2Sim, and these units are responsible for performing all memory load and store operations for the respective GPU architecture. This memory unit is called the Vector Memory Unit in AMD’s SI model and Load and Store (LSU) unit in NVIDIA’s Kepler model respectively.

The memory model was studied to find the cache memory implementation. First, the pre-defined sets and blocks in cache were modified to include set and block counters within their respective modular designs. These counters were incremented every time the cache block was accessed. Additionally, the cache code was examined to find the defined caching policies. The cache block used three cache replacement policies: Random, First In First Out (FIFO) and Least Recently Used (LRU). These cache replacement policies signify how the list of blocks are replaced in memory.

The Random replacement policy replaces blocks of memory at random when a new block is added to cache. Each set uses a list to store all the existing blocks in that respective set. This list of blocks is created during initialization. The new block replaces the old block at the exact same position in the list of blocks in cache. The FIFO Replacement policy replaces the blocks in memory using the First In First Out (FIFO) protocol as used in the Queue data structure. The oldest block is the first
one to be replaced. The new block replaces the old block at the same position in the list but with its new time stamp to indicate that it is the most recently added block.

The Least Recently Used (LRU) replacement policy works by removing the least recently used block from cache memory when the cache is full, and replaces that block with the new entry. Each set in cache uses its own LRU list, which is a list or collection of blocks for that respective set in cache. This LRU list is used only for the Replacement LRU caching policy. The Replacement LRU policy is the most broadly used replacement policy. Unlike the generic list of blocks used for Random and FIFO replacement policies, this LRU list organizes the blocks from Most Recently Used (MRU) to Least Recently Used (LRU) blocks. The LRU list keeps track of the blocks in cache memory and updates the position of the blocks based on the most recent cache accesses made. The new block replaces the old block at the same position of the LRU list. The new block is then removed and added to the front of the LRU list to indicate that it is the most recently used block in cache. For example, if the LRU list is organized to accommodate Blocks 1, 2 and 3 (in that order), when the cache is full and Block3 is the least recently used block in cache, then Block3 is replaced in the list with the new entry. The new block is placed in the same position as the replaced block in the LRU list. That new block node is then removed from the list and is added to the head of the LRU list indicating that Block3 is the most recently used block and is stored in the first MRU line. The LRU list is ordered from most recently used blocks to least recently used blocks in cache memory, for each set in cache. Figure 4.1 shows the three cache replacement policies implemented in the memory model and their working.

Cache specific counters were defined in the memory hierarchy to keep track of the Most Recently Used (MRU) and Least Recently Used (LRU) lines in cache memory. This helped identify the temporal locality of GPU cache and identify certain memory access patterns in cache memory. These counters were called MRU and LRU counters.
as the name signifies, and helped identify the accesses made to the most recently used and least recently used lines in cache memory. These MRU and LRU counters were defined for both L1 and L2 cache blocks.

A new LRU counter list was created to replicate the LRU caching list used by each set in cache. The list was defined in the architecture of the cache itself, such that it could be updated every time the LRU caching list was updated by the LRU cache replacement policy. As soon as a cache access was made to a specific block in cache memory, that block’s index in the LRU caching list was obtained and the block’s internal counter was incremented. The LRU counter list was then incremented at the same index to show that the block at that index was accessed. The defined LRU caching policy protocol identified the block being accessed and then removed the block node from the LRU caching list placing it at the head of the list. This indicated that the first block in the list was the most recently used block and the

![Figure 4.1: Cache Replacement Policies](image)

as the name signifies, and helped identify the accesses made to the most recently used and least recently used lines in cache memory. These MRU and LRU counters were defined for both L1 and L2 cache blocks.

A new LRU counter list was created to replicate the LRU caching list used by each set in cache. The list was defined in the architecture of the cache itself, such that it could be updated every time the LRU caching list was updated by the LRU cache replacement policy. As soon as a cache access was made to a specific block in cache memory, that block’s index in the LRU caching list was obtained and the block’s internal counter was incremented. The LRU counter list was then incremented at the same index to show that the block at that index was accessed. The defined LRU caching policy protocol identified the block being accessed and then removed the block node from the LRU caching list placing it at the head of the list. This indicated that the first block in the list was the most recently used block and the
last block was the least recently used block. The next time that block was accessed in cache, the LRU list would increment the counter at the block’s new index thus replicating the defined functionality.

The same process was repeated for every cache access made, and the LRU counters (at the respective block indices) were incremented every time a block was accessed based on the initial position of the block in the LRU caching list. This meant that the original index of the block was used to show which cache line was accessed in terms of most recently used down to least recently used.

Listing 4.1 shows the MRU counter implementation in cache. Listing 4.2 shows how the MRU counters were incremented and updated based on the original index of the accessed block. This was done in correlation with the LRU caching policy. The AccessBlock() function defined in the Cache.cc file was modified. Listing 4.3 shows a small portion of code from the file SystemEvents.cc. The code is used to call the AccessBlock() function every time a cache access is made. The memory module’s access counter and statistics are also updated here.
CHAPTER 4. ANALYZING CACHE MEMORY ACCESS PATTERNS

```cpp
class Set {
    // Only Cache needs to initialize fields
    friend class Cache;

    // List of blocks in LRU order - Pre-defined and used by the LRU caching policy
    misc::List<Block> lru_list;

    // Position in Cache::blocks where the blocks start for this set
    Block *blocks;

    // Creating a list of MRU/LRU counters
    int* lru_counts;

    void init_lru_counts(int n) {
        this->lru_counts = new int[n];
        for (int i=0; i<n; i++) {
            this->lru_counts[i] = 0;
        }
    }

    void set_lru_counts(int* arr) {
        this->lru_counts = arr;
    }

    int* get_lru_counts() {
        return this->lru_counts;
    }

    void increment_lru_counts(int position) {
        this->lru_counts[position]++;
    }
};
```

**Listing 4.1:** MRU implementation in the Set architecture of cache. (Defined in Cache.cc)
void Cache::AccessBlock(unsigned set_id, unsigned way_id)
{
    // Get set and block
    Set *set = getSet(set_id);
    Block *block = getBlock(set_id, way_id);

    // A block is moved to the head of the list for LRU policy. It will also be moved if
    // it is its first access for FIFO policy, i.e., if the state of the block was invalid.
    bool move_to_head = replacement_policy == ReplacementLRU ||
        (replacement_policy == ReplacementFIFO && block->state == BlockInvalid);

    // Move to the head of the LRU list
    if (move_to_head)
    {
        // *************** Incrementing the MRU counters ***************
        int index = 0;
        for (misc::List<Block>::Iterator iter = set->lru_list.begin();
            iter != set->lru_list.end(); ++iter)
        {
            if (iter.node == &(block->lru_node))
            {
                set->increment_lru_counters(index);
                break;
            }
            index++;
        }
        set->lru_list.Erase(block->lru_node);
        set->lru_list.PushFront(block->lru_node);
    }

    // Code - For each Block counter
    block->incrementCounter();
}

Listing 4.2: Incrementing the MRU counters in cache (Defined in Cache.cc)

// Statistics
module->incAccesses();
module->inc_intra_access_counters(frame->warp_pool_id, frame->warp_id_in_pool);
module->UpdateStats(frame);

// Entry is locked. Record the transient tag so that a
// subsequent lookup detects that the block is being brought.
// Also, update LRU counters here.
cache->setTransientTag(frame->set, frame->way, frame->tag);
cache->incAccessCounter();
cache->AccessBlock(frame->set, frame->way);

// Access latency
module->incDirectoryAccesses();
esim_engine->Next(event_find_and_lock_action, module->getDirectoryLatency());

Listing 4.3: Incrementing the memory statistics in cache (Defined in SystemEvents.cc)
4.3 Inter-warp and Intra-warp Cache Locality

Inter-warp and intra-warp counters were defined in order to understand the L1 and L2 data cache locality. Inter-warp accesses represented the accesses made in data cache such that the executed threads belonged to the same warp. Intra-warp accesses represented the accesses made in data cache where the executed threads did not belong to the same warp, but belonged to different warps within the warp-pool.

This set of inter and intra counters were defined in the same way for both AMD SI and NVIDIA Kepler GPU architectures on Multi2Sim. Since the AMD SI GPU architecture uses OpenCL APIs, these counters represent inter-wavefront and intra-wavefront accesses. In the OpenCL model, a wavefront is defined as a collection of 64 work-items. Contrarily, since the NVIDIA Kepler architecture uses CUDA APIs, these counters represent inter-warp and intra-warp accesses. In the CUDA implementation, a warp is defined as a collection of 32 threads.

These inter-warp (or inter-wavefront) and intra-warp (or intra-wavefront) accesses were recorded for each memory module of L1 and L2 data cache. The counters were defined and incremented in the memory model of Multi2Sim (Module.cc). Both AMD SI and NVIDIA Kepler used the same memory model on Multi2Sim, thus making the memory code modifications identical for both architectures, although, the Load and Store Units of both GPU architectures needed to be modified to pass on the warp id and warp-pool id (or wavefront id and wavefront pool id) for each warp/wavefront during execution. The vector memory unit was in charge of load and store instructions on the AMD SI architecture. Meanwhile the Load and Store Unit (LSU) was in charge of all memory operations in the NVIDIA CUDA architecture. The inter-warp and intra-warp counters definitions for the memory module are shown in Listing 4.4.
CHAPTER 4. ANALYZING CACHE MEMORY ACCESS PATTERNS

// Inter-warp and intra-warp code in Module.h and Module.cc files in the memory model of Multi2Sim

// Temp warp id and warp pool id to be stored by the LSU unit during scheduling warps to SMs
int temp_warp_pool_id = 0;
int temp_warp_id_in_pool = 0;

// The warp id and warp pool id from the previous frame accessed
int last_warp_id = -1;
int last_warp_pool_id = -1;

// The warp id and warp pool id from the previous frame that was hit for intra-warp and inter-warp
int last_intra_hit_warp_id = -1;
int last_intra_hit_warp_pool_id = -1;
int last_inter_hit_warp_id = -1;
int last_inter_hit_warp_pool_id = -1;

// Intra-warp and inter-warp access and hit counters
int intra_warp_counter = 0;
int inter_warp_counter = 0;
int intra_hit_counter = 0;
int inter_hit_counter = 0;

// Function to increment inter-warp and intra-warp counters
void inc_inter_warp_counter(int warp_pool_id, int warp_id){
if ((last_warp_id == -1) && (last_warp_pool_id == -1)) {
last_warp_pool_id = warp_pool_id;
last_warp_id = warp_id;
}
else {
if ((last_warp_id == warp_id) && (last_warp_pool_id == warp_pool_id)) {
intra_warp_counter++;
}
else {
inter_warp_counter++;
last_warp_pool_id = warp_pool_id;
last_warp_id = warp_id;
}
}

void inc_intra_hit_counter(int warp_pool_id, int warp_id){
if ((last_intra_hit_warp_id == -1) && (last_intra_hit_warp_pool_id == -1)) {
last_intra_hit_warp_pool_id = warp_pool_id;
last_intra_hit_warp_id = warp_id;
}
else {
if ((last_intra_hit_warp_id == warp_id) && (last_intra_hit_warp_pool_id == warp_pool_id)) {
intra_hit_counter++;
}
else {
inter_hit_counter++;
last_intra_hit_warp_pool_id = warp_pool_id;
last_intra_hit_warp_id = warp_id;
}
}
}

Listing 4.4: Inter-warp and intra-warp definitions in Module.cc
Although both GPU architectures used the same memory model in Multi2Sim, the Load and Store Units of both GPU architectures were defined differently. The Load and Store Units for both GPU architectures were modified to pass on the warp id and warp-pool id (or wavefront id and wavefront pool id) for each warp/wavefront to the data cache memory module during execution. The vector memory unit was in charge of load and store instructions on the AMD SI architecture. However, the Load and Store Unit (LSU) was in charge of all memory operations in the NVIDIA CUDA architecture. The code additions in the Load and Store Unit included recording the warp id and warp pool id for each warp being executed by the SM and passing on those values to the memory module. The memory module used the warp pool id and warp id and incorporated them in each frame object for every cache access made. This helped track the inter-warp and intra-warp accesses in the memory module. Every time the frame was accessed as a hit or a miss, the memory module updated its internal counters in the UpdateStats function. The warp id and warp pool id of the previously accessed frame were compared to those of the currently accessed frame to determine whether the data cache access represented inter-warp or intra-warp locality. The same process was repeated in the Vector Memory Unit of the AMD SI GPU architecture, but for wavefronts instead of warps. The inter-warp and intra-warp implementation in the LSU unit of the Kepler GPU architecture is shown in Listing 4.5. The UpdateStats function defined in the memory module is shown in Listing 4.6.
CHAPTER 4. ANALYZING CACHE MEMORY ACCESS PATTERNS

// Update Uop write ready cycle
for (auto it = uop->getWarp() ->ThreadsBegin(); e = uop->getWarp() ->ThreadsEnd(); it != e; ++it)
{
    // Get thread
    Thread *thread = it ->get()
;
    // Access memory for each active thread
    if (uop->getWarp() ->getThreadActive(thread->getIdInWarp()))
    {
        // Get the thread uop
        Uop::ThreadInfo *thread_info = &uop->thread_info_list
            [thread->getIdInWarp()];

        // Check if the thread info struct has already made a successful cache access. If so, move on to the next thread
        if (thread_info->accessed_cache)
            continue;

        // Translate virtual address to a physical address
        unsigned physical_address = sm->getGPU() ->getMmu() ->
            TranslateVirtualAddress( uop->getThreadBlock() ->getGrid() ->address_space, thread_info->global_memory_access_address);

        // Make sure we can access the cache if so, submit the access.
        // if not, mark the accessed flag of the thread info struct
        if (sm->cache->canAccess(physical_address))
        {
            // Setting warp id and warp pool id for Intra-warp and Inter-warp Accesses
            Warp *warp = uop->getWarp();
            int warp_id = warp->getId();
            int warp_pool_id = uop->getWarpPoolId();

            WarpPoolEntry *warp_pool_entry = uop->getWarpPoolEntry();
            int id_in_warp_pool = warp_pool_entry->getIdInWarpPool();

            // Setting the Warp Pool and Warp Ids in Module
            sm->cache->temp_warp_pool_id = warp_pool_id;
            sm->cache->temp_warp_id_in_pool = id_in_warp_pool;

            // This is where the Cache access call is made
            sm->cache->Access( module_access_type, physical_address);
            thread_info->accessed_cache = true;

            // Statistics
            this->cache_access_counter++;
            sm->inc_total_cache_accesses();
        }
        else
            all_threads_accessed = false;
    }
}
CHAPTER 4. ANALYZING CACHE MEMORY ACCESS PATTERNS

// UpdateStats function defined in Module.cc in the memory model of Multi2Sim
// The intra-warp and inter-warp counters are also incremented here

void Module::UpdateStats(Frame *frame) {
    // Assert that the frame module is in fact the module
    assert(this == frame->getModule());

    // Record access type. Hits and Misses recorded separately here so that
    // we can sanity check them against the total number of accesses.
    if (frame->request_direction == Frame::RequestDirectionUpDown) {
        if (frame->read) {
            num_reads++;
            if (frame->retry)
                num_retry_reads++;    
            if (frame->blocking)
                num_blocking_reads++;     
            else
                num_non_blocking_reads++;  
            if (frame->hit) {  
                // Incrementing the inter-warp and intra-warp counters
                inc_inter_intra_hit_counter(frame->warp_pool_id, frame->warp_id_in_pool);
                num_read_hits++;
                if (frame->retry)
                    num_retry_read_hits++;
            } else {     
                num_read_misses++;
                if (frame->retry)
                    num_retry_read_misses++;   
            }
        } else if (frame->nc_write) {
            num_ncWrites++;
            if (frame->retry)
                num_retry_ncWrites++;
            if (frame->blocking)
                num_blocking_ncWrites++;  
            else
                num_non_blocking_ncWrites++;  
            if (frame->hit) {  
                // Incrementing the inter-warp and intra-warp counters
                inc_inter_intra_hit_counter(frame->warp_pool_id, frame->warp_id_in_pool);
                num_nc_write_hits++;
                if (frame->retry)
                    num_retry_nc_write_hits++;   
            } else {      
                num_nc_write_misses++;
                if (frame->retry)
                    num_retry_nc_write_misses++;  
            }
        } else if (frame->write) {
            num_writes++;
            if (frame->retry)
                num_retry_writes++;  
        }
    }
}
CHAPTE...
Chapter 5

Results

5.1 Cache Hit Ratios

A set of benchmarks from the AMD SDK 2.5 Suite [21] and CUDA SDK 6.5 Suite [22] were run on Multi2Sim to test the cache behavior of the GPU for a variety of workloads. The OpenCL benchmarks from the AMD SDK suite were tested on the AMD SI GPU model, while the CUDA benchmarks were tested on the NVIDIA Kepler GPU model. Figure 5.1 shows the L1 and L2 cache hit ratio comparisons for the executed OpenCL benchmarks. Figure 5.2 shows the L1 and L2 cache hit ratio comparisons for the executed CUDA benchmarks.

Referring to the OpenCL benchmark results from Figure 5.1, the average hit ratios for L1 scalar cache, L1 vector cache and L2 cache were 70%, 46% and 35% respectively. The L1 scalar data cache is responsible for storing scalar and constant values during execution while the L1 vector data cache is responsible for storing the vector data for the workload being tested on the GPU. Comparing the hit ratios of L1 and L2 cache for the CUDA benchmarks in Figure 5.2, the average hit ratios for L1 and L2 data cache were 46% and 26% respectively.

For both sets of benchmarks, the L2 cache showed the lowest hit ratios among the L1 and L2 data caches. This was because unlike L1 cache modules, L2 modules were shared by multiple CUs or SMs during execution. These low level cache modules in
CHAPTER 5. RESULTS

Figure 5.1: Cache Memory Hit Ratios for the OpenCL benchmarks

Figure 5.2: Cache Memory Hit Ratios for the CUDA benchmarks
L2 cache were accessed only if the L1 cache access was not successful.

Both L1 and L2 data caches display hit ratios of under 50% on average, and this is because of the working of the GPU. The GPU relies heavily on parallel thread execution and less on cache memory. Contrarily, the CPU has fewer number of cores and processes a small number of threads at a time but relies more heavily on cache memory to show high performance levels. The GPU exploits Thread-level Parallelism (TLP) while the CPU exploits Instruction-level Parallelism (ILP).

### 5.2 MRU and LRU Temporal Locality Results

A set of benchmarks from the AMD SDK 2.5 Suite [21] and CUDA SDK 6.5 Suite [22] were run on Multi2Sim to find temporal locality of multiple MRU lines for L1 and L2 cache blocks. The OpenCL benchmarks from the AMD SDK 2.5 suite were used to test the cache behavior of the AMD SI GPU model, while the CUDA SDK 6.5 suite was used to test the cache behavior of the NVIDIA Kepler GPU model. Each benchmark was compiled and then run on the simulator for a specific number of elements as specified in the respective benchmarks kernel files. The temporal locality of multiple lines were recorded for all benchmarks. These results were compared to analyze the cache behavior for different types of workloads and the locality for each MRU line.

Table 5.1 and Figure 5.3 indicate the L1 cache statistics and MRU results for the 4-way set associative L1 cache architecture. Table 5.2 and Figure 5.4 indicate the L2 cache statistics and MRU results for the 16-way set associative L2 cache architecture. The MRU counters for each MRU line in cache were recorded.
Table 5.1: OpenCL Benchmark MRU results for L1 Cache (per module) of the AMD SI architecture

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MRU0</th>
<th>MRU1</th>
<th>MRU2</th>
<th>MRU3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitonic Sort</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Binomial Option</td>
<td>81%</td>
<td>17%</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>Floyd Marshall</td>
<td>76%</td>
<td>6%</td>
<td>8%</td>
<td>10%</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>96%</td>
<td>1%</td>
<td>1%</td>
<td>2%</td>
</tr>
<tr>
<td>Simple Convolution</td>
<td>59%</td>
<td>16%</td>
<td>7%</td>
<td>18%</td>
</tr>
<tr>
<td>Recursive Gaussian</td>
<td>70%</td>
<td>11%</td>
<td>6%</td>
<td>13%</td>
</tr>
<tr>
<td>Sobel Filter</td>
<td>81%</td>
<td>12%</td>
<td>5%</td>
<td>2%</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>85%</td>
<td>4%</td>
<td>5%</td>
<td>6%</td>
</tr>
<tr>
<td>Mersenne Twister</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>90%</td>
<td>3%</td>
<td>3%</td>
<td>4%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>84%</td>
<td>7%</td>
<td>3%</td>
<td>6%</td>
</tr>
</tbody>
</table>

Figure 5.3: OpenCL benchmark results for the AMD SI architecture for L1 cache
Table 5.2: CUDA Benchmark MRU results for L1 Cache of the NVIDIA Kepler architecture

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MRU0</th>
<th>MRU1</th>
<th>MRU2</th>
<th>MRU3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Add</td>
<td>96 %</td>
<td>1 %</td>
<td>1 %</td>
<td>2 %</td>
</tr>
<tr>
<td>Transpose</td>
<td>93 %</td>
<td>3 %</td>
<td>2 %</td>
<td>2 %</td>
</tr>
<tr>
<td>Simple Streams</td>
<td>51 %</td>
<td>28 %</td>
<td>15 %</td>
<td>6 %</td>
</tr>
<tr>
<td>Simple Call Back</td>
<td>87 %</td>
<td>2 %</td>
<td>2 %</td>
<td>9 %</td>
</tr>
<tr>
<td>Histogram</td>
<td>65 %</td>
<td>32 %</td>
<td>2 %</td>
<td>1 %</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>90 %</td>
<td>4 %</td>
<td>3 %</td>
<td>3 %</td>
</tr>
<tr>
<td>Fast Walsh Transform</td>
<td>58 %</td>
<td>37 %</td>
<td>1 %</td>
<td>4 %</td>
</tr>
<tr>
<td>Cpp Overload</td>
<td>99 %</td>
<td>1 %</td>
<td>0 %</td>
<td>0 %</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>77 %</td>
<td>15 %</td>
<td>4 %</td>
<td>4 %</td>
</tr>
</tbody>
</table>

Figure 5.4: CUDA benchmark MRU Results for L1 Cache of the NVIDIA Kepler architecture
CHAPTER 5. RESULTS

For the OpenCL benchmarks, the most recently used line (MRU0) on average comprised of 84% of the total number of access made to L1 cache. The second, third and fourth MRU lines consists of 7%, 3% and 6% of all the L1 cache accesses. For the CUDA benchmarks, the first MRU line on average consists of 77% of the total number of access made to L1 cache. The second, third and fourth MRU lines consists of 15%, 3.53% and 4% of all the L1 cache accesses.

Some OpenCL benchmarks like Bitonic Sort, Radix Sort, Mersenne Twister and Monte Carlo showed the highest cache access percentages for the most recently used line of 90% or more for the OpenCL benchmarks. Similarly, the VectorAdd, Transpose and Cpp Overload CUDA benchmarks also showed the highest number of cache accesses in the most recently used line of above 90% or more for the CUDA benchmarks. The Simple Convolution OpenCL benchmark shows the lowest cache access percentage for the most recently used line at under 50% for the OpenCL results. Similarly, Fast Walsh Transform and Simple Streams CUDA benchmarks show the lowest percentages of accesses in the most recently used line of under 60% for the CUDA results.

Tables 5.3 and 5.4 indicate the L2 cache statistics and MRU results for the 16-way set associative L2 cache architecture for the OpenCL benchmarks and CUDA benchmarks respectively. Similar to the L1 cache results, the MRU counters for each MRU line in cache were recorded. The graphical representation of the MRU results for L2 cache for the OpenCL and CUDA benchmarks are shown in Figures 5.5 and 5.6 respectively.
Table 5.3: OpenCL Benchmark MRU results for L2 Cache of the AMD SI architecture

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MRU0 %</th>
<th>MRU1 %</th>
<th>MRU2 %</th>
<th>MRU3 - 15 %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitonic Sort</td>
<td>97</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Binomial Option</td>
<td>99</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Floyd Marshall</td>
<td>31</td>
<td>12</td>
<td>16</td>
<td>41</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>75</td>
<td>24</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Simple Convolution</td>
<td>73</td>
<td>27</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Recursive Gaussian</td>
<td>51</td>
<td>14</td>
<td>9</td>
<td>26</td>
</tr>
<tr>
<td>Sobel Filter</td>
<td>36</td>
<td>13</td>
<td>12</td>
<td>39</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>32</td>
<td>22</td>
<td>18</td>
<td>28</td>
</tr>
<tr>
<td>Mersenne Twister</td>
<td>53</td>
<td>15</td>
<td>10</td>
<td>22</td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>58</td>
<td>2</td>
<td>2</td>
<td>38</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>60</strong></td>
<td><strong>13</strong></td>
<td><strong>7</strong></td>
<td><strong>20</strong></td>
</tr>
</tbody>
</table>

Figure 5.5: OpenCL benchmark results for the AMD SI architecture for L2 cache
Table 5.4: CUDA Benchmark MRU results for L2 Cache of the NVIDIA Kepler architecture

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MRU0</th>
<th>MRU1</th>
<th>MRU2</th>
<th>MRU3 - MRU15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Add</td>
<td>65 %</td>
<td>3 %</td>
<td>3 %</td>
<td>29 %</td>
</tr>
<tr>
<td>Transpose</td>
<td>33 %</td>
<td>11 %</td>
<td>10 %</td>
<td>46 %</td>
</tr>
<tr>
<td>Simple Streams</td>
<td>38 %</td>
<td>10 %</td>
<td>10 %</td>
<td>42 %</td>
</tr>
<tr>
<td>Simple Call Back</td>
<td>56 %</td>
<td>1 %</td>
<td>1 %</td>
<td>42 %</td>
</tr>
<tr>
<td>Histogram</td>
<td>45 %</td>
<td>13 %</td>
<td>11 %</td>
<td>31 %</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>31 %</td>
<td>22 %</td>
<td>17 %</td>
<td>30 %</td>
</tr>
<tr>
<td>Fast Walsh Transform</td>
<td>43 %</td>
<td>9 %</td>
<td>6 %</td>
<td>42 %</td>
</tr>
<tr>
<td>Cpp Overload</td>
<td>100%</td>
<td>0 %</td>
<td>0 %</td>
<td>0 %</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>46 %</strong></td>
<td><strong>10 %</strong></td>
<td><strong>8 %</strong></td>
<td><strong>36 %</strong></td>
</tr>
</tbody>
</table>

Figure 5.6: CUDA benchmark MRU Results for L2 Cache of the NVIDIA Kepler architecture
For the OpenCL benchmarks, the first MRU line on average is comprised of 60% of the total number of access made to L2 cache in the AMD SI architecture. The second and third MRU lines is comprised of 13% and 7%. Since the L2 cache was a 16-way set associative cache, all the remaining MRU lines (MRU3 to MRU15) were grouped together for clearer analysis, and these accounted for 20% of all cache accesses in L2 cache. For the CUDA benchmarks, the first MRU line on average is comprised of 46% of the total number of accesses made to L2 cache in the Kepler architecture. The second and third MRU lines is comprised of 10% and 8%. The remaining MRU lines accounted for 36% of all the cache accesses made to L2 cache. The L2 cache showed lower MRU locality than L1 cache because there were more number of threads sharing 16 MRU lines in L2 cache as opposed to 4 MRU lines in L1 cache.

Similar to the L1 cache results, some OpenCL benchmarks like Bitonic Sort, Binomial Option and Simple Convolution show a high percentage of cache accesses for the first MRU line of 80% or more. The Cpp Overload CUDA benchmark shows the highest percentage of accesses in the most recently used line of 100%. The Floyd Marshall and Matrix Multiplication OpenCL benchmarks show low cache access percentages for the most recently used line of under 35%. Similarly, Transpose and Matrix Multiplication CUDA benchmarks show the lowest percentages of accesses in the most recently used line of under 35%. Likewise, the most recently used line had the highest number of L2 cache accesses. After the first MRU line, the grouped value for MRU3 to MRU15 was the next highest access percentage for L2 cache. This was because of the compounding effect of grouping the MRU lines together.

The most recently used line dominated the number of cache accesses made to L1 and L2 caches for both sets of benchmarks. This indicates that if any line (or block) needs to preserved, transferred or improved in GPU cache then it is the most recently used line in each set. The second MRU line recorded the next highest cache access percentage and hence this line would be the next to be preserved or improved after
CHAPTER 5. RESULTS

the most recently used MRU line.

5.3 Inter-warp and intra-warp Locality Results

The L1 and L2 data cache accesses for inter-warp and intra-warp localities for the OpenCL are tabulated as shown in Tables 5.5 and 5.6. Table 5.5 represents the cache results for the OpenCL benchmarks, and Table 5.6 represents the cache results for the CUDA benchmarks. These results show the correlation between the data cache accesses and the warp based localities.

From Table 5.5 and Figure 5.7, the L1 cache showed higher data cache accesses resulting from intra-warp locality for the OpenCL benchmarks. The average intra-warp and inter-warp cache access percentages for L1 cache were 68% and 32% respectively. Meanwhile, L2 cache showed higher data cache accesses resulting from inter-warp locality. The average intra-warp and inter-warp cache access percentages for L2 cache were 28% and 72% respectively.

The CUDA results from Table 5.6 and Figure 5.8 show how the cache localities differed between L1 and L2 cache. Both L1 and L2 caches showed higher data cache accesses resulting from inter-warp locality. The inter-warp cache access percentages for L2 cache were slightly higher than those for L1 cache. The average intra-warp and inter-warp cache access percentages for L1 cache were 33% and 67% respectively. The average intra-warp and inter-warp cache access percentages for L2 cache were 21% and 79% respectively.
Table 5.5: Inter-warp and Intra-warp Access Percentages for L1 and L2 cache for OpenCL benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>L1 Cache</th>
<th>L2 Cache</th>
<th>L1 Cache</th>
<th>L2 Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Intra-warp %</td>
<td>Inter-warp %</td>
<td>Intra-warp %</td>
<td>Inter-warp %</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td>4 %</td>
<td>96 %</td>
<td>23 %</td>
<td>77 %</td>
</tr>
<tr>
<td>Eigen Value</td>
<td>49 %</td>
<td>51 %</td>
<td>84 %</td>
<td>16 %</td>
</tr>
<tr>
<td>Floyd Marshall</td>
<td>66 %</td>
<td>34 %</td>
<td>8 %</td>
<td>92 %</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>100 %</td>
<td>0 %</td>
<td>100 %</td>
<td>0 %</td>
</tr>
<tr>
<td>Simple Convolution</td>
<td>78 %</td>
<td>22 %</td>
<td>21 %</td>
<td>79 %</td>
</tr>
<tr>
<td>Recursive Gaussian</td>
<td>81 %</td>
<td>19 %</td>
<td>23 %</td>
<td>77 %</td>
</tr>
<tr>
<td>Sobel Filter</td>
<td>75 %</td>
<td>25 %</td>
<td>6 %</td>
<td>94 %</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>69 %</td>
<td>31 %</td>
<td>17 %</td>
<td>83 %</td>
</tr>
<tr>
<td>Mersenne Twister</td>
<td>93 %</td>
<td>7 %</td>
<td>10 %</td>
<td>90 %</td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>87 %</td>
<td>13 %</td>
<td>53 %</td>
<td>47 %</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>68 %</strong></td>
<td><strong>32 %</strong></td>
<td><strong>28 %</strong></td>
<td><strong>72 %</strong></td>
</tr>
</tbody>
</table>

Figure 5.7: OpenCL L1 vector and L2 cache intra-warp and inter-warp access percentages
### Table 5.6: Inter-warp and Intra-warp Access Percentages for L1 and L2 cache for CUDA benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>L1 Cache</th>
<th></th>
<th>L2 Cache</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Intra-warp %</td>
<td>Inter-warp %</td>
<td>Intra-warp %</td>
<td>Inter-warp %</td>
</tr>
<tr>
<td>Vector Add</td>
<td>28 %</td>
<td>72 %</td>
<td>35 %</td>
<td>65 %</td>
</tr>
<tr>
<td>Transpose</td>
<td>53 %</td>
<td>47 %</td>
<td>6 %</td>
<td>94 %</td>
</tr>
<tr>
<td>Simple Streams</td>
<td>17 %</td>
<td>83 %</td>
<td>5 %</td>
<td>95 %</td>
</tr>
<tr>
<td>Simple Call Back</td>
<td>21 %</td>
<td>79 %</td>
<td>32 %</td>
<td>68 %</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>15 %</td>
<td>85 %</td>
<td>3 %</td>
<td>97 %</td>
</tr>
<tr>
<td>Fast Walsh Transform</td>
<td>39 %</td>
<td>61 %</td>
<td>28 %</td>
<td>72 %</td>
</tr>
<tr>
<td>Cpp Overload</td>
<td>65 %</td>
<td>35 %</td>
<td>33 %</td>
<td>67 %</td>
</tr>
<tr>
<td>Histogram</td>
<td>24 %</td>
<td>76 %</td>
<td>23 %</td>
<td>77 %</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>33 %</strong></td>
<td><strong>67 %</strong></td>
<td><strong>21 %</strong></td>
<td><strong>79 %</strong></td>
</tr>
</tbody>
</table>

**Figure 5.8:** CUDA L1 vector and L2 cache intra-warp and inter-warp access percentages
CHAPTER 5. RESULTS

The inter-warp and intra-warp cache access hits and misses for L1 and L2 cache are graphed in Figures 5.9, 5.10, 5.11, and 5.12. Figures 5.9 and 5.10 represent the OpenCL cache results while Figures 5.11 and 5.12 represent the cache results for the CUDA benchmarks.

The OpenCL benchmarks showed a high percentage of misses for both inter-warp and intra-warp localities. This was directly related to the hit ratios recorded for L1 and L2 cache. Since L1 cache showed a higher intra-warp access percentage, the statistics were dominated by intra-warp hits or misses. Contrarily, L2 cache showed a higher inter-warp access percentage which meant that the L2 cache statistics were dominated by inter-warp hits or misses.

The CUDA benchmarks recorded a higher intra-warp access percentage for both L1 and L2 cache which meant that both L1 and L2 statistics were dominated by inter-warp hits or misses. The low hit ratios further suggested a higher number of inter-warp misses, especially for L2 cache which showed a hit ratio of 26%.

Most of the OpenCL and CUDA benchmark results show the fewer number of hits resulting from inter-warp or intra-warp locality. This shows the behavior of the default thread scheduler and the characteristics of the tested workloads. OpenCL benchmarks favoured intra-warp locality for L1 cache and inter-warp locality for L2 cache. CUDA benchmarks favoured inter-warp locality for both L1 and L2 caches. The thread scheduler can be modified to exploit inter-warp locality by allowing threads from different warps to load and reuse the data from cache, to limit the number of inter-warp accesses. This can be done provided all the threads are being processed at the same rate.
CHAPTER 5. RESULTS

**Figure 5.9:** OpenCL L1 vector intra-warp and inter-warp hits and misses

**Figure 5.10:** OpenCL L2 intra-warp and inter-warp hits and misses
CHAPTER 5. RESULTS

Figure 5.11: CUDA L1 intra-warp and inter-warp hits and misses

Figure 5.12: CUDA L2 intra-warp and inter-warp hits and misses
Table 5.7: MRU Results for the CPU [11]

<table>
<thead>
<tr>
<th></th>
<th>1st MRU line</th>
<th>2nd MRU line</th>
<th>Rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>92 %</td>
<td>6 %</td>
<td>2 %</td>
</tr>
</tbody>
</table>

5.4 Comparing the CPU and GPU results

Savaldor Petit et al. [11] studied the cache access patterns of the CPU. This paper records the percentage of cache hits along multiple MRU lines of each set in CPU cache. A variety of CPU benchmarks were tested to understand how the CPU handled different workloads. These benchmark results were tabulated and graphed, and the average cache access percentages for each MRU line in CPU cache were recorded. Table 5.7 summarizes those findings to compare the GPU’s cache access patterns to those of the CPU.

Referring to the CPU statistics from Table 5.7, the first MRU line dominates the cache accesses with the highest percentage of cache accesses. The second MRU line shows the next highest percentage of hits followed by the other MRU lines. The remaining MRU lines are grouped together for clearer results in the table as the percentages were minimal. The first MRU line accounts for an average of 92% of all the cache accesses. The second MRU line accounts for an average of 6%. The remaining set of MRU lines account for 2%.

In comparison to the results of the GPU simulations for AMD and Kepler architectures, the CPU cache access patterns seem to be very similar to the GPU. Recalling Table 5.1 in Section 5.2, the first, second, third and fourth MRU lines account for 84%, 7%, 4% and 5% of all L1 cache accesses on average, respectively for the AMD Southern Islands GPU architecture. Reviewing Table 5.3 in Section 5.2, the first, second and third MRU lines account for 60%, 13% and 7% of all L2 cache accesses on average, respectively for the AMD Southern Islands GPU architecture.
CHAPTER 5. RESULTS

Table 5.8: Comparison of MRU results for the CPU and the GPU [11]

<table>
<thead>
<tr>
<th>Device</th>
<th>Architecture</th>
<th>1st MRU line</th>
<th>2nd MRU line</th>
<th>Rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Itanium2 - L1 cache</td>
<td>92 %</td>
<td>6 %</td>
<td>2 %</td>
</tr>
<tr>
<td>GPU</td>
<td>AMD SI - L1 cache</td>
<td>84 %</td>
<td>7 %</td>
<td>9 %</td>
</tr>
<tr>
<td>GPU</td>
<td>AMD SI - L2 cache</td>
<td>60 %</td>
<td>13 %</td>
<td>27 %</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Kepler - L1 cache</td>
<td>77 %</td>
<td>15 %</td>
<td>8 %</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Kepler - L2 cache</td>
<td>46 %</td>
<td>10 %</td>
<td>44 %</td>
</tr>
</tbody>
</table>

The remaining set of MRU lines (MRU3 to MRU15) in L2 cache of the GPU was grouped together as shown in Table 5.2, and accounted for 20% of all L2 cache accesses. In Table 5.2 in Section 5.2, the first, second, third and fourth MRU lines account for 77%, 15%, 4% and 4% of all L1 cache accesses on average, respectively for the NVIDIA Kepler GPU architecture. Lastly, looking at Table 5.4 in Section 5.2, the first, second and third MRU lines account for 46%, 10% and 8% of all L2 cache accesses on average, respectively for the AMD Southern Islands GPU architecture. The remaining set of MRU lines (MRU3 to MRU15) in L2 cache of the GPU was grouped together and accounted for 36% of all L2 cache accesses. Table 5.8 shows the CPU and GPU comparisons for the respective MRU access percentages. The overall results reflect the GPU cache performance of different sets of workloads over various two GPU architectures.

In conclusion, The MRU0 cache access percentage for the GPU is lower than that for the CPU. This is because GPUs show lower temporal locality. This is a consequence of the high level of parallelism (mainly, threads sharing cache). This indicates that if any line (or block) needs to preserved, transferred or improved in CPU and GPU cache then it is the most recently used line in each set.
Table 5.9: Cache Hit Rates for the CPU [10]

<table>
<thead>
<tr>
<th></th>
<th>L1 Cache Hit Ratio</th>
<th>L2 Cache Hit Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>88 %</td>
<td>93 %</td>
</tr>
</tbody>
</table>

Choo et al. [10] analyzed GPU cache memory traffic for compute workloads and proposed two implementations to optimize cache memory, Multi-CU shared L1 data cache and clustered work-group scheduling. The research used Multi2Sim and ran benchmarks on the AMD SI model of the GPU and the default model of the CPU. The hit ratios for L1 and L2 cache were compared for the GPU and CPU architectures to identify the memory traffic and memory latency. Two new approaches were suggested to improve and optimize CPU and GPU cache memory performance for a variety of compute workloads. Table 5.9 shows the average cache hit rates for L1 and L2 cache of the CPU.

Collating results from Table 5.9, L2 cache indicates a higher hit ratio than L1 cache. The average L1 and L2 cache hit ratios were recorded as 88% and 93%, respectively. A comparison and analysis of the results of the GPU simulations for AMD and Kepler architectures show the CPU cache hit ratios seem to be much higher than those of the GPU.

Table 5.10 shows the L1 and L2 cache hit ratio comparisons between the CPU and GPU results. Figure 5.1 in Section 5.1 indicates 70%, 46% and 35% were the average cache hit ratios for L1 scalar cache, L1 vector cache and L2 cache, respectively, for the AMD Southern Islands GPU architecture. In addition, Figure 5.2 in Section 5.2 shows the average cache hit ratios for L1 cache and L2 cache were 46% and 26%, respectively, for the NVIDIA Kepler GPU architecture. These GPU cache hit ratios for AMD SI and NVIDIA Kepler GPU architectures were analogous. Both architectures show low cache hit ratios, and this is because of how CPU and GPU work differently.

The GPU is comprised of a high number of cores that allow the GPU to process
Table 5.10: Comparison of Cache Hit Ratios for the CPU and the GPU

<table>
<thead>
<tr>
<th>Device</th>
<th>Architecture</th>
<th>L1 Hit Ratio</th>
<th>L2 Hit Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Itanium2</td>
<td>88 %</td>
<td>93 %</td>
</tr>
<tr>
<td>GPU</td>
<td>AMD SI (L1 Scalar) %</td>
<td>70 %</td>
<td>35 %</td>
</tr>
<tr>
<td>GPU</td>
<td>AMD SI (L1 vector) %</td>
<td>46 %</td>
<td>35 %</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Kepler</td>
<td>46 %</td>
<td>26 %</td>
</tr>
</tbody>
</table>

thousands of threads at the same time. The GPU relies heavily on parallel execution for high levels of performance. The parallel execution framework hides longer latencies that may result from lower levels of cache. But this limits the cache performance of the GPU in comparison with the CPU. Contrarily, the CPU relies heavily on the memory hierarchy for higher performance. The CPU consists of fewer cores using large amounts of cache memory that process a low number of threads at a time. The CPU’s reliance on cache memory helps improve performance and show high levels of cache behavior. Hence the CPU will always record higher data cache hit rates than the GPU.

The CPU shows much higher cache hit ratios for both L1 and L2 cache. For L1 cache, the GPU shows approximately 45 % lower average cache hit ratios to those of the CPU. For L2 cache, the GPU shows approximately 65 % lower average cache hit ratios to those of the CPU. This implies that the GPU cache memory traffic for the tested benchmarks is significantly worse that of the CPU. The low GPU L1 and L2 cache hit ratios imply that there is higher memory traffic at lower memory levels like L2 cache and Global Memory [10]. This promotes long memory latency in the GPU architecture.
Chapter 6  

Machine Learning Benchmarks using CUDA

6.1 Design Methodology

Since machine learning benchmarks for GPU simulators were limited, a common set of machine learning algorithms were defined using CUDA as part of this work. These benchmarks are compatible with CUDA libraries and can be run on Multi2Sim’s Kepler GPU architecture.

Some of the common machine learning algorithms used for Convolution Neural Networks (CNNs) were identified, and they are listed in Table 6-1. The machine learning algorithms used for this research were derived from a Deep Neural Network (DNN) implementation in CUDA [20]. This reference consisted of a CUDA GPU kernel containing multiple math algorithms for machine learning. These algorithms were selected based on the working of a CNN and its layers. Both Matrix Multiplication benchmarks were used to parse and process the data for a set of matrices, which was helpful for the Mean Pooling, Max Pooling and Zero Padding benchmarks. The Max Pooling, Mean Pooling and ReLU benchmarks replicated the working of the Pooling layer and ReLU layer of a CNN. The Zero Padding and Linear Interpolation benchmarks replicate the math functions used to process and preserve the data during execution by the CNN.
Individual micro-benchmarks were created using CUDA APIs for each algorithm to understand the cache behavior of the GPU for those respective workloads. The kernels were modified to be compatible with the NVIDIA Kepler architecture. A set of functions were defined to initialize, run and validate the data for each benchmark.

The Matrix Multiplication benchmark was defined to perform basic matrix multiplication of two input matrices and generated a resulting output matrix. The Matrix Element-wise Multiplication was a variation of the Matrix Multiplication benchmark, such that it performed element-wise multiplication of two input matrices. The elements at the each index location of both the input matrices were multiplied to generate the result at the same index location of the output matrix. The Matrix Multiplication and Matrix Element-wise multiplication benchmarks served as a base for any computational matrix algebra operations by a CNN.

In a CNN, there are two main aspects that affect the output of each convolution layer during filtering: stride and padding. The stride offsets along the length of the data and the padding allows the output data to have the same spatial size as the input data. Zero padding is one of the common padding approaches which adds zeros to the output data evenly across the borders to preserve the data size. The Zero Padding benchmark was defined to replicate this functionality of CNNs. A pad input was defined to specify the row/column pad to be placed around the input matrix.

---

**Table 6.1: Defined Machine Learning algorithms in CUDA**

<table>
<thead>
<tr>
<th>Defined Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
</tr>
<tr>
<td>Matrix Element-wise Multiplication</td>
</tr>
<tr>
<td>Max Pooling</td>
</tr>
<tr>
<td>Mean Pooling</td>
</tr>
<tr>
<td>Zero Padding</td>
</tr>
<tr>
<td>Linear Interpolation</td>
</tr>
<tr>
<td>ReLU / Leaky-ReLU</td>
</tr>
</tbody>
</table>

---
The benchmark used an input parameter to specify a surrounding pad value. The default pad value was set to 0 for Zero Padding, but could be altered if needed. The output matrix was generated by adding the zero pad layers around the input matrix.

The *Max and Mean Pooling* were defined to replicate the pooling layer of CNNs. Both benchmarks were defined using a matrix size and a pooling factor as inputs. The pooling factor determined the dimensions of the pooling window while performing the pooling operation (Max Pooling or Mean Pooling). A filtering window was chosen, and the output matrix was generated by taking the mean or max of all the matrix values within that filtering window.

The *Linear Interpolation* benchmark was defined to implement a curve fitting algorithm between a set of points. This algorithm is also used by CNNs to interpolate images by using a filtering window across the source image.

The *Rectifier Linear Unit (ReLU)* benchmark was defined to replicate the ReLU layer of CNNs. This benchmark defined and used a rectifier, which used an activation function to preserve the positive values and remove any negative values during execution. This benchmark was also modified to include and test Leaky ReLU and Noisy ReLU functionality.

The GPU cache behavior for the implemented machine learning benchmarks was studied for three main metrics - L1/L2 cache hit ratios, MRU access percentages and inter-warp/intra-warp locality for L1 and L2 caches. In addition, the benchmarks were compiled and run in the same way as the CUDA SDK benchmarks as shown in Chapter 5. All the machine learning benchmark results were compared to the CUDA general purpose benchmark results as both sets of benchmarks were run on the same Kepler GPU architecture using CUDA libraries. This helped analyze the difference in GPU cache behavior for different workloads without changing the underlying GPU architecture (Kepler).
Table 6.2: L1 and L2 Hit Ratios for Machine Learning Benchmarks

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>L1 Cache Hit Ratio</th>
<th>L2 Cache Hit Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
<td>8 %</td>
<td>50 %</td>
</tr>
<tr>
<td>Matrix Element-wise</td>
<td>69 %</td>
<td>48 %</td>
</tr>
<tr>
<td>Mean Pooling</td>
<td>90 %</td>
<td>52 %</td>
</tr>
<tr>
<td>Max Pooling</td>
<td>92 %</td>
<td>51 %</td>
</tr>
<tr>
<td>Zero Padding</td>
<td>74 %</td>
<td>49 %</td>
</tr>
<tr>
<td>Linear Interpolation</td>
<td>75 %</td>
<td>53 %</td>
</tr>
<tr>
<td>ReLU</td>
<td>25 %</td>
<td>46 %</td>
</tr>
<tr>
<td>Average ML</td>
<td>62 %</td>
<td>50 %</td>
</tr>
<tr>
<td>Average Gen Purpose</td>
<td>46 %</td>
<td>26 %</td>
</tr>
</tbody>
</table>

6.2 Results

6.2.1 L1 and L2 cache hit ratios

Results for the L1 and L2 cache hit ratios were recorded and compared as shown in Table 6.2. Interpreting the results in Table 6.2, the average cache hit ratio for L1 cache was 62% and L2 cache was 50%. Matrix Multiplication and ReLU had the lowest L1 cache hit ratios of 8% and 25%. Mean Pooling and Max Pooling showed the highest L1 cache hit ratios, where Mean Pooling was 90%, and Max Pooling was 92%. The rest of the benchmarks showed 70% to 75% hit ratios for L1 cache. The L2 cache hit ratios for all the tested benchmarks were approximately 45% to 55%. The hit ratios of L1 and L2 cache for the CUDA SDK benchmark suite were recorded to be 46% and 26% respectively. The computational machine learning benchmarks showed better cache hit rates for L1 and L2 cache than both the general purpose CUDA SDK benchmarks.
Table 6.3: MRU Results for the Machine Learning benchmarks on the NVIDIA Kepler architecture for L1 cache

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MRU0</th>
<th>MRU1</th>
<th>MRU2</th>
<th>MRU3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
<td>95 %</td>
<td>2 %</td>
<td>2 %</td>
<td>1 %</td>
</tr>
<tr>
<td>Matrix Element-wise</td>
<td>81 %</td>
<td>9 %</td>
<td>10 %</td>
<td>0 %</td>
</tr>
<tr>
<td>Mean Pooling</td>
<td>69 %</td>
<td>27 %</td>
<td>4 %</td>
<td>0 %</td>
</tr>
<tr>
<td>Max Pooling</td>
<td>70 %</td>
<td>26 %</td>
<td>4 %</td>
<td>0 %</td>
</tr>
<tr>
<td>Zero Padding</td>
<td>100 %</td>
<td>0 %</td>
<td>0 %</td>
<td>0 %</td>
</tr>
<tr>
<td>Interpolation</td>
<td>98 %</td>
<td>2 %</td>
<td>0 %</td>
<td>0 %</td>
</tr>
<tr>
<td>ReLU</td>
<td>84 %</td>
<td>9 %</td>
<td>3 %</td>
<td>4 %</td>
</tr>
<tr>
<td>Average ML</td>
<td>85 %</td>
<td>11 %</td>
<td>3 %</td>
<td>1 %</td>
</tr>
<tr>
<td>Average Gen Purpose</td>
<td>77 %</td>
<td>15 %</td>
<td>4 %</td>
<td>4 %</td>
</tr>
</tbody>
</table>

6.2.2 MRU and LRU Temporal Locality Results

The MRU results of the defined benchmarks for L1 and L2 cache were recorded and compared. Tables 6-3 and 6-4 show the MRU access percentages for L1 cache and L2 cache, respectively. The MRU percentage values for each benchmark are graphed for L1 and L2 cache and compared in Figures 6.1 and Figure 6.2, respectively.

Based on the L1 cache results recorded in Table 6.3 and Figure 6.1, all the tested benchmarks showed similar cache access patterns to the AMD and CUDA SDK benchmark results from Chapter 5. On average, the first MRU line consists of 85% of all the L1 cache accesses for the respective benchmark being tested. The second, third and fourth MRU lines consists of 11%, 3% and 1% of all the L1 cache accesses. Percentages for the first MRU line for some benchmarks like Matrix Multiplication, Zero Padding, and Interpolation showed a percentage of 95% or higher. In addition, the next tier of benchmarks like Matrix Element-wise Multiplication, and ReLU showed approximately 80 to 85% of cache accesses. Finally, other benchmarks and algorithms like Mean Pooling and Max Pooling showed the smallest percentage of cache accesses for the first MRU line at 69% and 70%, respectively.

The MRU access percentages were also compared for L2 cache. The L2 cache
Figure 6.1: Machine Learning benchmark results for the NVIDIA Kepler architecture for L1 cache

Table 6.4: MRU Results for the Machine Learning benchmarks on the NVIDIA Kepler architecture for L2 cache

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MRU0</th>
<th>MRU1</th>
<th>MRU2</th>
<th>MRU3 - MRU15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
<td>39 %</td>
<td>34 %</td>
<td>25 %</td>
<td>2 %</td>
</tr>
<tr>
<td>Matrix Element-wise</td>
<td>83 %</td>
<td>16 %</td>
<td>1 %</td>
<td>0 %</td>
</tr>
<tr>
<td>Mean Pooling</td>
<td>88 %</td>
<td>11 %</td>
<td>1 %</td>
<td>0 %</td>
</tr>
<tr>
<td>Max Pooling</td>
<td>97 %</td>
<td>3 %</td>
<td>0 %</td>
<td>0 %</td>
</tr>
<tr>
<td>Zero Padding</td>
<td>95 %</td>
<td>5 %</td>
<td>0 %</td>
<td>0 %</td>
</tr>
<tr>
<td>Interpolation</td>
<td>98 %</td>
<td>2 %</td>
<td>0 %</td>
<td>0 %</td>
</tr>
<tr>
<td>ReLU</td>
<td>48 %</td>
<td>25 %</td>
<td>17 %</td>
<td>10 %</td>
</tr>
<tr>
<td>Average ML</td>
<td>78 %</td>
<td>14 %</td>
<td>6 %</td>
<td>2 %</td>
</tr>
<tr>
<td>Average Gen Purpose</td>
<td>46 %</td>
<td>10 %</td>
<td>8 %</td>
<td>36 %</td>
</tr>
</tbody>
</table>
showed similar cache access patterns to L1 cache where the first MRU line dominated the total number of accesses made consists of 78% of all L2 cache accesses. The second and third MRU line accounted for the next highest cache accesses made. The second MRU line consists of 14% and the third MRU line consists of 6% of all L2 cache accesses. Finally, the MRU counters for the fourth through sixteenth MRU lines were summed and recorded as one column result, and this accounted for 2% (on average) of all cache accesses made for each benchmark. Additionally, for the first MRU line, benchmarks like Matrix Element-wise Multiplication, Mean Pooling, Max Pooling, Zero Padding and Interpolation show very high cache access percentages between 80% to 100%. Contrarily, Matrix Multiplication and ReLU show lower percentages for the MRU line at 39% and 48%, respectively. The variation of the result is because of the workload and how the GPU processed the data for those algorithms utilizing cache memory.

This showed that the computational machine learning benchmarks performed better in terms of MRU access percentages than the general purpose CUDA SDK bench-
marks. In Chapter 5, the first MRU and second MRU lines were identified as the most important lines to be preserved in cache. They are even more important for the machine learning benchmarks, especially for the L2 cache which shows a very significant difference.

### 6.2.3 Inter-warp and Intra-warp Locality Results

Table 6.5 shows the inter-warp and intra-warp cache access percentages for L1 and L2 caches. The L1 and L2 cache inter-warp and intra-warp access percentages for all benchmarks are displayed as a column graph in Figure 6.3. The results show how the cache localities differed between L1 and L2 cache for a variety of machine learning benchmarks using CUDA. Both L1 and L2 caches showed higher data cache accesses resulting from inter-warp locality, provided that the inter-warp cache access percentages for L2 cache were slightly higher than those for L1 cache. The average intra-warp and inter-warp cache access percentages for L1 cache were 48% and 52%, respectively. The average intra-warp and inter-warp cache access percentages for L2 cache were 40% and 60%, respectively. This showed that the math algorithms being tested favoured inter-warp locality, but the the split between inter-warp and intra-warp locality was more balanced than that of the general purpose benchmarks.

The inter-warp and intra-warp cache access hits and misses for L1 and L2 cache are graphed in Figure 6.4 and 6.5. These cache statistics were directly related to the cache hit ratios recorded for L1 and L2 cache modules. Lower cache hit ratios resulted in more inter-warp and intra-warp misses. This shows the characteristics of the tested workloads on the GPU favored inter-warp locality. The machine learning benchmarks showed higher numbers of intra-hits than the general purpose benchmarks, and this was because the machine learning benchmarks showed higher data cache hit ratios for both L1 and L2 cache.

From the results recorded in Table 6.6, it was concluded that the machine learning
### Table 6.5: Inter-warp and Intra-warp Access Percentages for L1 and L2 cache

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>L1 Cache</th>
<th></th>
<th>L2 Cache</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Intra-warp %</td>
<td>Inter-warp %</td>
<td>Intra-warp %</td>
<td>Inter-warp %</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>43 %</td>
<td>57 %</td>
<td>5 %</td>
<td>95 %</td>
</tr>
<tr>
<td>Matrix Element-wise</td>
<td>27 %</td>
<td>73 %</td>
<td>46 %</td>
<td>54 %</td>
</tr>
<tr>
<td>Mean Pooling</td>
<td>52 %</td>
<td>48 %</td>
<td>58 %</td>
<td>42 %</td>
</tr>
<tr>
<td>Max Pooling</td>
<td>52 %</td>
<td>48 %</td>
<td>30 %</td>
<td>70 %</td>
</tr>
<tr>
<td>Zero Padding</td>
<td>58 %</td>
<td>42 %</td>
<td>77 %</td>
<td>23 %</td>
</tr>
<tr>
<td>Linear Interpolation</td>
<td>26 %</td>
<td>74 %</td>
<td>58 %</td>
<td>42 %</td>
</tr>
<tr>
<td>ReLU</td>
<td>43 %</td>
<td>57 %</td>
<td>6 %</td>
<td>94 %</td>
</tr>
<tr>
<td><strong>Average ML</strong></td>
<td>43 %</td>
<td>57 %</td>
<td>40 %</td>
<td>60 %</td>
</tr>
<tr>
<td><strong>Average Gen Purpose</strong></td>
<td>33 %</td>
<td>67 %</td>
<td>21 %</td>
<td>79 %</td>
</tr>
</tbody>
</table>

**Figure 6.3:** L1 and L2 cache inter-warp and intra-warp access percentages for Machine Learning benchmarks
CHAPTER 6. MACHINE LEARNING BENCHMARKS USING CUDA

Figure 6.4: L1 intra-warp and inter-warp hits and misses for Machine Learning benchmarks

Figure 6.5: L2 intra-warp and inter-warp hits and misses for Machine Learning benchmarks
Table 6.6: Comparison of the warp locality results for the two sets of CUDA benchmarks

<table>
<thead>
<tr>
<th>Cache</th>
<th>Average Values</th>
<th>CUDA SDK</th>
<th>Machine Learning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hit Ratio</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 Cache</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Intra Accesses</td>
<td>46 %</td>
<td>62 %</td>
</tr>
<tr>
<td></td>
<td>Intra Hits</td>
<td>33 %</td>
<td>43 %</td>
</tr>
<tr>
<td></td>
<td>Intra Misses</td>
<td>11 %</td>
<td>25 %</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21 %</td>
<td>18 %</td>
</tr>
<tr>
<td></td>
<td>Inter Accesses</td>
<td>67 %</td>
<td>57 %</td>
</tr>
<tr>
<td></td>
<td>Inter Hits</td>
<td>29 %</td>
<td>38 %</td>
</tr>
<tr>
<td></td>
<td>Inter Misses</td>
<td>39 %</td>
<td>21 %</td>
</tr>
<tr>
<td>L2 Cache</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hit Ratio</td>
<td>26 %</td>
<td>50 %</td>
</tr>
<tr>
<td></td>
<td>Intra Accesses</td>
<td>21 %</td>
<td>40 %</td>
</tr>
<tr>
<td></td>
<td>Intra Hits</td>
<td>7 %</td>
<td>15 %</td>
</tr>
<tr>
<td></td>
<td>Intra Misses</td>
<td>13 %</td>
<td>25 %</td>
</tr>
<tr>
<td></td>
<td>Inter Accesses</td>
<td>79 %</td>
<td>60 %</td>
</tr>
<tr>
<td></td>
<td>Inter Hits</td>
<td>19 %</td>
<td>35 %</td>
</tr>
<tr>
<td></td>
<td>Inter Misses</td>
<td>60 %</td>
<td>25 %</td>
</tr>
</tbody>
</table>

benchmarks showed the higher cache hit ratios for both L1 and L2 caches than the general purpose CUDA benchmarks, and this was mainly because of the difference in workloads.

The results from Table 6.6 were compared to find the breakdown of the L1 cache accesses for two sets of GPU benchmarks for inter-warp and intra-warp data cache localities. An analysis of the results indicates that the machine learning benchmarks had the higher intra-warp access percentage, while the CUDA SDK benchmarks recorded the higher inter-warp access percentage. Furthermore, the machine learning benchmarks showed higher numbers of hits resulting from intra-warp and inter-warp locality than the CUDA SDK benchmarks.

The results from Table 6.6 recorded the L2 cache access breakdowns for the two sets of GPU benchmarks to identify the best set of benchmarks for L2 cache. As a result, machine learning benchmarks had the higher intra-warp access percentage, while the CUDA SDK benchmarks recorded the higher inter-warp access percentage.
Moreover, the highest amount of hits and misses resulting from intra-warp locality were from machine learning benchmarks. In terms of the inter-warp results, the machine learning benchmarks showed the higher numbers of data cache hits resulting from inter-warp locality, whereas the CUDA SDK benchmarks showed the higher numbers of misses resulting from inter-warp locality.

The L1 and L2 cache hit ratios were the highest for the machine learning benchmarks as compared to the CUDA SDK general purpose benchmarks. Consequently, the machine learning benchmarks favoured inter-warp locality for both L1 and L2 caches. Likewise, the machine learning benchmarks had higher numbers of data cache hits resulting from inter-warp and intra-warp localities as the hit ratios for the machine learning benchmarks were higher for both L1 and L2 caches than the CUDA SDK benchmarks. This was because the machine learning benchmarks were mainly computational benchmarks and not general purpose benchmarks.
Chapter 7

Conclusion

Multi2Sim was successfully setup on an Ubuntu OS with all its dependencies to run GPU simulations and analyze GPU cache statistics and behavior. AMD SDK 2.5 and CUDA SDK 6.5 were the two benchmark suites used to run GPU simulations on the AMD SI and NVIDIA Kepler GPU architectures, respectively. The benchmarks helped analyze GPU cache behavior and cache access patterns for a variety of workloads.

Both AMD and NVIDIA Kepler GPU architectures, displayed L1 and L2 data caches hit ratios of under 50%. The GPU cache hit ratios were compared to those of the CPU. The CPU displayed higher hit ratios for both L1 and L2 caches. This is because the GPU relies heavily on parallel thread execution and less on cache memory. The GPU achieves high levels of performance using Thread-level Parallelism (TLP). Contrarily, the CPU has fewer cores and processes a small number of threads at a time but relies more heavily on cache memory to show high performance levels. The CPU achieves high level of performance using Instruction-level Parallelism (ILP).

The MRU results helped identify that the most recently used block dominated the number of cache accesses for both L1 and L2 cache. This indicates that if any line (or block) needs to preserved, transferred, or improved in GPU cache then it would be the most recently used line in each set, for a multi-way associative cache.

The inter-warp and intra-warp locality for cache accesses was studied for both
GPU architectures. This locality was directly dependent on the characteristics of the workloads being tested. The OpenCL benchmarks tested on the AMD SI GPU model showed higher intra-warp locality for L1 cache and high inter-warp locality for L2 cache. Contrarily, the CUDA benchmarks tested on the NVIDIA Kepler GPU model showed higher inter-warp locality for both L1 and L2 caches, more so in L2 cache than L1 cache.

A set of machine learning benchmarks was defined to analyze the GPU behavior for machine learning workloads. These benchmarks were defined using CUDA APIs and were tested on the NVIDIA Kepler GPU model on Multi2Sim. The cache hit ratios, MRU results and inter-warp/intra-warp localities were compared against the general purpose computing benchmarks from the CUDA SDK benchmark suite as both benchmarks were tested on the Kepler GPU model on Multi2Sim. The machine learning benchmarks showed higher cache hit ratios than the CUDA SDK benchmark suite. The general purpose CUDA SDK benchmarks recorded L1 and L2 cache hit ratios of under 50%. However, the machine learning benchmarks exhibited average cache hit ratios for L1 cache of 62.03% and L2 cache of 49.75%. The machine learning benchmarks showed similar cache access patterns and locality to the general purpose benchmark suites. The most recently used line dominated the number of cache accesses for both L1 and L2 caches by more than 75%. The difference in temporal locality was significant for L2 cache more so than L1 cache. The machine learning benchmarks favored inter-warp locality for both L1 and L2 caches.
7.1 Future Work

7.1.1 Unified Memory Model

In a heterogeneous multi-processor system, one of the main performance bottlenecks is that the CPU and GPU are responsible for their own sets of physical memory. Data need to be allocated for both CPU and GPU memory. Furthermore, the data get initialized and transferred from the CPU to the GPU for processing, and then get transferred back to CPU memory. This leads to large overheads and latencies.

NVIDIA introduced a new Unified Memory Model [26, 27] in the latest CUDA6 toolkit and SDK, which helped overcome the bottleneck of having physically separate memory blocks. The Unified Memory Model defines a block of shared memory in DRAM which is shared by the CPU and the GPU. The system handles this unified shared memory for both processors such that there is no explicit need to copy or transfer data from one memory unit to the other. The shared unified memory can be accessed by either CPU or GPU with a single pointer. The system handles any conflicts such that each processor thinks it is accessing memory from its respective memory block. This largely reduces latency and overhead, and helps improve the performance at run-time.

Even though this architecture change has been implemented in hardware in the latest GPUs using CUDA 6, the GPU simulators have not been updated for this architectural change yet. The simulator can be configured for the Unified Memory Model design to study the CPU and GPU cache behavior.

7.1.2 Machine Learning Benchmarks on the GPU

Most of the investigated machine learning benchmark suites [20, 28, 29, 30] required a GPU as a hardware component. None of the benchmark suites were built to run on GPU simulators. Shi Dong et al. [28] defined a Deep Neural Network benchmark suite
for general purpose GPUs. This benchmark suite had a really good implementation of
GPU CUDA kernels for multiple machine learning algorithms. The limitation of the
current version of the benchmark suite was that it could only be run using a GPU as
a hardware component and not GPU simulators. However, the authors are working
on defining GPU CUDA kernels for GPU simulators for the next release. This would
enable running the benchmarks on Multi2Sim to analyze GPU cache behavior for a
variety of machine learning workloads.

Figure 7.1: NVIDIA’s Unified Memory Model [27]
Bibliography


https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-
Architecture-Whitepaper.pdf


https://github.com/xingdi-eric-yuan/cuda-deep-neural-nets

[21] https://github.com/Multi2Sim/m2s-bench-amdsdk-2.5-src

[22] https://github.com/Multi2Sim/m2s-bench-cudasdk-6.5

[23] Wilson W. L. Fung Ivan Sham George Yuan Tor M. Aamodt, “Dynamic Warp
Formation and Scheduling for Efficient GPU Control Flow”, University of British
Columbia, Vancouver, BC, Canada

International Conference on Machine Learning (ICML ’09). ACM, New York, NY, USA

2011.


[29] https://github.com/doody1986/DNNMark

Appendix A: Code Listings

```c
#include "cuda_runtime.h"
#include "stdio.h"

#define BLOCK_SIZE 16
#define SRC_ROWS 128
#define SRC_COLS 128
#define DEST_ROWS 32
#define DEST_COLS 32
#define STRIDE_X 4
#define STRIDE_Y 4
#define N SRC_ROWS * SRC_COLS
#define VALUE 0

__global__ void MeanPooling_kernel( int* src, int* dst, const int rowssrc, const int colssrc,
    const int rowsdst, const int colsdst, const int stridex, const int stridey, const int n)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    while (tid < n)
    {
        int cdst = tid % colsdst;
        int rdst = tid / colsdst;
        int rsrct = rdst * stridey;
        int csrc = cdst * stridex;
        int xend = (csrc + stridex - 1) > (colssrc - 1) ? (colssrc - 1) : (csrc + stridex - 1);
        int yend = (rsrct + stridey - 1) > (rowssrc - 1) ? (rowssrc - 1) : (rsrct + stridey - 1);
        //loc[tid] = (rsrct * colssrc + csrc);
        int val = 0;
        for (int i = rsrct; i <= yend; ++i)
        {
            for (int j = csrc; j <= xend; ++j)
            {
                if (src[i * colssrc + j] > dst[tid])
                {
                    val += src[i * colssrc + j];
                }
            }
        }
        dst[tid] = val / (stridex*stridey);
        tid += stride;
    }
}

void MeanPooling(int* src, int* dst, int src_rows, int src_cols, int dest_rows, int dest_cols, int stridex, int stridey, int n);

void print_matrix( int* matrix, int height, int width);

int main()
{
    int *a = (int*) calloc(SRC_ROWS * SRC_COLS, sizeof(unsigned int));
    int *c = (int*) calloc(DEST_ROWS * DEST_COLS, sizeof(unsigned int));
    int *d = (int*) calloc(DEST_ROWS * DEST_COLS, sizeof(unsigned int));
    int i;
    for (i = 0; i < SRC_ROWS * SRC_COLS; i++)
    {
    
    
```
a[i] = 2*i;
} 

int *dev_a = 0;
int *dev_c = 0;

// Allocate GPU buffers for three vectors (two input, one output).
cudaMalloc((void**)&dev_c, DEST_ROWS * DEST_COLS * sizeof(unsigned int));
cudaMalloc((void**)&dev_a, SRC_ROWS * SRC_COLS * sizeof(unsigned int));

// Copy input vectors from host memory to GPU buffers.
cudaMemcpy(dev_a, a, SRC_ROWS * SRC_COLS * sizeof(unsigned int), cudaMemcpyHostToDevice);
dim3 dimBlock (BLOCK_SIZE, BLOCK_SIZE); // block( blockIdx, blockIDy)
dim3 dimGrid ((SRC_COLS + dimBlock.x - 1) / dimBlock.x,
(SRC_ROWS + dimBlock.y - 1) / dimBlock.y); // grid(gloalsizeX + blockidx -1)/blockidx,
gloalsizeY + blockidy -1)/blockidy);

MeanPooling_kernel<<<dimGrid, dimBlock>>>(dev_a, dev_c, SRC_ROWS, SRC_COLS, DEST_ROWS, DEST_COLS,
STRIDE_X, STRIDE_Y, N);

// Copy output vector from GPU buffer to host memory.
cudaMemcpy(c, dev_c, DEST_ROWS * DEST_COLS * sizeof(int), cudaMemcpyDeviceToHost);

//MeanPooling(a, d, SRC_ROWS, SRC_COLS, DEST_ROWS, DEST_COLS, STRIDE_X, STRIDE_Y, N);

if (flag = true;
// for(int i = 0; i < DEST_ROWS * DEST_COLS; i++)
{
//
// if (c[i] != d[i])
//{
// printf("Verification fail\n");
// printf("i = %d \n", i);
// //printf("A = %d\n", a[i]);
// //printf("B = %d\n", b[i]);
// printf("C = %d\n", c[i]);
// printf("D = %d\n", d[i]);
// flag = false;
// break;
//}
//}
//
// if (flag)
// printf("Verification pass\n");
// printf("Performing Mean Pooling...\n");
// printf("Benchmark executed successfully.\n");
printf("\n");
APPENDIX A. CODE LISTINGS

79
80
Listing A.1: CUDA mean pooling benchmark GPU Kernel
```assembly
.global bitonicSort

.data
FloatMode = 192
IeeeMode = 0

userElements[0] = IMM_UAV, 10, s[4:7]
userElements[1] = IMM_CONST BUFFER, 0, s[8:11]

COMPUTE_PGM_RSRC2:USER_SGPR = 16
COMPUTE_PGM_RSRC2:TGID_X_EN = 1
.arg

u32 theArray 0 uav10 RW
u32 value:stage 16
u32 passOfStage 32
u32 width 48
u32 direction 64

.text

.s_buffer_load_dword s0, s[8:11], 0x04 // 00000000: C2000904
.s_buffer_load_dword s1, s[8:11], 0x18 // 00000004: C2008918
.s_buffer_load_dword s2, s[12:15], 0x04 // 00000008: C2010D04
.s_buffer_load_dword s3, s[12:15], 0x08 // 0000000C: C2018D08
.s_waitcnt lgkmcnt(0) // 00000010: BF8C007F
.s_min_u32 s0, s0, 0x00000000 // 00000014: 8380FF00 0000FFFF
v_mov.b32 v1, s0 // 0000001C: 7E020200
v_mul.i32.i24 v1, s16, v1 // 00000020: 12020210
v_add.i32 v0, vcc, v0, v1 // 00000024: A4000300
v_add.i32 v0, vcc, s1, v0 // 00000028: A4000001
s_sub.i32 s0, s2, s3 // 0000002C: 81800302
v_lshrrev.b32 v1, s0, v0 // 00000030: 2C020200
s_lshl.b32 s1, 1, s0 // 00000034: 8F010081
.s_buffer_load_dword s3, s[12:15], 0x00 // 00000038: C2018D00
v_mul.lo.i32 v1, s1, v1 // 0000003C: D2D60001 02020201
v_lshrrev.b32 v1, 1, v1 // 00000044: 34020201
v_bfe.u32 v2, v0, 0, s0 // 00000048: D2900002 00101000
v_add.i32 v1, vcc, v1, v2 // 00000050: A4020501
v_add.i32 v2, vcc, s1, v1 // 00000054: A4040201
v_lshrrev.b32 v1, 2, v1 // 00000058: 34020202
v_lshrrev.b32 v2, 2, v2 // 0000005C: 34040482
.s_waitcnt lgkmcnt(0) // 00000060: BF8C007F
v_add.i32 v1, vcc, s3, v1 // 00000064: 4A020203
v_add.i32 v2, vcc, s3, v2 // 00000068: 4A040403
t_buffer_load_format.x v3, v1, s[4:7], 0 offset format:[BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]
// 00000006C: EBA01000 80010301
t_buffer_load_format.x v4, v2, s[4:7], 0 offset format:[BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]
// 00000074: EBA01000 80010402
.s_buffer_load_dword s0, s[12:15], 0x10 // 0000007C: C2000D10
.s_lshl.b32 s1, 1, s2 // 00000080: 8F010281
.s_waitcnt lgkmcnt(0) // 00000084: BF8C007F
.s_sub.i32 s2, 1, s0 // 00000088: 81820081
```
Listing A.2: OpenCL bitonic sort benchmark for GPU execution