Architectural and Runtime Enhancements for Dynamically Controlled Multi-Level Concurrency on GPUs

A Dissertation Presented by

Yash Ukidave

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Engineering

Northeastern University
Boston, Massachusetts

December 2015
## Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>List of Acronyms</td>
<td>x</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>xi</td>
</tr>
<tr>
<td>Abstract</td>
<td>xii</td>
</tr>
</tbody>
</table>

### 1 Introduction
1.1 Types of Concurrency on GPUs ............................................................... 2
1.2 Benefits of Multi-level concurrency ......................................................... 3
1.3 Challenges in Implementing Multi-Level Concurrency ............................... 5
   1.3.1 Runtime Level Challenges ................................................................. 5
   1.3.2 Architectural Challenges ..................................................................... 5
   1.3.3 Scheduling Challenges for Multi-Tasked GPUs  ................................... 6
1.4 Contributions of the Thesis .................................................................... 7
1.5 Organization of the Thesis ...................................................................... 9

### 2 Background
2.1 GPU architecture ..................................................................................... 10
   2.1.1 AMD Graphics Core Next (GCN) Architecture ...................................... 11
   2.1.2 NVIDIA Kepler Architecture .............................................................. 12
   2.1.3 Evolution of GPU Architecture .......................................................... 14
2.2 The OpenCL Programming Model ............................................................. 16
   2.2.1 Platform Model ................................................................................. 16
   2.2.2 Memory Model ................................................................................ 17
   2.2.3 Runtime and Driver Model .................................................................. 18
2.3 Multi2Sim: Cycle-level GPU Simulator ..................................................... 20
   2.3.1 Disassembler .................................................................................... 20
   2.3.2 Functional Simulation / Emulation ...................................................... 21
   2.3.3 Architectural Simulator ...................................................................... 22
   2.3.4 OpenCL Runtime and Driver Stack for Multi2Sim .............................. 23
### 2.4 Concurrent Kernel Execution on GPUs
- **2.4.1 Support for Concurrent Kernel Execution on GPUs**

### 2.5 Concurrent Context Execution on GPUs
- **2.5.1 Context Concurrency for Cloud Engines and Data Centers**
- **2.5.2 Key Challenges for Multi-Context Execution**

### 3 Related Work
- **3.1 GPU Modeling and Simulation**
  - **3.1.1 Functional simulation of GPUs for compute and graphics applications**
  - **3.1.2 Architectural Simulation of GPUs**
  - **3.1.3 Power and Area Estimation on GPUs**
- **3.2 Design of Runtime Environments for GPUs**
- **3.3 Resource Management for Concurrent Execution**
  - **3.3.1 Spatial Partitioning of Compute Resources**
  - **3.3.2 QoS-aware Shared Resource Management**
  - **3.3.3 Workload Scheduling on GPUs**
  - **3.3.4 GPU virtualization for Data Center and Cloud Engines**
  - **3.3.5 Scheduling on Cloud Servers and Datacenter**
  - **3.3.6 Address Space and Memory Management on GPUs**

### 4 Adaptive Spatial Partitioning
- **4.1 Mapping Multiple Command Queues to Sub-Devices**
- **4.2 Design of the Adaptive Spatial Partitioning Mechanism**
  - **4.2.1 Compute Unit Reassignment Procedure**
  - **4.2.2 Case Study for Adaptive Partitioning**
- **4.3 Workgroup Scheduling Mechanisms and Partitioning Policies**
- **4.4 Evaluation Methodology**
  - **4.4.1 Platform for Evaluation**
  - **4.4.2 Evaluated Benchmarks**
- **4.5 Evaluation Results**
  - **4.5.1 Performance Enhancements provided by Multiple Command Queue Mapping**
  - **4.5.2 Effective Utilization of Cache Memory**
  - **4.5.3 Effects of Workgroup Scheduling Mechanisms and Partitioning Policies**
  - **4.5.4 Timeline Describing Load-Balancing Mechanisms for the Adaptive Partitioning**
- **4.6 Summary**

### 5 Multi-Context Execution
- **5.1 Need for Multi-Context Execution**
- **5.2 Architectural and Runtime Enhancements for Multi-Context Execution**
  - **5.2.1 Extensions in OpenCL Runtime**
  - **5.2.2 Address Space Isolation**
- **5.3 Resource Management for Efficient Multi-Context Execution**
  - **5.3.1 Transparent Memory Management using CML**
  - **5.3.2 Remapping Compute-Resources between Contexts**
- **5.4 Firmware Extensions on Command-Processor (CP) for Multi-Context Execution**
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Basic architecture of a GPU</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>AMD GCN Compute Unit block diagram</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>NVIDIA Advanced Advanced Stream Multiprocessor (SMX)</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>OpenCL Host Device Architecture</td>
<td>17</td>
</tr>
<tr>
<td>2.5</td>
<td>The OpenCL memory model</td>
<td>18</td>
</tr>
<tr>
<td>2.6</td>
<td>The runtime and driver model of OpenCL</td>
<td>19</td>
</tr>
<tr>
<td>2.7</td>
<td>Multi2Sim’s three stage architectural simulation</td>
<td>21</td>
</tr>
<tr>
<td>2.8</td>
<td>Pipeline model of GCN SIMD unit</td>
<td>22</td>
</tr>
<tr>
<td>2.9</td>
<td>Multi2Sim Runtime and Driver Stack for OpenCL</td>
<td>24</td>
</tr>
<tr>
<td>2.10</td>
<td>Compute Stages of CLSurf</td>
<td>25</td>
</tr>
<tr>
<td>2.11</td>
<td>Kernel scheduling for Fermi GPU</td>
<td>26</td>
</tr>
<tr>
<td>2.12</td>
<td>Multiple hardware queues in GPUs</td>
<td>27</td>
</tr>
<tr>
<td>2.13</td>
<td>Active time of data buffers in GPU applications</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>High-level model describing multiple command queue mapping</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Flowchart for adaptive partitioning handler</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Procedure to initiate compute unit reallocation between kernels</td>
<td>46</td>
</tr>
<tr>
<td>4.4</td>
<td>Case study to demonstrate our adaptive partitioning mechanism</td>
<td>46</td>
</tr>
<tr>
<td>4.5</td>
<td>Workgroup scheduling mechanisms</td>
<td>47</td>
</tr>
<tr>
<td>4.6</td>
<td>Performance speedup using multiple command queue mapping</td>
<td>52</td>
</tr>
<tr>
<td>4.7</td>
<td>Memory hierarchy of the modeled AMD HD 7970 GPU device</td>
<td>52</td>
</tr>
<tr>
<td>4.8</td>
<td>L2 cache hit rate when using multiple command queue mapping</td>
<td>53</td>
</tr>
<tr>
<td>4.9</td>
<td>Effect of different partitioning policy and workgroup scheduling mechanisms</td>
<td>54</td>
</tr>
<tr>
<td>4.10</td>
<td>Timeline analysis of compute-unit reassignment</td>
<td>56</td>
</tr>
<tr>
<td>4.11</td>
<td>Overhead for CU transfer</td>
<td>57</td>
</tr>
<tr>
<td>5.1</td>
<td>Resource utilization and execution time of Rodinia kernels</td>
<td>60</td>
</tr>
<tr>
<td>5.2</td>
<td>Position of CML and context dispatch in the runtime</td>
<td>62</td>
</tr>
<tr>
<td>5.3</td>
<td>Address space isolation using CID</td>
<td>63</td>
</tr>
<tr>
<td>5.4</td>
<td>Combined host and CP interaction for TMM</td>
<td>65</td>
</tr>
<tr>
<td>5.5</td>
<td>TMM operation, managing three concurrent contexts</td>
<td>66</td>
</tr>
<tr>
<td>5.6</td>
<td>Speedup and resource utilization improvement using multi-context execution</td>
<td>72</td>
</tr>
<tr>
<td>5.7</td>
<td>Compute units utilization using multi-context execution</td>
<td>73</td>
</tr>
<tr>
<td>5.8</td>
<td>Increase in execution time for multi-context execution</td>
<td>74</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>5.9 Global memory bandwidth utilization using multi-context execution</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>5.10 Timeline of global memory usage for contexts from different mixes</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>5.11 Evaluation of different global memory management mechanisms</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>5.12 Performance of pinned memory updates over transfer stalling</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>5.13 Timeline analysis of the compute unit remapping mechanism</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>5.14 L2 cache hit rate for multi-context execution</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>5.15 Conflict miss in L2 cache as % of total misses</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>5.16 TLB hit rate for multi-context execution</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>5.17 Analysis of firmware tasks for the CP</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>5.18 Execution performance of firmware on CP cores</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>5.19 Context handling delay added by the CP</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>5.20 L2 cache hit-rate for static partitioned cache and CID-based cache</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>6.1 Need for supporting QoS awareness on multi-tasked GPU</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td>6.2 Opportunity for achieving QoS through resource management</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td>6.3 The three-layer architecture of Virtuoso</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>6.4 QoS settings register (OSR) and QoS information table (QIT)</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>6.5 Calculating the Token Release Register value using impact analysis</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>6.6 Resource allocation using tokens</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>6.7 Calculation of the appropriate QoS reallocation interval</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>6.8 Performance of Virtuoso based QoS enforcement</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>6.9 IPC and STP improvement using Virtuoso</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>6.10 Improvement in STP with Virtuoso guided CU allocation</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>6.11 L2 Cache performance when using Virtuoso</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>6.12 Global memory throughput improvement using DRAM cache</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>6.13 Timeline highlighting the behavior-aware resource allocation when using Virtuoso</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>6.14 Oversubscription of GPU memory using Virtuoso and TMM</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>6.15 Sensitivity analysis of high priority constraints</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>6.16 Side-effects of aggressive selection of high priority constraints</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>6.17 Improving global memory throughput using QoS-aware interconnect network</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>7.1 Performance using interference-oblivious scheduling</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>7.2 GPU Remoting Architecture and Implementation</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>7.3 SVD on a rating Matrix</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>7.4 Mystic Architecture</td>
<td>115</td>
<td></td>
</tr>
<tr>
<td>7.5 Collaborative Filtering for Predicting Performance</td>
<td>117</td>
<td></td>
</tr>
<tr>
<td>7.6 Comparing the clustering of applications using the Actual and Predicted metrics</td>
<td>119</td>
<td></td>
</tr>
<tr>
<td>7.7 Performance impact using Mystic with medium load</td>
<td>121</td>
<td></td>
</tr>
<tr>
<td>7.8 Performance of Mystic when scaling the number of compute nodes</td>
<td>122</td>
<td></td>
</tr>
<tr>
<td>7.9 Performance of our three schedulers with light and heavy load</td>
<td>122</td>
<td></td>
</tr>
<tr>
<td>7.10 System level performance of schedulers</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>7.11 GPU utilization on server</td>
<td>124</td>
<td></td>
</tr>
<tr>
<td>7.12 Variation in runtime for applications across 100 sequences</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>7.13 Scheduling Decision Quality</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>7.14 Quality of Collaborative Filtering Based Prediction</td>
<td>126</td>
<td></td>
</tr>
</tbody>
</table>
7.15 Overheads incurred with the Mystic scheduler
# List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Evolution of NVIDIA GPU architecture from Fermi to Kepler</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Evolution of AMD GPU architecture from Evergreen to GCN</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>The device configuration of the AMD HD Radeon 7970 GPU</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>Device configuration of AMD R9-290X GPU</td>
<td>70</td>
</tr>
<tr>
<td>5.2</td>
<td>Summary of benchmarks used for evaluation</td>
<td>71</td>
</tr>
<tr>
<td>5.3</td>
<td>Workload mix characteristics and with varied compute characteristics</td>
<td>71</td>
</tr>
<tr>
<td>5.4</td>
<td>Workload mix with varied sharing levels.</td>
<td>72</td>
</tr>
<tr>
<td>5.5</td>
<td>Area and power overheads of the CP alternatives</td>
<td>81</td>
</tr>
<tr>
<td>5.6</td>
<td>L2 cache area overhead</td>
<td>83</td>
</tr>
<tr>
<td>5.7</td>
<td>TLB area overhead</td>
<td>83</td>
</tr>
<tr>
<td>6.1</td>
<td>QoS parameters and resource token settings used in Virtuoso evaluation</td>
<td>96</td>
</tr>
<tr>
<td>6.2</td>
<td>Workload mixes used to evaluate our Virtuoso to support QoS goals</td>
<td>96</td>
</tr>
<tr>
<td>6.3</td>
<td>QoS success rates when using Virtuoso based resource management</td>
<td>105</td>
</tr>
<tr>
<td>7.1</td>
<td>Th metrics used for profiling resource usage</td>
<td>116</td>
</tr>
<tr>
<td>7.2</td>
<td>Server configuration for evaluating Mystic</td>
<td>119</td>
</tr>
<tr>
<td>7.3</td>
<td>Multi-application workloads from 8 application suites</td>
<td>120</td>
</tr>
</tbody>
</table>
List of Acronyms

**NDR**  OpenCL NDRange

**CU**  Compute Unit

**CML**  Context Management Layer

**CP**  Command Processor

**TMM**  Transparent Memory Management

**IPC**  Instructions Per cycle

**STP**  System Throughput

**GMT**  Global Memory Throughput
Acknowledgments

I would thank my parents Vandana, and Sanjeev for supporting me through this process. I thank my aunt Sadhana for being an important part of my family and supporting my work. I thank them for understanding my frustrations and keeping me motivated to complete this work.

A special thanks to all my colleagues at NUCAR group, especially Charu Kalra, Perhaad Mistry, Dana Schaa, and Xiangyu Li for their contribution towards the concepts and ideas in this thesis. I would also like to thank our collaborators Vilas Sridharan (AMD), John Kalamatianos (AMD), and Norman Rubin (NVIDIA) for their constructive feedback on our work.

I would acknowledge my friends Arvind, Parag, Niranjan, Kunal, Sushant, Venkatesh, and Disha who helped me remain sane in this process.

Finally, this thesis could not have never been possible without the constant support and guidance of my advisor Dr. David Kaeli. I thank him for his continued encouragement and feedback to improve the quality of our work.
Abstract

Architectural and Runtime Enhancements for Dynamically Controlled Multi-Level Concurrency on GPUs

by

Yash Ukidave

Doctor of Philosophy in Electrical and Computer Engineering Northeastern University, December 2015

Dr. David Kaeli, Adviser

GPUs have gained tremendous popularity as accelerators for a broad class of applications belonging to a number of important computing domains. Many applications have achieved significant performance gains using the inherent parallelism offered by GPU architectures. Given the growing impact of GPU computing, there is a growing need to provide improved utilization of compute resources and increased application throughput. Modern GPUs support concurrent execution of kernels from a single application context in order to increase the resource utilization. However, this support is limited to statically assigning compute resources to multiple kernels, and lacks the flexibility to adapt resources dynamically. The degree of concurrency present in a single application may also be insufficient to fully exploit the resources on a GPU.

Applications developed for modern GPUs include multiple compute kernels, where each kernel exhibits a distinct computational behavior, with differing resource requirements. These applications place high demands on the hardware and may also include strict deadline constraints. Their multi-kernel nature may allow for concurrent execution of kernels on the device. The use of GPUs in cloud engines and data centers will require a new class of GPU sharing mechanisms enforced at an application context level. What is needed are new mechanisms that can streamline concurrent execution for multi-kernel applications. At the same time, concurrent-execution support must be extended to schedule multiple applications on the same GPU. The implementation of application level and kernel level concurrency can deliver significantly improved resource utilization and application throughput. A number of architectural and runtime-level design challenges need to be addressed to enable efficient scheduling and management of memory and compute resources to support multiple levels concurrency.
In this thesis, we propose a dynamic and adaptive mechanism to manage multi-level concurrency on a GPU. We present a new scheduling mechanism for dynamic spatial partitioning on the GPU. Our mechanism monitors and guides current execution of compute workloads on a device. To enable this functionality, we extend the OpenCL runtime environment to map multiple command queues to a single GPU, and effectively partition the device. The result is that kernels, that can benefit from concurrent execution on a partitioned device, can more effectively utilize more of the available compute resources of a GPU. We also introduce new scheduling mechanisms and partitioning policies to match the computational requirements of different applications.

We present new techniques that address design challenges for supporting concurrent execution of multiple application contexts on the GPU. We design a hardware/software-based mechanism to enable multi-context execution, while preserving adaptive multi-kernel execution. Our **Transparent Memory Management (TMM)** combines host-based and GPU-based control to manage data access/transfers for multiple executing contexts. We enhance the **Command Processor (CP)** on the GPU to manage complex firmware tasks for dynamic compute resource allocation, memory handling and kernel scheduling. We design a hardware-based scheme that modifies the L2 cache and TLB to implement virtual memory isolation across multiple contexts. We provide a detailed evaluation of our adaptive partitioning mechanism while leveraging multi-context execution using a large set of real-world applications. We also present a hardware-based runtime approach to enable profile-guided partitioning and scheduling mechanisms. Our partitioning/scheduling mechanism uses machine learning to analyze the current execution state of the GPU. We improve the effectiveness of adaptive partitioning and TMM by tracking execution time behavior of real world applications.
Chapter 1

Introduction

The GPU (Graphics Processing Unit) has become ubiquitous as an accelerator device. Heterogeneous computing systems are being developed with multi-core CPUs, and single or multiple GPUs. The parallelism offered by the GPU architecture has attracted a lot of developers to port their applications to a GPU. The large number of cores on the GPU allows for launching thousands of compute threads to execute in a SIMD (Single-Instruction Multiple Data) manner. The large amount of data parallelism they offer has been leveraged by many applications belonging to a number of computing domains such as signal processing, molecular dynamics and high performance computing (HPC) [39, 59, 115, 116, 139].

Applications for GPUs are commonly developed using programming frameworks such as Kronos’s OpenCL (Open Compute Language) and NVIDIA’s CUDA (Compute Unified Device Architecture). Both OpenCL and CUDA are based on high-level programming constructs of the C and C++ languages. The data parallel and computationally intensive portions of an application are offloaded to the GPU for accelerated execution. The parallel portions of the application code which execute on the GPU are called kernels. These programming frameworks offer a rich set of runtime APIs (Application Programming Interfaces) to allow a developer to write optimized kernels for execution on GPUs. Applications developed using OpenCL or CUDA can launch a large number of threads, with each thread executing the same kernel code. This model provides for massive data parallelism by maximizing thread-level concurrency. Popular applications in scientific computing, molecular dynamics, graph analytics and medical imaging have been ported to GPUs to achieve attractive speedups.

The range of these applications is not limited to scientific computing or HPC domains. There has been growing use of GPUs in cloud engines, data centers, smart phones and hand held tablets, giving rise to a new class of applications for these platforms. The performance require-
CHAPTER 1. INTRODUCTION

ments of these applications go well beyond traditional thread-level concurrency on GPUs. These applications require maximizing kernel and application throughput and efficiency by fully utilizing the compute resources on the GPU. The varying degrees of parallelism present in this demanding class of applications requires us to explore concurrency at multiple levels of execution on the GPU. Future GPUs need to accommodate the growing demands of flexible concurrency in applications, and have to evolve architecturally to fulfill such demands. In this thesis, we will consider how best to exploit concurrency at multiple levels and explore enhancements to the GPU architectures and runtime environments deliver this concurrency.

1.1 Types of Concurrency on GPUs

GPU architectures have been known to support concurrent execution models. The architecture can be enhanced to support concurrency at multiple levels of execution. The different types of concurrency on a GPU include:

1. Thread-level concurrency to use device-level parallelism
2. Kernel-level concurrency for multi-kernel execution
3. Context-level concurrency to support multiple applications on the GPU

Most present-day GPUs are optimized to provide thread-level concurrency, which allows applications to launch thousands of threads on the GPU. Each thread, also known as a workitem, is an instance of the computation kernel. The threads execute in parallel on the large number of compute cores available on the GPU. These threads follow a SIMD execution pattern, where each thread operates on different data while performing the same compute function. Thread-level concurrency on GPUs was primarily designed for graphics computations, where each graphics kernel (shader) performed operations on the pixels in a frame. The same technique has been adopted for general purpose applications on GPUs, which perform computations on large array-based data structures. The GPU programming models such as OpenCL and CUDA expose thread-level concurrency to the developer. Applications have been configured to use this data-parallelism to achieve promising speedups on the GPU. Support for thread-level concurrency has also prompted many CPU-oriented multi-threaded applications to be ported to GPUs for maximizing execution throughput.

In kernel-level concurrency, an application can launch multiple kernels on the same GPU for simultaneous execution. The modern GPU architectures execute multiple data-independent kernels from the same application if sufficient compute resources are available. To facilitate the hosting of kernels, GPUs are equipped with multiple hardware queues, which handle the mapping of kernels
CHAPTER 1. INTRODUCTION

to the compute resources. Applications with multiple kernels for stage-based computation can execute their kernels concurrently on the GPU. The developer has to carefully pipeline the execution of these compute stages. GPU vendors such as NVIDIA support concurrent kernel execution using their HyperQ technology. NVIDIA provides 32 hardware queues which can theoretically launch 32 concurrent kernels, given that enough compute resources are available [109]. AMD GPUs use the Asynchronous Compute Engine (ACE) units to manage multiple kernels for concurrent execution [95]. The kernel-level concurrency is limited by the availability of compute resources on the GPU, and leads to a fixed static partitioning of the device when executing multiple kernels. Kernel level concurrency adds another level of execution parallelism over thread level concurrency. This improves compute resource utilization on the GPU and helps to improve overall kernel throughput of the application.

Application-level concurrency allows the execution of multiple host applications to execute their workloads simultaneously on the GPU. Current GPU architectures allow only a single host application to execute workload on the device at any given instance. Alternative schemes based on interleaved execution and time-sliced execution of multiple applications have been evaluated on the GPU [22, 73]. None of these schemes guarantee the physical sharing of compute resources by simultaneously executing host applications. Application-level concurrency benefits the datacenter and cloud engine environments, where the GPU is provided as a service to multiple users. Sharing of the GPU by applications from multiple users could potentially improve the resource utilization on the GPU and also increase application throughput.

1.2 Benefits of Multi-level concurrency

Computation on GPUs has been traditionally limited to thread-level parallelism for operating on large data sets with thousands of concurrently executing threads. Concurrency at multiple levels of execution provides the developers with added flexibility. Kernel-level concurrency can be used for executing several small-sized kernels simultaneously. The Madness framework launches multiple small-sized kernels simultaneously for solving multi-dimensional integrals and differential equations on GPUs [134]. Similarly, multiple kernels of different sizes are used for Biometric Optical Pattern Matching [17] and Speeded-Up Robust Features (SURF) on GPUs [98]. This class of applications typically cannot utilize the entire GPU when executing a single kernel. Many applications use multi-sized kernels, opening up the potential for concurrent execution. Such applications can benefit from using an adaptive policy of resource allocation with respect to the size of the computation. We describe a novel adaptive spatial partitioning mechanism for GPUs in Chapter 4. The
CHAPTER 1. INTRODUCTION

developer is given the freedom to assign the number of resources required by the application or can let the task be controlled by the runtime. Context-level concurrency empowers the administrators and schedulers to implement a flexible approach for physical sharing of GPU resources between applications from multiple hosts. Multi-context sharing of a GPU is extremely useful in cloud engines and data centers.

The major advantage of multi-level concurrency can result in more effective utilization of compute resources on the GPU. Simultaneous execution of multiple kernels from the same or different application contexts helps in achieving a fully utilized GPU. Dynamic assignment of compute resources at runtime help to better utilize the GPU. Our proposed mechanism provides a hardware-level approach to achieve runtime-level partitioning and load balancing. For multiple applications executing on the GPU, an advanced memory management mechanism is described in Chapter 5. The mechanism provides each application with a complete view of the GPU memory. It manages the data-buffer transfer between the CPU and the GPU using an approach that is transparent to the user. Load balancing is used to avoid GPU memory overflow. Our memory management mechanism employs a GPU-controlled and CPU-controlled buffer management, where a GPU can also initiate decisions for data transfers.

Among the multiple advantages of multi-level concurrency, the improvement in overall execution throughput is paramount. Thread-level parallelism provides impressive speedups for applications ported to the GPU. But, the addition of kernel-level concurrency adds to the kernel throughput of the applications. Concurrent execution of kernels also reduces the time needed for powering the GPU, potentially achieving considerable energy savings. Data centers and cloud engines have strict energy budgets and demand a highly efficient scheduler to achieve the maximum utilization of resources while minimizing energy consumption. Enabling multiple applications to execute concurrently on GPUs, and physical sharing of compute resources, can further increase overall application throughput, and may also reduce energy consumption. Each level of concurrency is built using the key features of the lower levels of execution. For example, supporting application-level concurrency enables each application to launch multiple kernels that execute simultaneously on the GPU, enabling the launch of a larger number of concurrent threads. In this thesis, we consider two complementary approaches to implement multi-level concurrency. In the first approach, we design the software support and runtime framework to support different programming constructs to provide multi-level concurrency. In the second, we implement hardware support on GPUs to efficiently manage concurrency at every level, while maintaining fairness and correctness in execution.
CHAPTER 1. INTRODUCTION

1.3 Challenges in Implementing Multi-Level Concurrency

We describe the challenges for our implementation of the two proposed approaches to enable multi-level concurrency.

1.3.1 Runtime Level Challenges

GPU programming languages such as OpenCL and CUDA provide a rich set of Application Programming Interfaces (APIs) for developing a GPU-based application [3, 100]. Support for kernel-level and application-level concurrency is exposed to the user when using a runtime API. New constructs must be defined in the programming framework that allow the user to design applications for the added levels of concurrency. The API libraries of the OpenCL and CUDA runtimes are managed by the Khronos group and NVIDIA, respectively [3, 100]. One important challenge is to conform to the standards defined by the API when implementing new constructs in programming frameworks. The new constructs for multi-level concurrency should provide a simple, yet efficient, mechanism of controlling the behavior of the application on the GPU. In this thesis, we implement the new constructs as a part of the existing API standards. This approach achieves our desired objectives, while using the standards defined for the programming language.

The runtime library interprets each of the APIs and generates several system-level tasks to perform the function requested by the API. The system-level tasks are used to send commands to the GPU using a device driver. The management of multiple kernels to support kernel-level concurrency is done at the lower runtime layer. The existing runtime modules do not provide support for the advanced scheduling and memory transfer mechanisms required for kernel-level and application-level concurrency. The runtime modules and the driver modules must be extended in order to take advantage of the multiple levels of concurrency. These extensions have to be implemented such that they do not impact the operational performance or correctness of the existing modules. Our work addresses the challenges posed by the API module, runtime module, and the driver, developing an efficient solution.

1.3.2 Architectural Challenges

The GPU architecture has evolved over the years to manage computations and graphics using the same Instruction Set Architecture (ISA). The implementation of new features on GPUs and provisions for multi-level concurrency introduce some key architectural challenges.

Traditionally, the GPU hardware, which includes the memory hierarchy, compute resources, and schedulers, were designed to execute a single kernel belonging to a single application...
context at any instance of time. GPUs would serialize the execution of kernels from a multi-kernel application. This resulted in wastage of compute resources and decreased kernel throughput. Modern GPU vendors such as NVIDIA and AMD have overcome this limitation by allowing multi-kernel execution using multiple hardware queues. But, the current approaches follow a strict partitioning scheme and do not account for changes in the characteristics of different kernels executing concurrently. Using a fixed partitioning can lead to unfair allocation of resources and can starve small kernels. For a true adaptive partitioning of resources, the scheduler on the GPU should be aware of the characteristics of the kernel. This requires changes to the scheduler and resource allocation unit on the GPU. The scheduling mechanism and adaptive partition handling mechanism are discussed in Chapter 4. The current GPU architecture has to be modified to support inter-kernel communication without the need for global memory synchronization. Implementation of an appropriate communication channel requires modifications to the memory object management on the GPU.

The next level of concurrency on GPUs focuses on execution of multiple application contexts running concurrently. Current GPUs do not provide support for executing multiple applications simultaneously. Enabling multi-kernel execution requires major architectural changes to support dynamic resource management, memory management for each context, and separate address space management for each context. Every context executing on the device must be allocated with a set of compute resources which can be dynamically adjusted during runtime. This requires a flexible compute resource reallocation policy to be implemented on the GPU. Each executing context may impose large memory requirements based on the input data-set sizes. A sophisticated memory management mechanism is required to monitor and control the global memory usage on the GPU. The GPU must be extended with a management processor to monitor and control the global memory transactions for multiple contexts. Another architectural challenge for multi-context execution is to isolate virtual address spaces for each executing context. The memory hierarchy and address translation mechanism on the GPU have to be modified to allow for simultaneous memory accesses to multiple virtual address spaces.

1.3.3 Scheduling Challenges for Multi-Tasked GPUs

Introducing multi-level concurrency on GPUs allows multiple application contexts and multiple kernels to co-execute on the GPU. While this support is required to improve the GPU occupancy, and to increase overall system throughput, it also leads to resource management and scheduling challenges. Many applications of varying compute behavior can be simultaneously scheduled on the GPUs which support multi-level concurrency. The performance of these applications may be dependent on multiple shared resource on the GPUs. The applications may contend for compute re-

CHAPTER 1. INTRODUCTION
CHAPTER 1. INTRODUCTION

sources, memory-subsystem, and the interconnect network on the GPU. The framework supporting multi-level concurrency on the GPU must be aware of the resource requirements of the applications, and should reduce the contention for resources. This may lead to interference among co-executing applications, which can degrade their individual performance and reduce system throughput. In addition to disparate compute behavior, the applications executing on the multi-tasked GPU can have different priority levels or may require strict Quality-of-Service (QoS) guarantees (e.g., graphics applications). These QoS requirements can be in the form of min/max performance or a dedicated resource allocation. The multi-level concurrency supported on the GPU must be aware of contention and must also provide an interface for applications to declare their QoS goals. The frameworks described in Chapter 6 and Chapter 7 present mechanisms which realize QoS-aware resource management, and machine learning based techniques to predict interference among co-scheduled applications on a GPU.

1.4 Contributions of the Thesis

The key contributions of this thesis are summarized below:

- We design and implement a new workgroup scheduling mechanism and adaptive partitioning handling scheme on a model of the AMD Graphics Core Next (GCN) architecture GPU using Multi2Sim, a cycle level GPU simulator. We define an OpenCL runtime API interface to expose the programming constructs for using adaptive partitioning to the programmer. We design a hardware-level compute resource reallocation scheme to execute adaptive partitioning. To the best of our knowledge, we are the first work to define, implement, and evaluate an adaptive partitioning scheme on a GPU by providing a combined runtime-level and architecture-level approach.

- Our proposed partitioning mechanism is evaluated using real world applications taken from signal and image processing, linear algebra, and data mining domains. For these performance-hungry applications we achieve a 3.1X performance speedup and a 41% improvement in resource utilization, using a combination of the proposed scheduling scheme versus relying on the conventional GPU runtime.

- We provide support for multi-context execution using the Multi2Sim simulation framework. We design and implement a hardware and software-level solution to manage multi-context execution. Our design of the Transparent Memory Management (TMM) combines host-based and GPU-based support to manage data access/transfers for multiple executing contexts. We
CHAPTER 1. INTRODUCTION

enhance the Command Processor (CP) on the GPU to manage complex firmware tasks for
dynamic compute resource management, memory handling and kernel scheduling. We evaluate a hardware-based scheme that modifies the L2 cache and TLB to provide virtual memory isolation across multiple contexts.

• We evaluate the performance of our design using workload mixes of real-world applications executed concurrently. Our approach shows an impressive speedup of 1.5X (on average) and 2.1X (peak). Further, we improve compute unit utilization by 19% (on average) and 21% (peak). The TMM shows improvements of 22% and 40.7% over the previously proposed software-managed and pinned-memory approaches, respectively. We achieve these benefits using a CP core which adds only a 0.25% increase in die area on the GPU and impact L2 cache area by less than 0.4%.

• We implement Virtuoso, a QoS-aware, resource management framework for multi-tasked GPUs. Virtuoso defines a novel hardware and software level mechanism for resolving resource contention while providing QoS guarantees for co-executing applications. Virtuoso uses dynamic performance monitoring, runtime analysis, and QoS policies to guide the allocation of multiple hardware resources including CUs, L2/L3 caches, and DRAM memory.

• Virtuoso improves the execution performance of applications by 17.8%, with an average 20% improvement in CU allocation over spatial multi-tasking. Similarly, the use of a priority-based and performance-based cache allocation mechanism improves the cache performance by 1.3x over spatial multi-tasking.

• We realize Mystic, a runtime framework enabling interference-aware scheduling for GPU workloads. Mystic utilizes a collaborative filtering approach from machine learning to accurately characterize an unknown application, and then predicts interference it would cause when co-scheduled with other executing applications. Mystic uses the predicted interference to guide the co-schdueling of applications on GPUs with multi-level concurrency. It exploits the data obtained from previously executed applications, coupled with an offline-trained application corpus, and a minimal signature obtained from the incoming application to make an informed prediction.

• Mystic uses 55 distinct real world applications belonging to a variety of GPU benchmark suites, executing them on a cluster with 32 NVIDIA K40m GPUs. Mystic is able to honor more than 90% of QoS guarantees for our applications, while improving overall system throughput by 27.5% over interference-oblivious schedulers.
1.5 Organization of the Thesis

The central focus of this thesis is to design and implement architectural enhancements and runtime-level constructs to facilitate concurrency at multiple levels (i.e., thread-level, kernel-level, and application-level) on GPUs. The remainder of the thesis is organized as follows: Chapter 2 presents the background information on GPU architecture and advanced features introduced for GPUs, along with a description of the OpenCL programming framework. We describe the Multi2Sim simulation infrastructure, kernel-level concurrency on GPUs, and concurrent context execution for cloud and data-center environments. In Chapter 3, we present related work in GPU modeling and simulation, including work for concurrent kernel execution on GPUs, describing various software and hardware approaches used to achieve such concurrency. We also survey the literature of related work in the area of concurrent application support on a single GPU and the runtime level techniques for virtualizing GPUs in a data center. In Chapter 4, we describe the runtime-based design of the adaptive partitioning on GPUs along with an analysis of workgroup scheduling mechanisms for concurrent kernel execution on GPUs. Chapter 5 describes our combined hardware and runtime-based implementation for multi-context concurrency on GPUs. We discuss our implementation of a dynamic resource allocator and a sophisticated memory management mechanism to manage multiple contexts on a GPU. Chapter 6 describes the Virtuoso mechanism to achieve QoS-aware resource allocation on a multi-tasked GPU. We present the implementation of our dynamic hardware monitors and priority guided management scheme for multiple shared resources on the GPU. Chapter 7 details the Mystic mechanism which provides techniques for interference aware scheduling on a multi-tasked GPU cluster. Mystic uses state-of-the-art machine learning mechanisms to predict the performance of the applications and detect interference among active applications to guide the scheduler.
Chapter 2

Background

In this chapter, we present background information and concepts used throughout this thesis, including GPU architecture, the OpenCL programming framework, the Multi2sim simulation framework, concurrent kernel execution on GPUs, and multi-context concurrency for cloud engines, data centers and inter-operable graphics and compute.

2.1 GPU architecture

The Graphics Processing Unit (GPU) has been traditionally used as an accelerator for computer graphics rendering. The device architecture of the GPU is primarily designed for efficient execution of applications on the graphics processing pipeline for OpenGL and DirectX [91, 133, 157]. The program executing on the GPU for graphics rendering is called a shader. The GPU Instruction Set Architecture (ISA) supports instructions and hardware for distinct shaders executed by OpenGL or DirectX. GPU architectures are well suited for processing matrix-based computations which represent a graphics frame. The GPUs use a large number of computation cores to facilitate simultaneous processing of multiple pixels of a frame. GPU cores represent vector processors with (Single-Instruction-Multiple-Data (SIMD) style computation. Instructions on the GPU are pre-dominantly vector instructions performed by each core.

The massive amounts of data parallelism provided by GPUs has attracted developers to port general purpose applications to GPUs. Early on, many scientific computations have been ported to the GPU using OpenGL shaders before the introduction of high-level language programming frameworks such as OpenCL and CUDA [52, 80, 82, 145]. The data for such computations was stored in the form of RGBA pixel data and accessed using texture-based operations using OpenGL and DirectX. The popularity of general purpose computing on GPUs led to the introduction of
CHAPTER 2. BACKGROUND

compute-unified ISAs for GPUs. The new ISA supported orthogonality, allowing the use of the same instructions for compute and graphics [89, 112].

Modern day GPUs provide a compute-unified ISA and support specialized programming frameworks (e.g., OpenCL, CUDA, OpenACC etc.) for general purpose computing. The GPU architecture of the latest state-of-art GPUs includes hardware modules specifically designed to better handle general purpose computations. We explain the architecture of the latest families of GPUs from AMD and NVIDIA.

2.1.1 AMD Graphics Core Next (GCN) Architecture

The latest AMD GPUs are based on the Graphics Core Next (GCN)™ architecture [95]. Figure 2.1a shows the basic structure of a GPU with compute units. The GPU consists of many compute units (CUs) as shown in Figure 2.1b, which are the hardware blocks for performing multi-threaded computations. The block-based structure of an AMD GCN CU is shown in Figure 2.2. A kernel executing on a GPU can launch a large number of threads which are scheduled for execution on CUs. The threads are executed on a CU in groups of 64 threads, called wavefronts. The front-end of the CU consists of 4 wavefront pools and a scheduler unit. The scheduler fetches the instructions from the oldest wavefront in the wavefront pool. The instructions are decoded and are assigned to different functional units. The CU consists of 4 SIMD units, where each unit consists of 16 SIMD lanes (Compute cores) which perform vector computation for 16 different threads. Each lane of the SIMD unit performs the same instruction for all the threads in each cycle. Every lane consists of an ALU and a floating point unit. Each SIMD unit is equipped with a 256KB vector register file. The other important functional unit is the branch unit, which handles the control-flow instructions. The local-data share (LDS) unit manages the instructions dealing with local memory. A scalar unit is a special unit to manage the scalar arithmetic and scalar memory operations. Each CU is equipped

Figure 2.1: Basic GPU architecture (a) Compute device (b) Compute unit (c) SIMD lane.
with 16KB of L1 data cache. The GPU consists of two types of memory:

- **Local Memory**: On-chip memory associated with each compute unit on the device. Wavefronts from separate CUs cannot share the local memory. Local memory accesses exhibit low access latency. 64 KB local memory is available for each CU on an AMD GCN GPU.

- **Global DRAM memory**: The off-chip DRAM memory on the GPU is called *global memory*. It can be accessed by all of the CUs on the device. Global memory is large in size (2GB, 4GB, 12GB, etc.). The major use of global memory is to hold the input and output data accessed by the kernel. The global memory exhibits high access latency due to off-chip access. GPUs use a GDDR-5 interconnect for accessing global memory.

The GCN AMD Radeon 7970 GPU includes 32 CUs, while the R9-290X GPU includes 44 CUs.

### 2.1.2 NVIDIA Kepler Architecture

The latest family of NVIDIA GPUs is the Kepler architecture. The primary unit for computation on a NVIDIA GPU is a Stream MultiProcessor (SMX). The SMX on the Kepler GK110 architecture is shown in Figure 2.3 Each of the Kepler GK110 SMX units features 192 single-precision CUDA cores, and each core has fully pipelined floating point and integer ALUs. Threads are executed on a SMX in groups of 32 called *warps*. The warps are scheduled to execute on the
CHAPTER 2. BACKGROUND

Figure 2.3: Advanced Stream Multiprocessor (SMX) block diagram for Kepler architecture by NVIDIA [108].

CUDA cores using the warp scheduler present on the SMX. The Kepler SMX uses a quad warp scheduling mechanism which issues and executes 4 warps concurrently. The quad warp scheduler can dispatch two independent instruction per warp using the 8 dispatch modules present on the SMX. Each SMX consists of 256KB vector register file, and each each thread can access upto 255 registers. The Kepler GPU provides 65KB of on-chip memory for each SMX. This memory can be partitioned as shared memory (local memory) and L1 data cache. The Kepler GPU permits three shared-memory/L1-cache partitions: (1) 48KB/16KB, (2) 16KB/48KB and (3) 32KB/32KB. The total off-chip memory of the Kepler series GPUs varies from 2GB to 12GB and is accessed using GDDR-5 interconnect. The NVIDIA GTX 680, K20m and K40m GPUs consist of 8, 15 and 16
CHAPTER 2. BACKGROUND

SMX units, respectively.

2.1.3 Evolution of GPU Architecture

<table>
<thead>
<tr>
<th>NVIDIA Devices</th>
<th>NVIDIA C2070</th>
<th>NVIDIA K40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microarchitecture</td>
<td>Fermi</td>
<td>Kepler</td>
</tr>
<tr>
<td>Fabrication</td>
<td>40nm</td>
<td>28nm</td>
</tr>
<tr>
<td>Compute Capability</td>
<td>2.0</td>
<td>3.5</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>448</td>
<td>2880</td>
</tr>
<tr>
<td>Core Frequency</td>
<td>575 MHz</td>
<td>745 MHz</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>144 GB/s</td>
<td>288GB/s</td>
</tr>
<tr>
<td>Peak Single Precision FLOPS</td>
<td>1288 GFlops</td>
<td>4290 GFlops</td>
</tr>
<tr>
<td>Peak Double Precision FLOPS</td>
<td>515.2 GFlops</td>
<td>1430 GFlops</td>
</tr>
<tr>
<td>Thermal Design Power (TDP)</td>
<td>247 W</td>
<td>235 W</td>
</tr>
</tbody>
</table>

Table 2.1: Evolution of NVIDIA GPU architecture from Fermi to Kepler.

<table>
<thead>
<tr>
<th>AMD Devices</th>
<th>Radeon HD 5870</th>
<th>Radeon HD 7970</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microarchitecture</td>
<td>Evergreen</td>
<td>Southern Islands</td>
</tr>
<tr>
<td>Fabrication</td>
<td>40nm</td>
<td>28nm</td>
</tr>
<tr>
<td>Stream Cores</td>
<td>320</td>
<td>2048</td>
</tr>
<tr>
<td>Compute Units</td>
<td>20</td>
<td>32</td>
</tr>
<tr>
<td>Core Frequency</td>
<td>850 MHz</td>
<td>925 MHz</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>153.6 GB/s</td>
<td>264 GB/s</td>
</tr>
<tr>
<td>Peak Single Precision FLOPS</td>
<td>2720 GFlops</td>
<td>3789 GFlops</td>
</tr>
<tr>
<td>Peak Double Precision FLOPS</td>
<td>544 GFlops</td>
<td>947 GFlops</td>
</tr>
<tr>
<td>Thermal Design Power (TDP)</td>
<td>228 W</td>
<td>250 W</td>
</tr>
</tbody>
</table>

Table 2.2: Evolution of AMD GPU architecture from Evergreen to GCN.

GPU architectures have evolved over the past few years to provide greater compute throughput with lower power consumption. GPUs have increased in performance, architectural complexity and programmability. This trend can be observed in various generations of GPUs from different vendors [107]. Table 2.1 shows how NVIDIA GPUs have improved from the Fermi to Kepler microarchitecture in terms of their compute capability, architectural complexity, and memory bandwidth. Computing performance has increased four fold, while power consumption (Thermal Design Power - TDP) of the GPU remains comparable. Advances in fabrication technology and power saving strategies have allowed GPUs to achieve superior power-performance (GFlop/Watt) ratings. The advancements in fabrication are also supported by architectural refinements to enhance...
CHAPTER 2. BACKGROUND

the compute capability of the GPU. Newer hardware modules have been added to the GPU architecture to enhance programmability using CUDA and OpenCL.

Table 2.2 shows similar improvements that have been made in AMD’s GCN architecture for the Southern Islands family of GPUs, over its predecessor. The AMD GCN architecture added more compute units, with a very small increase in power consumption over its predecessor. AMD also deprecated the prior Very Large Instruction Word (VLIW) based ISA used in their older GPUs, when they moved to the GCN architecture [95]. Modern AMD GPUs have also added hardware support for improving programmability using OpenCL. We highlight some of the hardware units present in modern GPUs, developed for improving execution efficiency of GPU workloads.

- **Advanced Hardware Queues**: A hardware queue is used to map a workload (kernel) to different SMXs/CUs. Traditionally, GPUs used one hardware queue which managed all the kernels issued to it. This reduced the degree of concurrency on the GPU by imposing a serial execution order for data-independent kernels. The latest GPUs from NVIDIA and AMD added multiple hardware queues to manage mapping of multiple kernels to SMXs/CUs. The Hyper-Q feature was introduced by NVIDIA on their Kepler GK110 architecture devices [109]. Multiple kernels get mapped to different hardware queues, which can schedule the execution of kernels on the device concurrently. Hyper-Q permits up to 32 simultaneous, hardware-managed, concurrent executions, if the kernels have no data dependency and the GPU has enough compute resources to support such execution. The multiple streams of the applications are handled by separate hardware queues and data-independent kernels can be executed concurrently. AMD introduced Asynchronous Compute Engines (ACE) on their GPUs as a hardware queue to schedule workloads on different compute units [95]. Work from different OpenCL command queues is mapped to different ACE units on the AMD GCN hardware. The ACE units support interleaved execution of compute kernels on the GPU. The latest GPUs in the GCN family include 8 ACE units to manage multi-kernel execution.

- **Scalar Unit**: The scalar unit is a specialized functional unit added to the AMD GCN architecture. The ISA of the GCN architecture supports special instructions for the scalar unit. The major function of this unit is to prevent wastage of SIMD resources (vector registers) for scalar computations. A computation on a GPU which uses the same data and produces the same results across all threads is called a scalar. The best case of a scalar operation is the management of loop variables, which are common across all threads and do not require vector processing using SIMD execution. The scalar unit provides minimum wastage of vector registers.
CHAPTER 2. BACKGROUND

- **Nested Parallelism**: Nested parallelism is defined as the process of launching child threads from a parent thread. A child kernel performing the same or different computation can be launched using a thread from its parent kernel. *Dynamic Parallelism* is an extension of the CUDA and OpenCL programming models \[3\ [100]\. It provides the user with constructs to implement nested thread-level parallelism. The ability to create and control workloads directly from the GPU avoids the need to transfer execution control and data between the host and the device. This reduces the overhead of invoking a GPU kernel from the host CPU. Dynamic Parallelism also offers applications the flexibility to adapt thread assignments to cores during runtime. Nested Parallelism enables an application to establish the runtime pattern dynamically through the parent threads executing on the device. Additionally, since child threads can be spawned by a kernel, the GPU’s hardware scheduler and load balancer are utilized dynamically to support data-driven applications.

### 2.2 The OpenCL Programming Model

The emerging software framework for programming heterogeneous devices is the Open Computing Language (OpenCL) \[101]\. OpenCL is an industry standard maintained by Khronos, a non-profit technology consortium. The support for OpenCL has been increasing from major vendors such as Apple, AMD, ARM, MediaTek, NVIDIA, Intel, Imagination, Qualcomm and S3.

OpenCL terminology refers to a GPU as the *device*, and a CPU as the *host*. These terms are used in the same manner for the remainder of this work. Next, we summarize the OpenCL platform and memory models.

#### 2.2.1 Platform Model

The platform model for OpenCL consists of a host connected to one or more OpenCL devices. An OpenCL device is divided into one or more Compute Units (CUs) which are further divided into one or more Processing Elements (PEs). Computations on a device occur within the processing elements. The architecture is shown in Figure 2.4. An OpenCL application runs on a host according to the models native to the host platform. The OpenCL application submits commands from the host to execute computations on the processing elements within a device. The processing elements within a compute unit execute a single stream of instructions as SIMD units (execute in lockstep with a single stream of instructions).
CHAPTER 2. BACKGROUND

Figure 2.4: OpenCL Host Device Architecture

The code that executes on a compute device is called a kernel. OpenCL programs execute on a GPU in the form of kernel(s) that execute on one or more OpenCL devices and a host program that executes on the CPU. The host program defines the context for the kernels and manages their execution. The kernel defines an index space that is unique to each application. An instance of the kernel executes for each point in this index space. This kernel instance is called a work-item and is identified by its point in the index space, by a global ID for the work-item. Each work-item executes the same code, but the specific execution pathway through the code and the data operated upon can vary per work-item based on the work-group ID.

Work-items are organized into work-groups. Work-groups are assigned a unique work-group ID with the same dimension as the index space used for the work-items. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit. The index space is called an \textit{NDRange} which is an \textit{N-dimensional} index space, where \( N \) is equal to one, two or three.

2.2.2 Memory Model

Work-item(s) executing a kernel have access to four distinct memory regions. This is clearly illustrated in Figure 2.5.
CHAPTER 2. BACKGROUND

Figure 2.5: The OpenCL memory model.

- **Global Memory**: This memory region permits read/write access to all work-items in all work-groups. Work-items can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device.

- **Constant Memory**: This memory is a region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.

- **Local Memory**: This is a region of memory that is local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group. It may be implemented as dedicated regions of memory on the OpenCL device. Alternatively, the local memory region may be mapped onto sections of the global memory.

- **Private Memory**: This region of memory is private to a work-item. Variables defined in one work-items private memory are not visible to another work-item.

2.2.3 Runtime and Driver Model

Runtime environments for GPU programming frameworks handle the creation, management and deletion of all the program objects and memory objects required by the application. The host application uses a rich API interface to communicate with the GPU. The required kernel objects and memory buffers are created using an API. The transfer of data to the GPU from the host
CHAPTER 2. BACKGROUND

side and also the launch of kernel on the GPU is managed using the runtime API. The OpenCL framework consists of a runtime environment which supports all the API described in the OpenCL standard [101, 100]. While the OpenCL standard is maintained by Khronos, the runtime environment is separately provided by each GPU vendor. The runtime layer interprets the API and executes them on the GPU using a device driver for the GPU. Each vendor with a OpenCL conformant GPU provides a runtime environment which follows the specifications and behaviors of each API, as described by the standard.

![Figure 2.6: (a) CML position in OpenCL runtime stack and (b) context dispatch using CML.](image)

The OpenCL runtime environment can be visualized as a two-level structure, as shown in Figure 5.2a. The upper-level runtime consists of the OpenCL API library and data structures to manage the memory and program objects for the GPU computation. The upper-level runtime creates the user-mode objects for all the memory and program objects. A memory buffer created using the OpenCL API is first registered with the upper-level runtime. It creates a place-holder for the buffer and associates it with a respective pointer on the host side. The upper-level runtime is also responsible for creating kernel objects from OpenCL source code. The OpenCL compiler is invoked by the upper-level runtime to compile the OpenCL source code to a machine readable binary format. The binary is also maintained by the runtime for the entire duration of the application. This avoids the need to re-compile each kernel for multiple launches. The data transfer and kernel launch API require specific action from the host to the device. Such commands are passed to the GPU using a software abstraction called the command queue. The upper-level runtime monitors each command queue and communicates with the GPU using a lower-level runtime interface.
CHAPTER 2. BACKGROUND

The lower-level runtime consists of the OS-level interface and the Application Binary Interface (ABI) for handling the execution of the OpenCL program. The commands for the GPU are communicated by the lower-level runtime using the device driver \[127\]. The lower-level runtime uses the OS system call interface similar to the Intel TBB model, to communicate with the device driver of the GPU \[123\]. The device driver is invoked in kernel-mode (OS controlled), and manages all communication to the GPU using the PCIe interconnect. The Figure \[5.2b\] illustrates the implementation of an OpenCL kernel execution API using all the layers of the OpenCL runtime.

2.3 Multi2Sim: Cycle-level GPU Simulator

We introduce the Multi2Sim framework which supports cycle-level simulation of x86 CPUs, ARM CPUs and AMD GCN GPUs \[148\]. We use Multi2Sim for the implementation of our hardware and runtime-based support for multi-level concurrency, described in this thesis proposal.

Multi2Sim uses a three-stage approach for implementing architectural simulation of the CPU-GPU based heterogeneous system, shown in Figure \[2.7\]. The architectural simulation includes a detailed timing analysis of the microarchitecture and memory hierarchy of the selected CPU and GPU. Multi2Sim provides software layers defined as “runtime” and “driver”, to execute OpenCL applications on different simulated GPUs. We focus on the simulation of the x86 CPU as the host, and an AMD GCN Southern Islands GPU as device, to execute OpenCL applications. The three stages of CPU-GPU based simulation include:

1. Disassembler
2. Functional Simulator
3. Architectural Simulator

2.3.1 Disassembler

Given a bit stream representing machine instructions for a specific instruction set architecture (ISA), the disassembler decodes these instructions into an alternative representation that allows a straightforward interpretation of the instruction fields, such as operation code, input/output operands, and immediate constants.

Multi2Sim’s disassembler for a given microprocessor architecture can operate autonomously or serve later simulation stages. The disassembler reads directly from a program binary generated by a compiler, such as an x86 application binary, and outputs machine instructions one-by-one in the form of organized data structures that split each instruction into its comprising fields.
In the case of GPU simulation, the disassembler for the AMD GPU decodes the instructions present in the binary generated after the compilation of the OpenCL source code.

2.3.2 Functional Simulation / Emulation

The purpose of the functional simulator, also called the emulator, is to reproduce the original behavior of a guest program, providing the illusion that it is running natively on a given microarchitecture. To accomplish this effect, Multi2Sim keeps track of the guest program state, and dynamically updates it instruction-by-instruction until the program finishes. The state of a program is expressed as its virtual memory image and the architected register file. The virtual memory image consists of the set of values stored at each possible memory location addressable by the program. The state of the architected register file is formed of the values for each register defined in a specific architecture (e.g., x86, GCN). For a program state associated with a specific point in its execution, the emulator updates it to the next state after consuming one single ISA instruction. This process is done in 4 steps: 

1. The new instruction is read from the memory image containing the program’s code at the location pointed to by the instruction pointer architected register.
2. The instruction is decoded, taking advantage of the interface provided by the disassembler module.
3. The instruction is emulated, updating the memory image and architected registers according to the instruction’s opcode and input/output operands.
4. The instruction pointer is updated to point to the next instruction to be emulated. A simulation loop keeps emulating instructions repeatedly until the program runs its termination routine. The functional simulator also provides an interface for the timing simulation phase. In this phase, the timing simulator requests that it emulate the next available instruction.
CHAPTER 2. BACKGROUND

The emulation of an x86 OpenCL host program sets up the Southern Islands emulator as a result of the OpenCL API calls intercepted by the OpenCL runtime, such as `clGetDeviceIDs`, `clCreateBuffer`, or `clEnqueueWriteBuffer`. The call that ultimately launches the Southern Islands emulator is `clEnqueueNDRangeKernel`, which initializes the ND-Range, work-groups, wavefronts, and work-items, and transfers control to the GPU. During emulation, Multi2Sim enters a loop that runs one instruction for each wavefront in all work-groups of the NDRange at the same time, until the whole ND-Range completes execution.

2.3.3 Architectural Simulator

The architectural simulator is also referred to as the timing simulator for Multi2Sim. The timing simulator provides a cycle-level simulation of an application by modeling each hardware stage of the processor pipeline with variable latencies. Multi2Sim supports timing simulation for multi-core x86 CPUs with superscalar pipeline structures. The modeled hardware includes pipeline stages, pipe registers, instruction queues, functional units, and cache memories. The timing simulator is structured as a main loop, calling all pipeline stages in each iteration. One iteration of the loop models one clock cycle on the real hardware. While hardware structures are modeled in the timing simulator, the flow of instructions that utilize them is obtained from invocations to the functional simulator. The timing simulator requests emulation of a new instruction by the functional simulator, after which the latter returns all information about the emulated instruction, as propagated by its internal call to the disassembler. While the instruction travels across pipeline stages, it accesses different models of hardware resources such as functional units, effective address calculators, and data caches with potentially diverse latencies.
CHAPTER 2. BACKGROUND

The timing simulator for the GPU model also follows the same methodology but for multiple compute units. The GPU timing model is based on the design of the GCN compute unit shown in Figure 2.2. Each compute unit in the GPU is replicated with an identical design. The compute unit front-end fetches instructions from instruction memory for different wavefronts, and sends them to the appropriate execution unit. The execution units present in a compute unit are the scalar unit, the vector memory unit, the branch unit, the LDS unit (Local Data Store), and a set of Single-Instruction Multiple-Data (SIMD) units. The LDS unit interacts with local memory to service its instructions, while the scalar and vector memory units can access global memory, shared by all compute units. The pipeline stages of each functional unit are modeled and are replicated for each compute unit. The modeled pipeline structure for the SIMD unit is shown in Figure 2.8.

2.3.4 OpenCL Runtime and Driver Stack for Multi2Sim

Execution of a OpenCL application using the CPU and the GPU simulator requires a full functional OpenCL runtime. Multi2Sim provides a software stack which models the OpenCL runtime and driver. This software stack creates an interface between the CPU simulator and the GPU simulator, and allows the user to realize a complete heterogeneous system simulation. The four major components of OpenCL runtime and driver stack include:

- A runtime library is a software module running in the user space, implementing the OpenCL interface. Functions using this interface are invoked by a user program, with the likely ultimate purpose of accessing system hardware. Runtime libraries on a native framework (sometimes referred to as user-mode drivers (UMD)) can be either generic or provided by the hardware vendor. Runtime libraries on Multi2Sim run as emulated code which is a part of the CPU simulation.

- A device driver is a software module running in privileged mode. When the runtime library requires access to the hardware, it invokes the device driver through system calls, such as ioctl, or open/read/write calls using special file descriptors. On a native environment, the code serving these system calls is a kernel module that has direct access to the hardware.

- The Application Programming Interface (API) is the interface between a runtime library and the user program. The API is exactly the same for the OpenCL runtime as is used in the Multi2Sim runtime, which is why they can be used interchangeably to satisfy external references in the user program.
CHAPTER 2. BACKGROUND

- The Application Binary Interface (ABI) is the interface provided between a device driver and the runtime library, and is based on a system call interface. In Multi2Sim, each ABI between a runtime-driver pair is assigned a unique system call code unused in Linux. The part of the ABI which executes in the kernel mode and invokes the device driver is also referred to as the kernel-mode driver (KMD).

The Figure 2.9 shows the interaction of the host-code with the simulated runtime and driver stack provided by Multi2Sim.

2.4 Concurrent Kernel Execution on GPUs

The large number of compute resources available on the GPU lead to a problem of efficient utilization of these resources. Many real-world applications possess multiple kernels for computation \([17, 32, 98, 113, 134]\). Many of these kernels do not fully utilize the available re-
CHAPTER 2. BACKGROUND

Allowing data independent kernels to execute concurrently on the GPU can help utilize the resources, as well as increase the kernel throughput for the application.

As mentioned in Section 1.2, the Madness (Multiresolution Adaptive Numerical Environment for Scientific Simulation) framework is a commonly used computational chemistry code that is run on GPUs. It uses the GPU to compute multi-dimensional integrals to simulate an adaptive multiresolution grid of molecules [134]. The integral calculation in the application is implemented as repeated launches of small-sized matrix multiplication kernels. The integrals operate on a small portion of the data set. This leads to a launch of a large number of data independent kernels on the GPU. The small size used in the kernels do not completely utilize the available resources on the GPU, which impacts kernel throughput. Similarly, the application Biometric Optical Pattern Matching has been migrated to run on a GPU [17] using multiple kernels of different sizes. The launch order of these kernels is non-deterministic, and is dependent on the data set and the resolution of computation requested by the user. Concurrent kernel execution allows for such multi-sized kernels to execute simultaneously. This can avoid under-utilization of resources or starvation of certain kernels due to the launch order. In recent years, researchers have also augmented Linux system services such as AES encryption and garbage collection, employing the services of a GPU. The computations present in these services can be launched on the GPU using multiple kernels of various sizes [93, 141]. These applications can potentially benefit from the concurrent kernel execution, where independent services are processed simultaneously. An advanced resource partitioning and scheduling mechanism required to facilitate such concurrency is described in Chapter 4.

clSURF is an OpenCL and OpenCV based framework for image processing on GPUs using the SURF (Speeded-Up Robust Features) algorithm [21, 29, 98]. This application uses multiple
kernels to provide a pipelined-based computation of an image or video frame. Each frame passes through 5 stages of computation using 7 kernels. These kernels are launched multiple times in different stages. The basic computation stages of the *clSURF* application are shown in Figure 2.10. The kernels used by each stage can be launched concurrently on the GPU. This can improve the kernel throughput, as well as provide an efficient pipelined operation for streaming data (video processing).

### 2.4.1 Support for Concurrent Kernel Execution on GPUs

Concurrent execution of kernels on a single GPU was first supported by Nvidia’s Fermi GPUs [51]. A scheduling policy similar to a left-over policy, which schedules a complete kernel at a time [113], is supported on Fermi GPUs. The left-over policy only schedules the next kernel if the required number of compute units are available. The Fermi GPUs used a single hardware work queue for scheduling kernels on the device. This commonly leads to false dependencies being introduced between kernels, and does not allow for concurrent execution of data independent kernels. Figure 2.11 illustrates a scenario of three kernels scheduled for execution on the GPU. K\_A executes on the GPU, while K\_B waits since it is dependent on K\_A data. K\_C is an independent kernel, which can be scheduled for execution, but is not due the single hardware queue. Thus, K\_C encounters a false dependency on K\_B. The single hardware queue policy inhibits the concurrent execution of kernels, and does not experience good kernel throughput.

The current state-of-the-art GPUs from AMD and NVIDIA have resolved this issue by implementing multiple hardware queues on the GPU (discussed in Section 2.1.3). Nvidia’s Kepler GPUs achieve concurrent execution of kernels using Hyper-Q technology [109], while AMD GPUs support multiple hardware queues using Asynchronous Command Engine (ACE) units defined in the GCN architecture [95]. These multiple hardware queues can manage the scheduling of more than one kernel simultaneously. Figure 2.12 illustrates the scheduling effects of multiple hardware queues. The scheduling mechanism associated with the hardware queues allows the applications to use the free/unused compute units, even if it does not completely satisfy the resource requirements of the kernel. The scheduler assigns more resources to the kernel as they become available. The
use of hardware queues improves the overall resource utilization of the device and avoids false dependencies between kernels.

2.5 Concurrent Context Execution on GPUs

The concurrency supported on current GPUs is limited to kernels belonging to the same application context on the GPU. An application context on a GPU defines the virtual memory space created for the execution of one or more kernels. A context is typically created by an application executing on the CPU, as defined by the CUDA and OpenCL programming frameworks. Many applications include only a single kernel or a few small-sized kernels that do not fully utilize the massive compute resources on the GPU. For GPUs used in enterprise level computing, wastage of compute resources reduces the power-performance efficiency of the cluster.

2.5.1 Context Concurrency for Cloud Engines and Data Centers

GPUs are used as accelerators on the computing nodes of many leading supercomputers in the Top-500 list [147]. Cloud computing services provide users with flexible options of choosing computing hardware. Cloud computing service providers such as Amazon Web Services\(^1\), and Nimbix\(^2\) have recently started to offer GPUs as a part of their cloud engine. Such infrastructure allows users to adopt the pay-as-per-use model, which enables small-scale users to execute their applications on large clusters without the need to own or maintain them.

Workloads from multiple users are submitted to the cloud engine to be executed on the GPU. MapReduce applications for big-data analytics [139], NAMD for parallel molecular dynamics

---

\(^1\)http://aws.amazon.com/
\(^2\)http://www.nimbix.net/
CHAPTER 2. BACKGROUND

simulation [116], the *Madness* framework for multi-dimensional integral solvers [134], WRF for numerical weather prediction [97], the Caffe framework for neural network analysis [67], are a few examples of applications that have been adapted to be executed on a cloud engine possessing a GPU. While the GPU can provide benefits to these applications, many applications are not able to fully utilize the GPU. If only these applications could physically *share* the GPU resources with other applications, we could then see a significant increase in application throughput. This would also improve the utilization of the GPUs in a cluster environment. Architectural and software support is required to facilitate concurrent execution of multiple application contexts.

2.5.2 Key Challenges for Multi-Context Execution

Support for multi-context execution on GPUs requires us to consider some major architectural and runtime level challenges.

![Figure 2.13: Average active time of data buffers in applications.](image)

**Global memory management:** Modern GPU applications use multiple kernels and numerous data buffers for completing their computation. These buffers occupy space on the global memory of the GPU, but not all buffers are frequently used in each kernel. The frequently used data buffers are referred to as *persistent* buffers. The data buffer is *active* for the duration of the execution of the kernel that is accessing the buffer. Figure 2.13 reports the average *active* time of data buffers in multi-kernel GPU applications from the Rodinia benchmark suite [32]. We observe that the majority of data buffers are inactive for 48% (avg.) of an application’s execution time. The space occupied in GPU memory by such buffers can inhibit the launch of multiple contexts due to memory limitations. A sophisticated management of global memory is required to prevent memory consumption by inactive data buffers.

**GPU task control:** Multi-context execution demands constant interaction between the runtime and the GPU. The global memory management mechanism needs to frequently transfer
data buffers between the CPU and the GPU. The status of buffer transfers can be collected by querying the GPU. The runtime layer also queries the GPU for the amount of free resources and the state of applications executing on the GPU. Given the number of queries required and the limited speed of the PCIe bus, timely context management would be hampered and PCIe traffic would be substantially increased. This issue can be addressed by enhancing the controller on the GPU, which can monitor the status of global memory and initiate decisions relative to buffer transfers. The controller can also update the runtime periodically, reporting the state of the GPU. This facilitates an efficient CPU-GPU interaction for context management. The multiple contexts executing on the GPU require frequent and timely dynamic task scheduling and resource allocation decisions.
Chapter 3

Related Work

In this chapter, we review related work in the areas of architecture and runtime modeling, scheduling schemes, spatial partitioning, and support for concurrent context execution on GPUs.

3.1 GPU Modeling and Simulation

Researchers have frequently used simulators for designing new architectural features, or studying the behavior of applications while considering microarchitectural tradeoffs. Functional simulators are generally used for verifying the correctness of designs and for first-order evaluation. Popular functional simulators such as MARS [154], Simics [94], M5 [28], and QEMU [24, 25] are used for emulating instructions from various CPU ISA (ARM, MIPS, SPARC, x86, etc.). Timing simulators such as SimpleScalar [31], Gem5 [27], PTLsim [160], Multi2Sim [148, 149], and MARSS [114] are used to understand the performance of the CPU microarchitecture, memory hierarchy, and interconnect networks. Timing simulators realize a cycle-level simulation of the each microarchitectural unit. Power, energy and area estimation are required to judge the feasibility of the proposed architectural design. The simulators such as McPAT [85] and Watch [30] provide detailed analysis of the power consumption and area requirements of the proposed CPU design. Many CPU simulators provide an interface to these power and area estimation tools.

Simulation tools have been developed to analyze the performance of the architecture of a GPU device. GPU simulators have supported execution of GPU applications developed using CUDA and OpenCL. Similar to the CPU simulators, the GPU simulators are also classified in three categories: (1) functional simulators (i.e., emulators), (2) cycle-level architectural simulators and (3) power and area simulators.
CHAPTER 3. RELATED WORK

3.1.1 Functional simulation of GPUs for compute and graphics applications

A functional simulation for the OpenGL graphics pipeline and D3D pipeline was implemented in the Attila simulator [37]. The Attila simulator consists of a functional simulator for render-based GPUs, such as the AMD R580 and the NVIDIA G80. The simulator provides a driver layer to trap OpenGL calls and obtain render traces from OpenGL. The functional emulation of these traces can be done on the Attila framework. The GPU simulation framework of Attila includes a timing model to estimate the performance of the simulated GPU configuration when rendering a frame. The timing simulator is based on understanding of GPU hardware from a graphics perspective, and is not validated with any physical GPU. The Attila emulation framework supports both unified and non-unified shader architecture models.

Functional simulators for GPUs are able to emulate a GPU kernel and operation of each thread. They support emulation of the ISA belonging to different GPU vendors. The Ocelot functional simulator supports emulation of CUDA applications and provides an implementation of the CUDA Runtime API [25]. Ocelot supports emulation of NVIDIA PTX (Parallel Thread Execution) virtual ISA, which is used as a device-agnostic program representation that captures the data-parallel SIMT execution model of CUDA applications. Ocelot also supports translation of PTX to other backend targets such as x86 CPUs and AMD GPUs. The Ocelot tool has been used in various studies on GPUs, including software reliability [86], instrumentation and workload characterization [47], and binary translation for cross-vendor execution of CUDA applications [42].

The Barra simulator was designed specifically for emulation of CUDA applications at the assembly language level (Tesla ISA) [35]. Its ultimate goal is to provide a 100% bit-accurate simulation, offering bug-for-bug compatibility with NVIDIA G80-based GPUs. It works directly with CUDA executables; neither source modification nor recompilation is required. Barra can be used as a tool for research on GPU architectures, and can also be used to debug, profile and optimize CUDA programs at the lowest level.

The functional simulators provide a limited amount of detail regarding the architectural behavior of the GPU. The major use of such functional models is for first-order analysis and to debug applications.

3.1.2 Architectural Simulation of GPUs

Architectural simulators model the microarchitectural features of the GPU. They provide a detailed cycle-level analysis of the kernel execution on the selected GPU configuration. We use the Multi2Sim simulation framework which provides detailed cycle-level CPU and GPU models for
executing OpenCL applications. The features of Multi2Sim are described in Section 2.3. In this section we discuss other popular frameworks used for cycle-level GPU simulation.

**GPGPU**_sim_ is a simulation framework for executing CUDA applications and OpenCL applications on cycle-level models of the NVIDIA GPUs belonging to the Fermi class [18, 51]. The simulator provides detailed models of the (Single-Instruction-Multiple-Thread (SIMT) engines and Streaming Multi-Processors (SMs) of NVIDIA GPUs. The SIMT cores contain separate ALU and memory unit pipelines. The simulator also models a workgroup and a warp (wavefront) scheduler. The main feature of the simulator is the detailed modeling of the interconnect networks, cache memories and the DRAM memory. Along with the cycle-level analysis of the kernel performance, the simulator also provides statistics for each module such as caches, SIMT units, operand collectors, interconnect networks and DRAM.

The architectural configuration of the GPU can be modified using configuration files in the simulator. The framework also allows the user to describe fine-grained details such as ld/st buffer lengths, SIMT fetch width, and latencies for each cache level and interconnect network. GPGPUsim provides an interface to Booksim, which is a network simulator for advanced in-depth simulation of interconnect networks and topologies [68]. The tool reports a 98.3% correlation with the NVIDIA GT200 GPU and 97.3% correlation with Fermi hardware for the applications from Rodinia benchmark suite [32].

GPGPUsim does not provide a detailed model for the host CPU of the heterogeneous system. It also does not model the runtime for CUDA/OpenCL and does not include the overheads due to the runtime and communication with the host. Our mechanisms for multi-level concurrency presented in this thesis proposal depend on the GPU architecture, as well as the runtime environment of OpenCL. GPGPUsim is not a good fit for our proposed research.

**Gem5-GPU:** The gem5-gpu simulation framework is an amalgamation of the gem5 CPU simulator and the GPGPUsim GPU simulator [119]. Gem5 provides cycle-level models of x86, Alpha, SPARC and ARM CPUs. It provides detailed models for the processor pipeline stages, cache hierarchy and DRAM memory. The gem5-gpu tool uses a specialized interface to glue the CPU and GPU simulations together. It provides host code simulation using the gem5 model, while the CUDA kernel is simulated using the GPU model of GPGPUsim. Gem5-gpu routes most memory accesses through Ruby, which is a highly configurable memory system in gem5. By doing this, it is able to simulate many system configurations, ranging from a system with coherent caches and a single virtual address space across the CPU and GPU, to a system that maintains separate GPU and CPU physical address spaces. The simulation model traps and emulates the CUDA/OpenCL calls originating from the host code in the gem5 model. The GPU simulator is initiated when a kernel
CHAPTER 3. RELATED WORK

starts execution. The simulation framework provides details about the host code execution but does not simulate the OpenCL/CUDA runtime. Hence, the effects of the runtime environment cannot be captured using the framework. Our proposed mechanism heavily relies on accurate simulation of the OpenCL runtime, and so our work cannot be evaluated using the gem5-gpu simulator.

3.1.3 Power and Area Estimation on GPUs

Prior research suggested modifications to the microarchitecture of the GPU which requires analysis of power consumption and area overheads. The GPUWattch framework is a detailed power simulator for GPUs [84]. The model uses a bottom-up methodology which abstracts parameters from the microarchitectural components such as model inputs. It also ensures that both program and microarchitectural level interactions are captured during execution, thereby enabling new power-management techniques specifically targeted at GPUs. Stability of the model is examined by validating the power model against measurements of two commercial GPUs (GTX 480 and Quadro FX 5600). The power model modifies and extends the McPAT CPU power model simulator [85] to model the power of contemporary GPGPUs and drive the modified McPAT version with GPGPUSim [18], a cycle-level GPU simulator. The area estimation of a microarchitectural feature is also provided by the modified version of the McPAT tool.

Statistical and empirical power models have also been developed for GPUs from AMD and NVIDIA [70, 92, 102, 162]. Work by Diop et al. provides an empirical power model for AMD GPUs using Multi2Sim simulation framework [40]. Their work captures the activity factors of different modules of the GPU using Multi2Sim statistics and provides a power estimate using a linear regression based model.

3.2 Design of Runtime Environments for GPUs

One of the goals of a runtime system is to simplify application development and provide support for task scheduling. OpenCL due to the architectural abstraction it provides is commonly referred to as a virtual device driver [46]. Sun et al. demonstrate software based support for task scheduling across CPU and GPU devices [140].

Synchronization requirements and disparity in computational capacity when targeting multiple OpenCL compute devices makes writing scalable code challenging. Runtimes targeting multiple heterogeneous systems have been developed [14, 135] which are similar to equivalent CPU based frameworks such as Intel’s TBB [124]. Profiling and specialization extensions discussed in this thesis are complimentary to such frameworks.
CHAPTER 3. RELATED WORK

GPU architectures presently are commonly accessed via calls to a driver. However the driver software stack imposes an overhead on kernel launch on modern GPUs. Driver overhead is caused by events such as switching to and from kernel and user space [128]. This software architecture prevents the GPU from being utilized for lightweight tasks and operating system services such as encryption [128]. Runtimes such as Bothnia allow access to compute devices such as GPUs without the overhead of the driver stack [34]. Other runtimes which provide the application developer with abstractions such as a dataflow programming model managed by an operating system has been shown to simplify data management between the kernel and user space [128].

One of the common goals of runtimes targeting heterogeneous systems is to perform different data transformations on user data in order to simplify programming. The goal of the data transformations is usually to improve data locality for memory performance or to manage host-to-device IO. Related work in runtime data transformations on GPUs is usually specific to molecular dynamics [158]. However, recent novel approaches in data transformations and application runtimes have begun to utilize the CPU as a symbiotic resource to facilitate the execution of GPGPU programs on fused CPU-GPU architectures [159].

3.3 Resource Management for Concurrent Execution

Multi-level concurrency requires efficient management of compute and memory resources on the GPU. In this section, we discuss prior work relevant to the resource management mechanisms described in this thesis proposal.

3.3.1 Spatial Partitioning of Compute Resources

Multi-level concurrency requires a fair mechanism be used to allocate compute units to the executing kernels/contexts. Adriaens et al. proposed spatial partitioning of compute resources across concurrent applications using heuristics such as best obtained speedup, best profiling results, and equal partitioning of compute units [10]. They developed their heuristics by simulating several kernels from different applications running in parallel. Their work focuses on evaluating the benefits of spatial multitasking, rather than proposing the mechanisms to implement it. Tanasic et al. introduce flexible spatial partitioning of GPU resources to support preemption on GPUs [143]. Their partitioning mechanism requires explicit assignment of compute units (CUs) to kernels, which is then modified in presence of additional workloads. Our approach described in Chapter 4 provides the choice of using explicit assignment of CUs, or runtime controlled assignment with superior load-balancing.
Several software techniques have been proposed in the past to establish kernel-level and context-level concurrency. Wang et al. proposed an approach of context funneling to support multi-application execution on the GPUs [155]. Their approach provides interleaved execution of the applications by using the same shared context. They propose the use of multiple CUDA streams (or OpenCL command queues) to launch different kernels belonging to different applications. Real-time and adaptive applications have used multiple OpenCL command queues to establish concurrency across multiple kernels and extract performance benefits from the device [78, 153]. Kayiran et al. propose an integrated concurrency management strategy that modulates thread-level parallelism on GPUs to control performance of both CPU and GPU applications [74].

Another software-based approach includes a technique known as Kernel Fusion, which statically transforms the code of two kernels into one. This single fused kernel is launched with an appropriate number of workgroups. Guevara et al. also propose a similar system which chooses between the execution of fused kernels and sequential execution [56]. Gregg et al. present KernelMerge, which occupies the entire GPU and dynamically invokes kernels for execution [54]. They show that resource allocation of concatenated kernels limits performance. We extend their evaluation by independently scheduling the workgroups of each kernel separately.

The three dimensional NDRange in OpenCL and CUDA has also been used in a prudent manner for increasing concurrency between kernels. Ravi et al. rely on the molding technique, which changes the dimensions of the grid and workgroups while preserving the corrects of the computation [121]. The molding technique allows the user to execute different computations in different dimensions, and thereby establishing concurrent execution. Pai et al. proposed a similar technique and associated code transformation based on iterative wrapping to produce an elastic kernel [113] [137]. The elastic kernel also implements the same strategy of using dimensional execution for establishing concurrency. These techniques rely on the developer or compiler transformation to prepare for concurrent execution. The mechanisms suggested for concurrent kernel and context execution in this thesis proposal describe a hardware-level implementation of the mechanisms on the GPU.

3.3.2 QoS-aware Shared Resource Management

Resource sharing for contention management and QoS has been long been studied in real-time embedded environments [129] [130], networking [16] [55], and CMP architectures [9] [62] [163]. Notable approaches have considered dynamic contention management for interconnect networks, memory controllers and caches in CMP architectures [9] [55] [77] [105]. Kim et al. use fairness as an objective for shared caches [76], whereas Iyer et al. propose QoS-aware shared caches for
CHAPTER 3. RELATED WORK

CMPs [62]. Similarly, Qureshi et al. implement a QoS-aware dynamic spill receiver for capacity sharing on last level caches in CMPs [120].

For GPUs, Kyu et al. dynamically adapt the priorities of GPU memory requests to meet deadlines on a MPSoC environment [65]. Their work is focused on reducing memory bandwidth contention on GPUs by rescheduling memory accesses. Aguilera et al. implement resource allocation on GPUs for concurrent applications with QoS requirements [11]. While their study focuses on CU allocation, it does not consider the effects of memory resource contention on QoS. None of the existing work has considered multi-resource contention management, and neither do they evaluate the effects of multiple resource management on QoS for applications on a multi-tasked GPU.

3.3.3 Workload Scheduling on GPUs

Scheduling of workgroups and wavefronts on compute units requires special attention when implementing kernel and context concurrency. We describe a novel workgroup scheduling scheme in Chapter 4 of this thesis proposal. In this section, we highlight some important related work in the area of workload scheduling on GPUs.

NVIDIA introduced Multi-Process Service (MPS) to support the execution of multiple MPI applications on the GPU [2]. MPS uses a time-sliced scheduler to dispatch kernels from different MPI applications to the GPU. Context switching and multiple kernel management is done using the Hyper-Q technology on the device. Several other techniques have been proposed to implement time-multiplexing on the GPU. The kernel-slicing technique launches a transformed kernel several times, and includes the launch offset so that the slice performs only a part of the computation [19, 113, 121].

Bautin et al. and Kato et al. proposed RGEM and the TimeGraph approach, respectively, to focus on scheduling of graphics applications. Their work allows a GPU command scheduler to be integrated in the device driver. They monitor the GPU commands using the CPU host, to dynamically schedule OpenGL graphics commands on the GPU using different scheduling strategies [20, 72]. Kato et al. also design the RGEM software runtime library, which is targeted at providing responsiveness to prioritized CUDA applications by scheduling DMA transfers and kernel invocations [71]. RGEM implements memory transfers as a series of small transfers to effectively hide the transfer latency. The Gdev framework is also developed using the same approach, but integrates the runtime support for GPUs in the OS [73]. Rossbach et al. design the PTask approach to schedule GPU workloads by making the OS aware of the GPU using a task-based data flow programming approach [127]. All the described strategies cannot control the assignment of the workgroups to the compute units of the GPU as they do not target the hardware schedulers of the GPU.
CHAPTER 3. RELATED WORK

Fine-grain scheduling of kernel wavefronts (or warps) using various control parameters has been suggested in the past. Rogers et al. suggested a wavefront/warp scheduling mechanism which is data cache aware [125]. Their mechanisms provide a hardware-based implementation to detect cache thrashing and reschedule active warps. As an improvement to cache aware scheduling, Rogers et al. proposed a divergence-aware scheduling of warps [126]. Similar warp/wavefront scheduling schemes have been proposed by Fung et al., which regroup different threads which diverge on a branch execution [48]. Another approach for warp scheduling was presented by Narasimhan et al., which used two-level scheduling to improve the utilization on the GPU [103]. They also tackle the issue of occupancy on the GPU by proposing the use of large sized warps.

3.3.4 GPU virtualization for Data Center and Cloud Engines

Virtualization is typically used to enable sharing of a GPU in a cloud environment. Becchi et al. proposed a runtime-level implementation in combination with VM-based cloud computing services to enable virtualization of GPUs [22]. Their runtime provides a software-level abstraction, allowing for isolation of concurrent applications which share the GPU. Li et al. suggest a GPU resource virtualization approach for multi-context execution [87], similar to NVIDIA MPS [2]. In this approach, the runtime virtualization layer manages the underlying compute and memory resources, and data transfers. Ravi et al. implement different scheduling algorithms to concurrently execute GPU applications on a cluster node [122]. Their approach also considers the execution of the application on the CPU as opposed to the GPU, for improving the utilization on the cluster node. Other approaches for GPU sharing provide support to virtualize a physical GPU into multiple logical GPUs by adding a scheduling layer in the OS (rCUDA [43], GDev [73]). These approaches manage memory transfers and kernel execution on the GPU using a software scheduler transparent to the user. They provide a software-level abstraction for sharing a GPU without implementing actual GPU partitioning. The absence of physical resource sharing can introduce false dependencies between applications and can reduce overall application throughput. Apart from GDev [73] and rCUDA [43], other studies on GPU virtualization and server management include GVim [57], vCUDA [132], Pegasus [58] and gVirtus [50]. GVim and Pegasus support QoS-aware scheduling, but do not address scheduling workloads across multiple GPUs. Similarly, vCUDA and rCUDA provide a good multiplexing mechanism using a CUDA runtime library, but do not provide scheduling strategies for GPU clusters.
3.3.5 Scheduling on Cloud Servers and Datacenter

Interference aware scheduling has been at the center of the work on datacenter and cloud server schedulers. For CPU based servers, Nathuji et al. present a system level control mechanism which considers the interference effects of cache, hardware prefetching, and memory while co-executing applications [104]. Delimitrou et al. introduced QoS aware scheduling for data centers using machine learning for interference detection. Their work on the Paragon scheduler presents the advantages and accuracy of using machine learning through recommendation systems for providing estimates of heterogeneity and interference of CPU-bound workloads [38]. The work by Mars et al. showed that the performance of Google workloads can degrade up to 2x when co-executing multiple applications [96]. They propose an offline combinatorial optimization technique to guide a scheduler to tune datacenter workloads. The recommendation strategy presented by Paragon detects interference by comparing application performance against microbenchmarks for each contending resource. Their works implements a stochastic gradient descent (SGD) model to predict performance values [38]. Their interference classification does not consider the cumulative effect of all resources.

There has been an increasing focus on work targeting techniques to improve scheduling on GPU clusters and cloud servers. Some of these heterogeneous schedulers, including StarPU [15], use an learning model to detect available resources and schedule different optimized versions of the same application targeted for different resources. The work by Sengupta et al. presents the Strings scheduler which decouples CPU and GPU execution, and guides scheduling using feedback about execution time, GPU utilization, data transfer, and memory bandwidth utilization from each GPU [131]. Phull et al. discuss an interference-aware scheduler for GPUs using a static profile of applications to guide the co-location of applications [117].

3.3.6 Address Space and Memory Management on GPUs

Concurrent execution of contexts requires efficient management of memory and address space isolation. Memory management should ensure that the contexts are not stalled due to memory contentions or overcommitment. We discuss some important prior work from the area of memory management on GPUs for concurrent execution.

The virtualization scheme for GPUs developed by Li et al. provided an effective approach using host pinned memory for asynchronous data transfers and kernel execution [87]. While pinned memory accesses provide better I/O bandwidth for single context execution, over-allocation of pinned memory may lead to degradation in performance due to increased network traffic when running multiple contexts. Work by Gelado et al. describes a memory management scheme for their
(Chapter 3. RELATED WORK)

(Asymmetric Distributed Shared Memory) (ADSM) mechanism to handle shared virtual memory transactions on GPUs [49]. Their approach for shared data transfers is transparent to the user and is completely controlled in software on the host side. Their approach achieves performance comparable to programmer-managed memory transfers, without the need for code modifications. Our work for memory management implements a technique partly controlled by the GPU and does not make excessive use of pinned memory. We compare the performance of these notable works with our approach in Chapter 5. Jog et al. proposed a memory scheduling mechanism for improving performance running multiple applications on the GPU. Their analysis was limited to studying the interaction of two-kernel applications on the memory hierarchy and the resulting global memory bandwidth utilization [69]. Pichai et al. study the impact of TLBs on cache conscious wavefront scheduling and evaluate performance when TLBs and page table walkers are placed before the private caches [118, 125]. Their work provides a good foundation for exploring address translation techniques on the GPU. The NVIDIA Grid enhances MMU hardware to keep the virtual address space of multiple processes physically separate for GPUs on the cloud [4].
Chapter 4

Adaptive Spatial Partitioning for Concurrent Kernel Execution

Modern GPU workloads demand flexible techniques to schedule multiple kernels on the device. It has become increasingly important to support effective sharing of a single GPU for parallel applications. In order to leverage the benefits of GPUs on multicore platform, a mechanism which can allow for high utilization and elastic resource allocation is required. High Performance Computing (HPC) clusters that include GPUs have even a stronger case for exploring dynamic allocation and deallocation of compute resources when applications need to share GPU resources \[50\]. In these scenarios, if we want to maximize application throughput, a kernel may need to adjust its own resource demands in order to leave room on the GPU to service the needs of other kernels executing concurrently.

As described in Chapter\[2\] concurrent execution of kernels was first supported on Nvidia’s Fermi GPUs. The left-over policy is supported on Fermi GPUs \[113\], which only schedules the next kernel if the required number of compute units are available. To broaden the scope of device level parallelism, Nvidia’s Kepler GPUs included support for concurrent execution of kernels using Hyper-Q technology \[109\]. Hyper-Q employs multiple hardware queues. However, the number of concurrent kernels executing on the GPU is limited by the number of available compute units and leads to a fixed partitioning of the GPU. The opportunity to schedule small kernels is lost in the presence of a dominant compute-hungry kernel. AMD GPUs also support multiple hardware queues using the Asynchronous Command Engine(ACEs) \[25\]. To address these issues, we introduce an adaptive spatial partitioning mechanism. Our mechanism dynamically changes the compute-unit allocation according to the resource requirements of the kernels scheduled on the device. This leads to better utilization of the GPU and avoids starvation by small-sized kernels. The fine-grained
control required for this implementation cannot be achieved at a source code level of the application. This has motivated us to develop OpenCL runtime-level support for concurrent execution using adaptive partitioning. We expose the options for adaptive and fixed partitioning to the user through the OpenCL runtime API. OpenCL 1.1 allows for the division of compute resources of the GPU to form a logical collection of compute units known as sub-devices [10]. We use multiple OpenCL command queues and sub-devices to submit workloads to the same GPU.

The remainder of this chapter is organized as follows. Section 4.1 describes how we map multiple command queues to several sub-devices. Section 4.2 describes our adaptive partitioning mechanism. Section 4.3 details the different workgroup scheduling mechanisms and partitioning policies. Sections 4.4 and 4.5 present our evaluation methodology and discuss the performance of adaptive partitioning, respectively.

Figure 4.1: High-level model of a device capable of executing multiple command queues concurrently. The lists allow for flexible mapping of NDRange workgroups to compute units on the Southern Islands device.

### 4.1 Mapping Multiple Command Queues to Sub-Devices

To enable scheduling of workloads with multiple NDRanges enqueued on multiple command queues, enhancements are made to the workgroup scheduling subsystem. These modifications are made to the scheduler of the simulated architectural model of the AMD Southern Islands (SI) GPU. The OpenCL sub-device API (clCreateSubdevices) is used to assign the requested number of compute units to the sub-device. Multiple command queues are mapped to different sub-devices to
CHAPTER 4. ADAPTIVE SPATIAL PARTITIONING

submit workloads for execution. The sub-device maintains a record of the command queues mapped to it. Figure 4.1 shows the described mapping scheme of multiple command queues to sub-devices. An NDRange which is enqueued for execution on a sub-device utilizes the compute units assigned to that specific sub-device. This leads to a fixed spatial partitioning of the GPU. Later, we will relax this constraint and describe a methodology for adaptive partitioning.

The workgroups of an NDRange can be scheduled on the compute units of a sub-device using our proposed workgroup scheduler, as shown in Algorithm 1. The implementation uses lists to track availability of compute units from a sub-device and also to track the workgroups of the NDRanges mapped to that sub-device.

1. **Available CU List**: a list of compute units (CUs) where workgroups of active NDRanges can be dispatched. This list includes all of the compute units present on the GPU. The available CU list can be changed to study the effects of power gating of CUs.

2. **Pending Workgroup (WG) List**: a list of workgroups of active NDRanges which are yet to be assigned a CU for execution. The list is queried to by the scheduler to assign the pending workgroups whenever a new CU becomes available, or when a CU has enough resources to map a new workgroup.

3. **Usable CU List**: a list of CUs assigned to each sub-device. It is a subset of the available CUs on the CU list. NDRanges mapped to a particular sub-device use CUs as described in the usable list. The CUs which have not exhausted their allocated resources are present on this list. Once a CU belonging to this list exhausts all its resources, it is marked as full and cannot be used to assign additional workgroups.

Our scheduler implementation can be described as a round robin scheduling scheme. Once a workgroup is scheduled to a compute unit, it’s local memory requirements and wavefront resource requirements do not change, which facilitates the use of a round robin algorithm. Algorithm 1 assumes a fixed workgroup-level granularity for populating the wavefront pools of compute units. We provide resource checking to verify if a workgroup can be mapped to a compute unit. We check the available local memory, wavefront pool entries, vector registers and scalar registers before scheduling a workgroup on a compute unit. This scheduling step differs from the scheduling mechanism used on older GPU systems, where the resource requirements of all workgroups could be calculated prior to kernel dispatch (occupancy). The scheduler starts with all the workgroups of the NDRange marked as pending. The Algorithm 1 acquires the usable CU list and pending workgroups for a given NDRange. The workgroup is mapped only if the required resources are available.
Algorithm 1 Workgroup scheduling mechanism to map workgroups to usable compute units.

1: **Input:** Usable Compute Unit (CU) Pool
2: **Input:** Active NDRanges
3: **Output:** Mapping of Workgroups to Compute Units
4: **for all** Active NDRanges **do**
5: Get NDRange.Usable_CUs
6: WG = Get NDRange.Pending_Workgroups
7: **for all** NDRange.Pending_Workgroups **do**
8: \( k = \text{ListHead}( \text{NDRange.Usable.CUs} ) \)
9: Schedulable = Check Resources(\( k \), WG)
10: **if** Schedulable == True **then**
11: Map Workgroup (\( k \), WG)
12: Remove WG from NDRange.Pending_Workgroups
13: **end if**
14: **end for** //Break after one scheduling iteration for each workgroup
15: **end for** //Break if No Active NDRanges

The scheduling step is repeated for each active NDRange and continues until all NDRanges finish execution.

### 4.2 Design of the Adaptive Spatial Partitioning Mechanism

The command queue mapping and workgroup scheduling scheme for *fixed* partitioning of the GPU is described in Section 4.1. This scheme can lead to resource starvation for small-sized NDRanges in applications possessing multiple NDRanges which can execute concurrently. We extend the fixed spatial partitioning mechanism to support adaptive partitioning of resources (i.e., compute units) across different sub-devices. The OpenCL sub-device API (*clCreateSubdevices*) is modified to support two new flags that specify: (1) *Adaptive* or (2) *Fixed* partitioned sub-device. The compute units, and the NDRanges that belong to an adaptive/fixed sub-device, are referred to as adaptive/fixed compute units and adaptive/fixed NDRanges, respectively. This added support for adaptive partitioning allows the runtime to allocate compute units to sub-devices based on the size of the NDRange.

Figure 4.2 presents our handler for adaptive partitioning on a GPU. The adaptive partitioning handler is invoked only whenever a new NDRange is scheduled for execution or an active
 CHAPTER 4. ADAPTIVE SPATIAL PARTITIONING

Figure 4.2: Flowchart describing the mechanism of handling adaptive partitioning of a GPU in the OpenCL runtime.

1. Dispatcher:
   - Checks for new NDRanges
   - Dispatches for scheduling.
   - Invoked only when an NDRange arrives or departs from the device.

2. NDR Scheduler:
   - Checks sub-device property of new NDRange
   - Manages Pending NDRanges
   - Schedules Fixed NDRanges and calls Load-Balancer

3. Load Balancer:
   - Handles all adaptive NDRanges
   - Assigns CUs to adaptive NDRanges according to the NDRange size
   - Maintains 1 CU per active adaptive NDRange
   - Calls Workgroup scheduler

Stages of adaptive partitioning.

- New NDR Arrival?
  - Yes: New → PendingList
  - No: Curr_NDR = Head(PendingList)

- Is PendingList Empty?
  - Yes: Execute Load Balancer
  - No: Execute NDR Scheduler for Curr_NDR

- Get NDR Sub-device property
  - Check sub-device property
    - Adaptive: Execute Load Balancer
    - Fixed
      - Get CUs required for Sub-device (CU_required)
      - N = Free_CUs + Adapt_CUs - active_adapt_NDRs

- N > CU_required?
  - Yes: Add NDR → PendingList
  - No: Add CU_required to NDR usable CU list

- New Adaptive NDR
  - Yes: Add new NDR to Adaptive NDR list
  - No: M = Free_CUs + Adapt_CUs

- Assign M CUs to adapt_NDRs according to ratio of the NDR size

- For each active_adapt_NDRs
  - CU / UNDR ≥ 1
    - Yes: Ensure CU / active_adapt_NDRs ≥ 1
      - Execute Workgroup Scheduler (Algorithm 1)
      - Exit Loop
    - No: Move Adaptive NDR → Active Adaptive NDRranges
  - No: Move Adaptive NDR → PendingNDRanges
CHAPTER 4. ADAPTIVE SPATIAL PARTITIONING

NDRange completes its execution on the GPU. The adaptive handler includes three modules: (1) a Dispatcher, (2) a NDRange Scheduler, and (3) a Load Balancer.

- **Dispatcher**: The dispatcher maintains the *Pending List* of NDRanges queued for execution on the device. If the pending list is not empty, the NDRange which is at the head of this list is scheduled for execution. If the pending list is empty, the Load Balancer is invoked to reassign resources to the active NDRanges.

- **NDR Scheduler**: The NDR scheduler is run by the dispatcher whenever an NDRange begins execution on the device. The sub-device property of the NDRange is checked by the scheduler. For a sub-device with the *fixed* property type, the scheduler checks for available compute units from the free list, as well as from the adaptive compute units. If the number of available compute units is less than the number requested, the associated NDRange is added to the pending list. The Load Balancer is invoked once a fixed sub-device is assigned with the requested number of compute units or if the incoming NDRange is labeled as *adaptive*.

- **Load-Balancer**: The Load-Balancer manages resource allocation for all of the adaptive NDRanges on the device. It forms sets of compute units proportional to the size of each adaptive NDRange. After mapping compute units to the adaptive NDRanges, the load balancer executes the workgroup scheduler, as described in Algorithm 1. The load balancer prevents reallocation of an active NDRange by maintaining at least 1 compute unit for each active adaptive NDRange. An incoming adaptive NDRange is added to the pending list to avoid reallocation of an active adaptive NDRange. Our mechanism ensures that there is at least 1 compute unit available for each adaptive NDRange executing on the device. A long running adaptive NDRange can be terminated to prevent blocking of resources. Such termination is done using a timer, which can be configured by the user. The default value of timer is set to 10 billion GPU cycles.

4.2.1 Compute Unit Reassignment Procedure

Figure 4.3 shows the steps involved in transfer of one compute unit between kernels. In the *detection* phase, the CU to be transferred is identified using the adaptive partitioning mechanism. The CU is *locked*, thereby disallowing any more workgroups from being scheduled to the CU. The *Quiesce* stage allows the compute unit to complete the execution of already active workgroups. The Quiesce stage of the transfer is variable and depends on the computation being executed by the kernel. Once the CU is free, it is registered with the adaptive handler (NDR Scheduler). The transfer
4.2.2 Case Study for Adaptive Partitioning

Figure 4.4 shows a case where our adaptive partitioning mechanism is applied. In the first stage, only one NDRange executes on a sub-device with the adaptive property. The load-balancer allocates 22 compute units to NDRange0, as shown in Figure 4.4a. A second NDRange on a different adaptive sub-device is scheduled for execution in the second stage, as shown in Figure 4.4b. The load balancer allocates 12 compute units to NDRange1, based on the ratio of the sizes of all adaptive NDRanges (i.e., NDRange0 and NDRange1). This results in a change in allocation for the two compute units (CU#20 and CU#21), which are now reassigned from NDRange0 to NDRange1.

stage, transfers the CU to another kernel and updates the resource tables. As a last step, the CU is unlocked and can accept more workgroups from the new kernel.
CHAPTER 4. ADAPTIVE SPATIAL PARTITIONING

The reassigned compute units are blocked from scheduling new workgroups from NDRange0. The mechanism allows both compute units to complete execution of the previously scheduled workgroups from NDRange0. The compute units of the GPU cannot be pre-empted once a workgroup is scheduled for execution. Hence, the adaptive partitioning mechanism implements a strict blocking policy once an active compute unit is reassigned. NDRange1 can schedule workgroups to CU#20 and CU#21, but only after there are no pending workgroups from NDRange0 running on either compute units. Figure 4.4c shows a successful mapping of both NDRanges after adaptive partitioning is performed.

4.3 Workgroup Scheduling Mechanisms and Partitioning Policies

Figure 4.5: Workgroup scheduling mechanisms, (a) Occupancy-based scheduling and (b) Latency-based scheduling. The Kernel A consists of 8 workgroups, which are mapped to the compute units (CUs), using one of the two scheduling mechanisms.

1. **Occupancy-based:** The scheduler maps pending workgroups of an NDRange to each available compute unit. The scheduler only moves to the next compute unit if the maximum
allowable number of workgroups per compute unit are mapped, or when the compute unit ex-
pends all of its resources. This mechanism assures maximum occupancy of a compute device
and can help preserve resources.

2. **Latency-based**: The scheduler iterates over the compute units in a round-robin fashion and
assigns one pending workgroup to each usable compute unit. The scheduler does not consider
whether the compute units are fully occupied. The mapping process continues until all pend-
ing workgroups are assigned. This mechanism ensures that each usable compute unit does
useful work on each GPU cycle and minimizes compute latency.

A description of each of these scheduling schemes is illustrated in Figure 4.5.

We also introduce three different partitioning policies which can be adopted by users to
suit the requirements of their applications.

- **Full-Fixed partitioning**: This policy maps the NDRanges of each application to a sub-device
  with a fixed property. Kernels using the full-fixed partition policy are limited to use the
  compute units allocated to their sub-device. The compute unit allocation for the particular sub-
device cannot be changed. The users can benefit from a dedicated compute unit allocation,
  but can also suffer from a lack of flexibility as new compute units become available.

- **Full-Adaptive partitioning**: Full-adaptive partitioning maps different NDRanges of an ap-
plication to sub-devices possessing the adaptive property. In this policy, the compute units are
  assigned to each sub-device by the load-balancer, as described in Section 4.2. The NDRange
  that is using adaptive partitioning is allowed to occupy the entire GPU in the absence of any
  other NDRange.

- **Hybrid partitioning**: The hybrid partitioning policy uses a combination of adaptive and
  fixed sub-devices for mapping different NDRanges of an application. The user can choose
to allocate only a few NDRanges to a fixed sub-device and can allow other NDRanges to be
mapped adaptively to sub-devices. Allocating the performance critical NDRanges to a fixed
partition, while allocating adaptive sub-devices to other NDRanges, should yield performance
benefits.
CHAPTER 4. ADAPTIVE SPATIAL PARTITIONING

4.4 Evaluation Methodology

4.4.1 Platform for Evaluation

Our evaluation requires fine-grained control over workgroup scheduling, NDRange creation, and support for OpenCL sub-devices on the targeted platform. We have implemented multiple command queue mapping, a new workgroup scheduler, and an adaptive partitioning handler using Multi2Sim. The runtime layer provided by Multi2Sim includes our library that extends the OpenCL API. Support for simulating multiple command queues and sub-devices is added within the OpenCL runtime layer. The resource usage tracking within the compute units, and information regarding the runtime requirements of each workgroup, is added to the driver layer. To execute applications on Multi2Sim, we link our program with the Multi2Sim OpenCL runtime library. We simulate the AMD GCN-based HD Radeon 7970 GPU model in the results presented here. The details of the configuration of this simulated GPU are shown in Table 4.1.

<table>
<thead>
<tr>
<th>Device Configuration</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Unit Config</td>
<td></td>
</tr>
<tr>
<td># of CU’s</td>
<td>32</td>
</tr>
<tr>
<td># of Wavefront Pools / CU</td>
<td>4</td>
</tr>
<tr>
<td># of SIMD Units / CU</td>
<td>4</td>
</tr>
<tr>
<td># of lanes / SIMD</td>
<td>16</td>
</tr>
<tr>
<td># of vector reg / CU</td>
<td>64K</td>
</tr>
<tr>
<td># of scalar reg / CU</td>
<td>2K</td>
</tr>
<tr>
<td>Frequency</td>
<td>1GHz</td>
</tr>
<tr>
<td>Memory Architecture</td>
<td></td>
</tr>
<tr>
<td>L1 (1/CU)</td>
<td>16KB</td>
</tr>
<tr>
<td># of shared L2</td>
<td>6</td>
</tr>
<tr>
<td>L2 Size</td>
<td>128KB</td>
</tr>
<tr>
<td>Global Memory</td>
<td>1GB</td>
</tr>
<tr>
<td>Local Memory / CU</td>
<td>64K</td>
</tr>
</tbody>
</table>

Table 4.1: The device configuration of the AMD HD Radeon 7970 GPU.

We also study the behavior of multiple command queue-based applications on devices equipped with Nvidia’s Hyper-Q technology. The evaluation for this study is performed on a Nvidia Tesla K20c GPU hardware.

4.4.2 Evaluated Benchmarks

The benchmarks described in this section are used to evaluate the potential for the different partitioning policies. Existing suites such as Rodinia and Parboil are not suitable for our study since they only utilize single command queue mapping [32, 136]. We use benchmarks representing real
CHAPTER 4. ADAPTIVE SPATIAL PARTITIONING

world applications from different domains of computation. Each of the benchmarks uses multiple command queues which map to unique sub-devices.

1. **Matrix Equation Solver (MES):** Application is a Linear Matrix based solver implemented on GPUs \[134\]. The benchmark evaluates the equation \( C = \alpha (AB^{-1}) \ast \beta ((A + B) \ast B) \), where \( A, B, \) and \( C \) are square matrices. The computation is done in three parts:
   (a) \( C_0 = \alpha (AB^{-1}) \),
   (b) \( C_1 = \beta ((A + B) \ast B) \) and
   (c) \( C = C_0 \ast C_1 \).
Parts (a) and (b) execute in parallel using different subdevices. Part (c) is computed using the entire GPU device.

2. **Communication Channel Analyzer (COM):** The application emulates a communication receiver with 4 channels. Each channel receives the same data from the sender and processes the data in different ways.
   - Channel 0: Performs lossless packet processing using a series of data manipulations.
   - Channel 1: Lossy packet processing using Gaussian Elimination and Reduction over data packet.
   - Channel 2: Perform priority-based packet processing over the received data (Lossless for high priority and lossy for low priority data).
   - Channel 3: Scrambles the received data packet with a pseudo-random code (PRC).

Each channel is assigned a separate command queue and executes on different compute units.

3. **Big Data Clustering (BDC):** The application belongs to the domain of big-data analysis using three computational kernels. Each kernel is mapped to a different command queue. The first kernel performs clustering of input data, involving all of the data points. This kernel results in a large NDRange size, with long-running computations. The second kernel is a reduction kernel, where each thread operates over 20 clusters to reduce those clusters to 100 representative points. The third kernel is a sorting computation which sorts the representative points. All three kernels have different NDRange sizes.

4. **Texture Mixing (TEX):** This application belongs to the graphics and image processing domain. TEX mixes three different textures to form one blended texture output \[41\]. The first kernel operates on an image to extract the background using a filtering computation. The background kernel has the largest NDRange size. The second kernel performs blurring of a
CHAPTER 4. ADAPTIVE SPATIAL PARTITIONING

texture to form a saturation hue for the final image. The third kernel extracts the foreground from a texture to apply to the final image. All the kernels have different NDRRange sizes and are mapped to separate command queues for simultaneous execution.

5. Search Application (SER): Search application benchmark from the Valar benchmark suite [99]. Application performs a distributed search using two kernels. The kernels are mapped to different command queues for parallel execution.

4.5 Evaluation Results

In this section, we present results evaluating the performance of the proposed adaptive spatial partitioning, command queue mapping, and workgroup scheduling mechanism.

4.5.1 Performance Enhancements provided by Multiple Command Queue Mapping

We evaluate the execution performance of the Set 1 benchmark applications when applying multiple command queue mapping for a fixed-partition run on both a simulated AMD device and an actual Nvidia K20 GPU device with Hyper-Q. We use single queue mapping as the baseline for evaluating speedup provided by multiple command queues on both platforms. The execution performance is shown in Figure 4.6.

COM enqueues 4 kernels to the same device by mapping each kernel to 8 compute units on the SI GPU. The MES application also enqueues 2 kernels by mapping each of them to 16 compute units on the AMD Southern Islands device. Both applications experience a performance gain of 2.9x over the single-queue implementation. Applications using multiple command queues on the fixed-partition SI device exhibit an average performance speedup of 3.1x over the single command queue. Concurrent execution of kernels tries to overlap computation of each kernel and reduces execution time of the application.

The applications using multiple command queues on the K20 GPU show an average performance gain of 2.2x over a single queue implementation. The multiple command queues on the Kepler GPU get mapped to different hardware queues using the Hyper-Q feature and are executed concurrently. We can see significant benefits for applications consisting of kernels which do not saturate GPU resources. The applications with all kernels optimized to saturate the GPU would result in marginal performance improvements.
CHAPTER 4. ADAPTIVE SPATIAL PARTITIONING

Figure 4.6: Performance improvement when using multiple command queues mapped to different sub-devices on the simulated SI GPU and K20 GPU.

Figure 4.7: Memory hierarchy of the modeled AMD HD 7970 GPU device.
CHAPTER 4. ADAPTIVE SPATIAL PARTITIONING

Figure 4.8: L2 cache efficiency for applications implemented using multiple command queues for Set 1 benchmarks on the simulated SI GPU.

4.5.2 Effective Utilization of Cache Memory

We explore the performance of the L2 cache on a simulation model of the SI GPU using multiple command queues. Our L2 cache study is motivated by the fact that the L2 is the first shared memory resource (L1s are private on a GPU). As shown in Figure 4.7, each compute unit has its own L1 cache unit, while all of the CUs share six L2 cache units. Applications using multiple command queues execute on different sets of compute units. Kernels which utilize the same input data benefit from the shared L2 caches.
CHAPTER 4. ADAPTIVE SPATIAL PARTITIONING

The L2 cache efficiency is evaluated in Figure 4.8. Applications such as MES, COM and BDC show a 22% improvement in the L2 cache hit rate when using multiple command queues. As the global writes are visible to the L1 and L2 caches on SI GPUs, the single command queue implementation flushes the L2 cache after completion of the kernel [60]. Such L2 cache flushes do not occur when using multiple command queues. This improves the cache hit-rate of concurrent kernels operating on the same input data. SER and TEX enqueue kernels which operate on different input data. Hence, the kernels do not access the same data and exhibit a cache hit-rate similar to their single command queue implementation. BDC schedules three different kernels which use the same data buffers, and hence shows an improvement in cache hit rate.

4.5.3 Effects of Workgroup Scheduling Mechanisms and Partitioning Policies

Next, we evaluate the performance of the workgroup scheduling and partitioning policies described in Section 4.3. The performance of occupancy-based scheduling for different partitioning schemes is observed in Figure 7.8a. Full-fixed partitioning yields superior performance as compared to single command queue implementations, as seen in Figure 4.6. But fixed partitioning also limits the applications to use only a fixed set of compute units for each NDRange. An active NDRange cannot use all the available compute units on the GPU in the fixed partition, even if it is the only active NDRange on the device. This leads to poor occupancy on the GPU.

Using full-adaptive partitioning lets the runtime handle the assignment of compute units to the sub-devices. It leads to better occupancy on the GPU by adapting to the resource requirements of the NDRanges. All applications (except for BDC) show an average improvement of greater than

![Figure 4.9: Execution performance of benchmarks for different workgroup scheduling mechanisms and different partitioning policies.](image-url)
26\% in execution performance using full-adaptive partitioning. The BDC application shows 32\% degradation in performance compared to full-fixed partitioning. The largest NDRange in BDC is a long-running kernel executing lengthy computations. This NDRange occupies a large number of compute units when using adaptive partitioning. This results in a shortage of compute units for other smaller NDRanges present in the BDC application.

The hybrid partitioning approach assigns smaller NDRanges in the applications to adaptive sub-devices and the larger NDRanges to fixed sub-devices. As a result, BDC and TEX show an improvement in execution performance of 35.7\% and 23\%, respectively (as compared to full-adaptive partitioning). Hybrid partitioning produces a compute unit assignment similar to full-fixed partitioning for applications which use similar-sized NDRanges for computations. Thus, MES and SER (which have similar-sized NDRanges) do not receive any significant advantage in performance when using hybrid partitioning. In contrast, COM shows a performance improvement of 16.6\% over adaptive partitioning when using hybrid partitioning. This improvement is attributed to the improved load-balancing of the adaptive NDRanges in the COM application. Assigning large NDRanges to adaptive sub-devices in hybrid partitioning may lead to over-subscription of that sub-device and thereby degrading the performance of the large NDRange.

Latency-based scheduling yields the best performance when using full-fixed and hybrid partitioning policies, as seen in Figure 7.8b. The compute units assigned to the fixed sub-device are not re-assigned during execution. This enables latency-based scheduling to use all compute units as effectively as possible in each fixed sub-device when applying hybrid and full-fixed partitioning. Latency-based scheduling experiences some degradation in performance for full-adaptive partitioning. The first NDRange scheduled for execution occupies the entire GPU and the subsequent NDRanges do not receive any free compute units for scheduling the computation. This leads to a delay due to re-assignment of compute units from the first NDRange in the case of full-adaptive partitioning. The first NDRange of the BDC application is a large long-running kernel and experiences a 96\% increase in execution time (cycles) versus using full-fixed partitioning and latency-based scheduling. All other applications show an average increase of 45.2\% in execution time (cycles) for full-adaptive partitioning using latency-based scheduling.

4.5.4 Timeline Describing Load-Balancing Mechanisms for the Adaptive Partitioning

The load-balancer described in Section 4.2 allocates the compute units of the GPU to different adaptive sub-devices according to the ratio of NDRange sizes. The allocation of compute units considers the number of active adaptive NDRanges on the GPU. The process of re-allocation
Figure 4.10: Timeline showing re-assignment of compute units for each NDRange of the applications (a) COM, (b) TEX and (c) SER using full-adaptive partitioning.
of compute units is described in Section 4.2. All applications execute using the full-adaptive partitioning policy.

Figure 4.10 shows the timeline for allocation of compute units to different NDRanges of an application. The NDR0 (i.e NDRange 0) is the first to be executed on the GPU for all the applications. NDR0 arrives when the GPU is unoccupied and hence receives a large number of compute units (20 CUs for COM, 22 CUs for SER, and 22 CUs for TEX). As new NDRanges arrive on the device, the load-balancer re-assigns the compute units. An equal number of compute units are assigned to same-sized NDRanges in COM and SER. As observed in Figure 4.10b, the NDRanges of different sizes in TEX are allocated with different number of compute units when the applications reach a steady-state. The load-balancer re-assigns the compute units whenever an NDRRange completes execution. As seen in Figure 4.10a, NDR2 and NDR3 in application COM are assigned additional compute units as NDR0 and NDR1 complete their execution. The same effect is observed for NDR1 of the SER application and the NDR2 and NDR3 of the TEX application. The time taken to complete the re-assignment of the compute units from one NDRange to another can also be observed from Figure 4.10. The average overhead incurred while re-assigning one compute unit across two NDRanges for all the applications is 9300 cycles. Figure 4.11 shows the breakdown of the time involved in transferring the compute unit from one kernel to another.
4.6 Summary

The previous sections highlight the performance gains produced when employing multiple command queue mapping. A performance speedup of 3.4x is obtained with the use of multiple command queue mapping, as observed in Sections 4.5.1. Benchmarks operating on shared data buffers showed increased L2 cache hit rates of more than 72%. The adaptive partitioning of compute units results in performance gains and better utilization of device resources. The applications with large NDRanges (BDC) obtained better performance on hybrid and fixed partitioning schemes. The different partitioning mechanisms can enable the user to select an appropriate policy to suit their application. A developer with no inherent knowledge of the device architecture can rely on the adaptive partitioning for their application. Developers with a better understanding of the device architecture can select the hybrid or fixed partitioning policy to tune their application for maximum performance.
Chapter 5

Runtime and Hardware Assisted Multi-Context Execution

We have explored the limitations of kernel-level concurrency on current GPUs in Chapter 4. The concurrency supported on current GPUs is limited to kernels belonging to the same application context on the GPU. An application context defines the virtual memory space created for the execution of one or more kernels. Many applications include only a single kernel or a few small-sized kernels that do not fully utilize the massive compute resources on the GPU. For GPUs used in datacenter and cluster environments, such wastage of compute resources reduces the power/performance efficiency of the cluster. The concurrency mechanisms present on current GPUs should be expanded to allow for simultaneous execution of multiple application contexts, leveraging any or all unused resources. New and efficient mechanisms are required to manage memory and compute resource allocation to enable multiple application contexts on GPUs.

In this chapter, we describe a novel hardware and runtime-level implementation to support execution of multiple application contexts on the same GPU. We develop the Context-Management-Layer (CML) as a part of the OpenCL runtime to manage multiple contexts launched for execution on the device. The CML and the Command Processor (CP) present on the GPU are responsible for handling data transfers for each context executing concurrently. The CP is also responsible for creation, execution, and destruction of contexts. We extend the role of the CP to perform additional functions such as memory management, compute resource management, and kernel scheduling. We characterize the firmware tasks on the GPU to better understand the required design complexity of the CP. We pursue a hardware-based scheme to provide virtual address space isolation in the shared L2 caches and TLB for multi-context execution. The CML, along with the CP, and memory isolation mechanism, provide an efficient framework to host multiple contexts on a GPU.
CHAPTER 5. MULTI-CONTEXT EXECUTION

The remainder of the chapter is organized as follows: Section 5.1 discusses the motivation for our work. The architectural and runtime modifications to support the proposed design are described in Section 5.2. The details for resource management using CML and CP are provided in Section 5.3 and 5.4. Section 5.5 provides the evaluation methodology, Section 5.6 provides the evaluation results, and conclusions are presented in Section 5.8.

5.1 Need for Multi-Context Execution

Figure 5.1: Normalized execution time and average resource utilization for the Rodina kernels on NVIDIA and AMD GPUs.

From our previous study described in Chapter 2 and Chapter 4, we know that the Hyper-Q technology and Asynchronous Compute Engine (ACE) units are used to support concurrent kernel execution on NVIDIA Kepler and AMD GCN GPUs, respectively [95, 109]. We evaluate the efficiency of these concurrent kernel execution mechanisms on modern GPUs. Figure 5.1 shows the execution time of six distinct multi-kernel benchmarks, selected from the Rodinia suite [32] and normalized to the K10 for the NVIDIA GPUs and normalized to HD7970 for the AMD GPUs. The NVIDIA K10 and the AMD HD7970 do not support concurrent kernel execution, whereas the NVIDIA K40 and AMD R9-290X GPUs provide support for concurrent kernel execution using hardware queues. Each pair of GPUs from the same vendor possess a comparable number of compute cores, but differ in their support for concurrent kernel execution. When running concurrent kernels, we observe an average speedup of 1.39X on the NVIDIA K40 and 1.31X on the AMD R9-290X GPU, respectively. Although we observe impressive speedups, most of the applications do not fully utilize the compute resources (compute units, global memory, and hardware queues) on the GPU. The average resource utilization reported by vendor profiling tools [12, 5] is 67% on
NVIDIA K40 and 64% on AMD R9-290X. The issue of unused compute resources can be partially addressed by allowing multiple application contexts to execute concurrently on the GPU. Physical sharing of GPU resources with other applications can result in better resource utilization, which can lead to an overall increase in application throughput.

Implementing multi-context execution faces some key challenges for global memory management and GPU task control (discussed in Chapter 2, Section 2.5.2). Apart from these challenges, simultaneous execution of multiple contexts requires isolation of memory state and architectural state between contexts. This requires either logically or physically partitioning compute resources, and management of the L2 cache and TLB. We implement hardware-based L2 cache and TLB management mechanisms.

5.2 Architectural and Runtime Enhancements for Multi-Context Execution

In this thesis we consider the support needed to allow for multi-context execution using OpenCL, a programming framework for heterogeneous computing maintained by Khronos [100]. The OpenCL runtime environment can be visualized as a two-level structure, as shown in Figure 5.2a. The upper-level runtime consists of the OpenCL API library and data structures to manage memory and program objects for the GPU computation. The lower-level runtime consists of the OS-level interface and the Application Binary Interface (ABI) for handling the execution of the OpenCL program.

5.2.1 Extensions in OpenCL Runtime

We implement a Context Management Layer (CML) which monitors and performs the necessary bookkeeping for each context to be executed on the device. The CML is an extension to the low-level runtime of OpenCL.

Figure 5.2a shows the operation of the CML with respect to the runtime and driver stack of the GPU. The CML maintains information about the contexts created on the host through data structures maintained in the upper-level runtime. In addition to context dispatch, the CML is also responsible for the movement of data buffers between the host and the device for each executing context. Our CML layer implements a sophisticated Transparent-Memory-Management (TMM) mechanism using copy engine\textsuperscript{1} and the command processor (CP) present on the GPU to control data buffer transfers to/from the GPU.

\textsuperscript{1}Copy engines are implemented using the DMA units present on the GPU
We extend the OpenCL context creation API to provide support for multi-context execution. The OpenCL context can be clearly identified for shared execution on the GPU using the new extensions. This provides GPU programmers with the option of indicating the minimum resource requirement for their applications. The clCreateContext API is extended to support three additional properties for context-sharing.

- **cl_context_shared_75**: Creates an OpenCL context allotted 75% of the compute-resources available on the GPU.

- **cl_context_shared_50**: Creates an OpenCL context allotted 50% of the compute-resources on the GPU.

- **cl_context_shared_25**: Creates an OpenCL context allotted 25% of compute-resources on the GPU.

While we select a set of initial allocations based on the maximum number of contexts we evaluate in this thesis (i.e., a maximum of 4 concurrent contexts), these values could easily be modified to support a larger number of contexts. The values provide initial allocation shares that are under programmer control. If a sharing allocation is not specified, the context is treated as a non-shared context and is allotted 100% of the GPU resources. A context specifying an allotment
less than 100% can be promoted to utilize all available resources on the GPU. Users with knowledge about the resource requirements of their application can choose a specific sharing property. Other users (e.g., those with no knowledge about their resource requirements) can select the minimum sharing value \((cl\_context\_shared\_25)\) and rely on our dynamic allocation mechanism to assign additional resources if available. We use this set of sharing allocations in our evaluations to analyze the impact of varying resource utilization on memory behavior and application throughput, while executing multiple concurrent contexts.

**Context Dispatch using CML:**
Each context dispatched for execution on the GPU can issue commands to the device using the OpenCL command queue-based APIs. The CML manages the dispatch of these OpenCL contexts for execution on the GPU. As shown in Figure 5.2b, CML maintains one context queue for dispatching both shared contexts and non-shared contexts. The dispatcher implements a FCFS policy for selecting contexts for execution. The dispatcher checks for availability of compute resources before selecting which contexts to dispatch. If the available compute resources are insufficient for a selected context, the dispatcher waits for the GPU to release resources.

![Figure 5.3: Address space isolation using CID for (a) L2 cache lines and (b) TLB entries.](image)

**5.2.2 Address Space Isolation**
When executing multiple kernels from a single context, the virtual address space for all the kernels is the same. But for multi-context execution, the memory hierarchy has to add support to enforce memory protection between different address spaces, preventing address space aliasing [63]. GPUs implement memory hierarchies with two forms of address translation [118]. The memory hierarchy can be accessed using a virtual address, with translation being performed in the memory controller. Alternatively, the address translation can be performed at private levels of the hierarchy.
in the compute unit of the GPU. Current AMD GPUs use the first method of managing address
translation, relying on the memory controller. For effective sharing of caches and TLB, a static
partitioning of cache has been proposed in past work [83]. An alternative approach is to include
context specific information in the GPU compute units [143]. We use the latter approach to under-
stand the effects of multi-context execution on the memory hierarchy of the GPU, and avoid the use
of complex additional hardware required for implementing static cache partitioning [83].

We include a 2-bit context-identifier (CID) tag for address space isolation in the shared
L2 cache lines and TLB segments. The 2-bit CID supports 4 concurrent contexts. We include a
16-bit register for each compute unit to save the CID, which is appended to each L2 access issued
from that compute unit. We also include a page table register for each context, to allow efficient
page-table walks on a TLB miss. The additional tag bits and registers add some storage overhead to
the memory hierarchy. The CID length can be increased to support additional concurrent contexts.
Figure 5.3 illustrates our proposed CID-based tagging for L2 cache and TLB.

5.3 Resource Management for Efficient Multi-Context Execution

5.3.1 Transparent Memory Management using CML

Along with context scheduling, the CML also handles data transfers between the host
and the GPU device for every context. This Transparent-Memory-Management (TMM) mechanism
is implemented using the CML, device driver, copy engines, and the CP. The TMM maintains
information about all data buffers created in each context in the form of lists. It monitors the
command queues to determine the order of kernel execution, and also records the number of the
kernels accessing each data buffer. The TMM also ensures that the data buffers required by each
kernel are present on the GPU when kernel execution begins. On the GPU side, the CP keeps
track of the data buffers on the GPU and monitors the global memory usage. Figure 5.4 illustrates
TMM-controlled data transfer procedure from host to device (\(\text{Host} \to \text{GPU}\)), and from device to host
(\(\text{GPU} \to \text{Host}\)) involving CP and the CML. The TMM in the CML manages the host-to-device transfer
of data buffers. The TMM checks the context list and kernel list to determine the active kernel. It
chooses the buffer required for that kernel from the buffer list (\(\text{Host} \to \text{GPU}\)) and fetches the data associated
with that buffer from system memory (\(\text{Host} \to \text{GPU}\)). The copy engines are used (\(\text{Copy Engines} \to \text{GPU}\)) to transfer data to the
GPU global memory (\(\text{Copy Engines} \to \text{GPU}\)). On completion of the transfer, the CP inserts the buffer on the buffer list
on GPU side. This action completes the transfer from the host and signals the availability of the
buffer on the GPU (\(\text{Host} \to \text{GPU}\)). The CP also analyses if the new buffer is a hot-buffer or cold-buffer, and
updates the respective buffer list. The transfer of cold data buffers is initiated by the CP using the

64
buffer lists (1). Data is fetched from global memory (2) and transferred to the host using the copy engines (3). Once the buffer is stored in host memory (4 and 5), the CML updates the buffer list to indicate the presence of the buffer on the host side (6). The *host-to-device* and *device-to-host* transfers can occur simultaneously using multiple copy engines, under the control of CML and CP, respectively. All data transfers can be overlapped with application execution on the GPU, either reducing or eliminating the cost of communication. The hardware-software-based management of TMM ensures efficient data transfer control for multi-context execution. Hiding communication overhead is another motivation to pursue concurrent context execution.

Figure 5.5 shows the operation of the TMM for three contexts scheduled for execution on the GPU. This example shows the state of the GPU physical memory, the executing NDRanges, and the state of the copy engines. At time $t_0$, the copy engine transfers the data required for execution for the first kernel belonging to each context. At time $t_1$, the copy engine transfers data buffers
CHAPTER 5. MULTI-CONTEXT EXECUTION

Figure 5.5: TMM operation managing three concurrent contexts.

(c1, d1) to the device, required for executing future kernels (0,k1, 1,k1, 2,k1). The data buffers (e.g., a1 and b2) are maintained on the device, as they are required by the next kernel. At times t2 and t3 we can see the activity of the TMM mechanism when executing three concurrent contexts. At t2, the copy engine transfers cold data buffers (a0, b1, a2) back to the host. At time t4, only one kernel (0,k3) is executing on the device, and the copy engine concurrently transfers data back to the host. If a data buffer that was created by the context does not find enough global memory available on the GPU, the TMM allocates host-based pinned memory to the buffer. All of the changes to this buffer are updated in pinned memory on the host side using the PCIe interface. The TMM manages the data buffers efficiently and minimizes the need for pinned memory buffers.

5.3.2 Remapping Compute-Resources between Contexts

The set of kernels present in a single context may not fully utilize the compute resources allocated to that context. The CP maintains a usage log of each compute unit assigned to a context. If a context exhausts all the allocated resources for execution of its kernels, it can utilize the un-
used compute units (CUs) belonging to other contexts. For the CUs that are assigned to a context, but never used, they are considered *mappable* CUs. The process of remapping unused compute resources helps in achieving better utilization of such resources.

**Algorithm 2** Dynamic Compute Unit remapping using the CP.

```plaintext
1: // Process for each context (Ctx)
2: if Mappable_CUs != NULL then
3:     for all Exec_ctx do // Other Executing contexts (Exec_ctx)
4:         if (Exec_ctx.Pend_work != NULL) then
5:             // Assign 1 mappable CU to each executing context
6:             Exec_CTX.CUs ← Exec_ctx.CUs + 1
7:             Ctx.Mappable_CUs ← Ctx.Mappable_CUs - 1
8:         end if
9:     end for
10: else // No mappable CUs
11:     if (Ctx.Pend_work != NULL) and (Ctx.CUs > Ctx.min_CUs) then
12:         Reclaim_Min(Ctx) // Reclaim minimum requested CUs
13:     end if
14: end if
15:
16: Reclaim_Min(Ctx) // Procedure to gain minimum requested CUs
17: while Ctx.CU != Ctx.Min_CUs do
18:     for all Exec_ctx != Ctx do
19:         if Exec_ctx.CUs > Exec_ctx.min_CUs then
20:             Ctx.CU ← Ctx.CU + 1
21:             Exec_ctx.CU ← Exec_ctx.CU - 1
22:         end if
23:     end for
24: end while
```

The CP performs the remapping task described in Algorithm 2 every 10 million cycles (a design choice) of GPU execution. CP refers to the usage log of the CUs that belong to the context and checks for *mappable* CUs. The CP reassigns *mappable* compute units to other contexts that have already fully utilized their assigned compute units, and have additional work to complete. The CP iterates over the other executing contexts in a round-robin fashion and assigns one *mappable* CU.
CHAPTER 5. MULTI-CONTEXT EXECUTION

per iteration to every executing context. This process continues until all mappable CUs are assigned.
The reassignment of mappable CUs can lead to situations where a context continues execution with
a smaller number CUs than the number requested. If any such context has additional pending work,
the CP performs a CU reclaim procedure. The CP restores the minimum requested CUs for that
context by reclaiming CUs from other contexts with surplus CUs. Our remapping strategy ensures
that the mappable CUs are evenly distributed across the other executing contexts. It also maintains
fairness by providing a mechanism to reclaim the CUs. This helps the contexts complete their
execution faster. The time period used for remapping is kept configurable.

5.4 Firmware Extensions on Command-Processor (CP) for Multi-
Context Execution

The Command Processor on the GPU has traditionally been used to schedule work
through commands from the graphics runtime. The CP is used to initialize the GPU units, in-
cluding the compute units and the rasterizer for executing various shaders belonging to the graphics
pipeline [37]. The AMD R9-290X GPU architecture utilizes the CP for tasks such as global data
sharing among kernels, scheduling of kernels, and managing cache coherence [95]. The CP plays
an important role in managing the execution of multiple contexts on the GPU. We extend the func-
tionality of the CP by adding complex firmware tasks to manage compute resources and data buffers
on the GPU. The CP also establishes communication with the runtime through the CML.

The firmware tasks implemented on the CP include:

1. **Resource Checking**: The CP receives information regarding the initial sharing allocation for
   the context and verifies the availability of compute resources requested by the context. This
   verification step checks for the availability of physical global memory, CUs and hardware
   queues.

2. **Resource Partitioning and Remapping**: The CP allocates compute units and hardware
   queues to each context according to their sharing allocation. The CP also initiates and controls
   the CU remapping mechanism described in Section 5.3.2.

3. **TMM control and global memory management**: The amount of global memory made
   available to the context is based on the sharing allocation requested by the context. The
   TMM mechanism shown in Figure 5.4 uses the CP to monitor the use of global memory by
   each context. The CP makes decisions regarding transfers of non-persistent data buffers from
   the GPU to the host, and initiates such actions using the copy engines.
CHAPTER 5. MULTI-CONTEXT EXECUTION

4. **Kernel Scheduling:** The CP allocates and controls the hardware queues to manage scheduling of the kernels belonging to an application context \[13, 95\]. For a context with multiple kernels, each kernel can be managed by a separate hardware queue. The hardware queues control the assignment of compute units to the multiple concurrent kernels of an application context. The hardware queues can execute data-independent kernels in parallel in an application context. In addition to the context-level concurrency implemented by the CP, kernel-level concurrency for each context is obtained by using the hardware queues.

5. **Hardware synchronization:** Kernels belonging to a context can demand synchronization for executing control barriers. The CP is used to manage synchronization between the compute units allocated to each context. Compute unit synchronization is also required before the launch of each new context on the GPU.

5.5 Evaluation Methodology

5.5.1 Platform for Evaluation

To evaluate the potential of our multi-context execution scheme, we use the Multi2Sim simulation framework. The proposed design for the CML is implemented as an extension to the lower-level runtime provided by Multi2Sim. We also implement the proposed extensions to the CP on the simulated model of an AMD Radeon R9-290X GPU using Multi2Sim. The configuration of the simulated AMD Radeon R9-290X GPU is provided in Table 6.1. The TLB latencies are set to 500 cycles for accesses to the L1 TLB and 550 cycles for accesses to the L2 TLB - these values are based on measurements on an actual R9-290X GPU and take into consideration future design projections.

5.5.2 Applications for Evaluation

To properly evaluate multi-context execution, we have elected to use multi-kernel applications from the Rodinia [32], Parboil [136] and AMD SDK [1] benchmark suites. This set of applications are described in Table 5.2. Note that we include multi-kernel applications from these suites, except for 4 multi-kernel benchmarks from Rodinia which are not currently supported by the simulation framework.

To simulate multiple contexts, we execute multiple benchmarks concurrently. The characteristics of one context can affect the performance and resources available to the other contexts. If all contexts executing concurrently are memory bound, then memory accesses and memory resources
CHAPTER 5. MULTI-CONTEXT EXECUTION

<table>
<thead>
<tr>
<th>Device Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td># of CU’s</td>
</tr>
<tr>
<td># of Hardware Queues (ACE units)</td>
</tr>
<tr>
<td># of SIMD Units / CU</td>
</tr>
<tr>
<td># of lanes / SIMD</td>
</tr>
<tr>
<td># of vector reg / CU</td>
</tr>
<tr>
<td># of scalar reg / CU</td>
</tr>
<tr>
<td>Frequency</td>
</tr>
<tr>
<td>L1 (1 /CU)</td>
</tr>
<tr>
<td># of shared L2</td>
</tr>
<tr>
<td>L2 Size</td>
</tr>
<tr>
<td>Global Memory</td>
</tr>
<tr>
<td>Local Memory / CU</td>
</tr>
</tbody>
</table>

Table 5.1: Device configuration of AMD R9-290X GPU [13].

become a bottleneck for execution. This may lead to degradation in the execution performance for all of the contexts. We consider the impact of pairing application contexts that possess a range of execution characteristics.

Table 5.3 lists four workload mixes formed out of the individual benchmarks from Table 5.2. Our goal here is to produce concurrent multi-context workloads that possess different computational characteristics, including memory-bound execution, compute-bound execution and balanced execution. We will refer to these compound workloads as the Characteristic-based mixes.

We also evaluate the behavior of multi-context execution by scaling the number of concurrently executing contexts using a different set of application mixes given in Table 5.4. Each mix comprises a distinct number of concurrently executing contexts, referred to as the Sharing-Level. Each context requests compute resources based on the initial sharing allocation. This allows us to evaluate the concurrent benchmarks with varied resource allocations. The workload mixes included in this set are referred to as the Sharing-allocation-based mixes.

5.6 Evaluation Results

5.6.1 Speedup and Utilization with Multiple Contexts

We have evaluated the execution performance of the context mixes described in Section 5.5. We report on the efficiency of multi-context execution by using sequential execution of contexts as our baseline. In sequential execution, the individual contexts from different mixes execute on the GPU with 100% of the compute resources assigned to each context. The sequential
CHAPTER 5. MULTI-CONTEXT EXECUTION

<table>
<thead>
<tr>
<th>#</th>
<th>Applications</th>
<th>Suite</th>
<th>Datasets</th>
<th># Kernels</th>
<th># Wrk Grps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Stream Cluster(SC)</td>
<td>Rodinia</td>
<td>1048576 points, 512 dimensions</td>
<td>2</td>
<td>2048</td>
</tr>
<tr>
<td>2</td>
<td>HotSpot(HS)</td>
<td>Rodinia</td>
<td>1000 X 1000 data points</td>
<td>2</td>
<td>1024</td>
</tr>
<tr>
<td>3</td>
<td>Back Propagation(BP)</td>
<td>Rodinia</td>
<td>1048576 points</td>
<td>2</td>
<td>4096</td>
</tr>
<tr>
<td>4</td>
<td>Leukocyte Tracking(LC)</td>
<td>Rodinia</td>
<td>486 X 720 pixels/frame</td>
<td>3</td>
<td>3600</td>
</tr>
<tr>
<td>5</td>
<td>LU Decomposition (LUD)</td>
<td>Rodinia</td>
<td>1024 X 1024 data points</td>
<td>3</td>
<td>2048</td>
</tr>
<tr>
<td>6</td>
<td>Similarity Score(SS)</td>
<td>Rodinia</td>
<td>1024 points, 128 features</td>
<td>2</td>
<td>1024</td>
</tr>
<tr>
<td>7</td>
<td>CFD Solver (CFD)</td>
<td>Rodinia</td>
<td>125K elements</td>
<td></td>
<td>500</td>
</tr>
<tr>
<td>8</td>
<td>Kmeans Clustering(KM)</td>
<td>AMD SDK</td>
<td>204800 points, 48 features</td>
<td>2</td>
<td>1600</td>
</tr>
<tr>
<td>9</td>
<td>Gaussian Elimination (GE)</td>
<td>AMD SDK</td>
<td>1024 x 1024 data points</td>
<td>2</td>
<td>2048</td>
</tr>
<tr>
<td>10</td>
<td>Mri-Q calculation (MRI)</td>
<td>Parboil</td>
<td>64 x 64 X 64 data points</td>
<td>2</td>
<td>2048</td>
</tr>
<tr>
<td>11</td>
<td>Breadth-First Search(BFS)</td>
<td>Parboil</td>
<td>1000000 nodes</td>
<td>2</td>
<td>1952</td>
</tr>
</tbody>
</table>

Table 5.2: Summary of benchmarks used for evaluation.

<table>
<thead>
<tr>
<th>Application Mixes</th>
<th>Overall Characteristic</th>
<th>Benchmarks (Characteristic)</th>
<th>Resources Allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHR-1</td>
<td>Memory Bound (Worst Case)</td>
<td>BFS (Memory)</td>
<td>50.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SS (Memory)</td>
<td>25.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BP (Memory)</td>
<td>25.00%</td>
</tr>
<tr>
<td>CHR-2</td>
<td>Compute Bound (Worst Case)</td>
<td>SC (Compute)</td>
<td>50.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LC (Compute)</td>
<td>25.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LUD (Compute)</td>
<td>25.00%</td>
</tr>
<tr>
<td>CHR-3</td>
<td>Memory + Compute (Compute Heavy)</td>
<td>BFS (Memory)</td>
<td>50.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SC (Compute)</td>
<td>25.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>KM (Compute)</td>
<td>25.00%</td>
</tr>
<tr>
<td>CHR-4</td>
<td>Memory + Compute (Memory Heavy)</td>
<td>GE (Compute)</td>
<td>50.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BFS (Memory)</td>
<td>25.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SS (Memory)</td>
<td>25.00%</td>
</tr>
</tbody>
</table>

Table 5.3: Workload mix characteristics and with varied compute characteristics.
CHAPTER 5. MULTI-CONTEXT EXECUTION

<table>
<thead>
<tr>
<th>Application Mixes</th>
<th>Sharing Level</th>
<th>Benchmarks</th>
<th>Resources Allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHR-1</td>
<td>2</td>
<td>BFS</td>
<td>50.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SC</td>
<td>50.00%</td>
</tr>
<tr>
<td>SHR-2</td>
<td>2</td>
<td>SC</td>
<td>75.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CFD</td>
<td>25.00%</td>
</tr>
<tr>
<td>SHR-3</td>
<td>3</td>
<td>LUD</td>
<td>50.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HS</td>
<td>25.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BP</td>
<td>25.00%</td>
</tr>
<tr>
<td>SHR-4</td>
<td>4</td>
<td>SS</td>
<td>25.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BFS</td>
<td>25.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MRI</td>
<td>25.00%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LC</td>
<td>25.00%</td>
</tr>
</tbody>
</table>

Table 5.4: Workload mix with varied sharing levels.

execution permits the concurrent execution of kernels belonging to the same context. The overall speedup and increase in utilization of compute resources, such as compute units and hardware queues used in multi-context execution, are presented in Figure 5.6.

Figure 5.6: Speedup and resource utilization improvement using multi-context execution versus sequential execution of contexts.

The evaluated context mixes show an effective speedup when executed concurrently. The sharing-allocation-mixes (SHR-1 to SHR-4) and the characteristic-mixes (CHR-1 to CHR-4) show an average speedup of 1.53X and 1.61X, respectively, when using multi-context execution. These improvements are attributed to better utilization of hardware resources and overlap of time-consuming communications. The average improvement in compute unit utilization for all the con-
text mixes is 18%. The BFS, SC, BP and SS workloads have low compute unit utilization when executed individually. The mixes SHR-1, SHR-2, CHR-1, and CHR-4 formed by combining these workloads obtain the greatest benefit from concurrency, resulting in much higher compute unit utilization.

The hardware queues manage the mapping of kernels to the compute units for a particular context. Hardware queues are assigned to each context based on their sharing allocation. For sequential execution, many hardware queues on the GPU remain unused. But when contexts execute concurrently, the total number of active hardware queues increases. This improves the hardware queue utilization. The average improvement in utilization of hardware queues is 19%.

The overall speedup decreases as we increase the number of concurrent contexts. Due to increased memory pressure and a limited number of compute units (CUs), we see an increase in the execution time of individual contexts in the mix. The SHR-1 and SHR-2 mixes have only 2 contexts competing for resources, and so experience better speedups as compared to other shared-allocation-mixes. The SHR-4 mix executes 4 concurrent contexts and achieves only a 1.25X speedup.

The characteristic-mixes use 3 contexts in each mix. The CHR-1 and CHR-4 mixes combine memory-bound workloads, which experience very little benefit from concurrency (i.e., their performance is similar to sequential execution). The memory-bound contexts increase the number of in-flight memory accesses. This results in a memory bottleneck in the individual contexts, leading to slower execution of each context. The compute-bound mixes such as CHR-2 and CHR-3 show an impressive speedup as compared to the other mixes. The compute unit remapping mechanism aids the compute-bound applications to effectively reclaim unused resources, further accelerating the associated context. Figure 5.7 details the improvement in CU utilization on a per mix basis.
CHAPTER 5. MULTI-CONTEXT EXECUTION

Figure 5.8 shows an increase in execution time for individual contexts for multi-context execution. The memory-bound mixes show a 36% increase in execution time due to memory bottle-necks. The compute-bound and balanced mixes experience an average increase of 34% in execution time, due to the smaller number of compute resources available for each individual context. Even though we observe a slowdown in individual context execution, we achieve an average speedup of 1.56X for concurrent execution of these contexts in the mixes. Dynamic compute unit remapping and the TMM mechanism using the runtime and CP, attribute to the overall speedup.

5.6.2 Evaluation of Global Memory Efficiency

Figure 5.9: Global memory bandwidth utilization using multi-context execution.

Executing multiple contexts concurrently on the GPU leads to an increase in the overall number of global memory accesses. This can lead to improvements in global memory bandwidth
CHAPTER 5. MULTI-CONTEXT EXECUTION

utilization. The global memory bandwidth utilization of contexts for individual execution is low, but increases when combined with other contexts for concurrent execution. Figure 5.9 shows the global memory bandwidth utilization when using multi-context execution. The sharing-allocation mixes show an increase in their bandwidth utilization as the number of concurrent contexts increases. The mixes with fewer concurrent contexts (e.g., SHR-1 and SHR-2) and the compute-bound mixes (e.g., CHR-2 and CHR-3) show a 17.8% improvement in global memory bandwidth utilization over sequential execution. The memory-bound mixes such as CHR-1 and CHR-4 show an significant improvement of 22% in global bandwidth utilization for multi-context execution. The average increase in the global memory bandwidth utilization for multi-context execution over sequential execution is 18.1%.

5.6.3 TMM evaluation

![Timeline of global memory usage for contexts from different mixes](image)

Figure 5.10: Timeline of global memory usage for contexts from different mixes.
Figure 5.10 shows the timeline-based usage of global memory by contexts belonging to different mixes for multi-context execution. Figure 5.10 also shows the operation of the TMM mechanism (described in Section 5.3.1), which manages data buffer transfers. The impact of the TMM mechanism can be clearly seen in the BFS context present in SHR-1 and SHR-4 mixes. BFS is initially allocated 25% resources in SHR-4 and uses 27% (max) of global memory. The same BFS application uses 44% (max) global memory when executed in SHR-1 mix with 50% resource allocation and more free memory. In the case of SHR-1 and SHR-2, the TMM does not need to transfer the cold data buffers out of the GPU memory since there is enough free memory available. We do see frequent data buffer movement in the SHR-3 and SHR-4 mixes since we have to manage four concurrent contexts, which stress the global memory usage. This kind of frequent movement is not present in SHR-1 and SHR-2 since they have fewer concurrent contexts and sufficient free memory. The TMM mechanism in the CML manages global memory and data buffer transfers efficiently, even when working with a multiple concurrent contexts. The transfers handled by the TMM mechanism are overlapped with computation on the GPU and do not add any significant data-transfer delay.

We compare the performance of the TMM against previously proposed memory management approaches. We model a software-managed memory transfer approach implemented in ADSM [49]. A software-managed approach performs data buffer state (active, inactive, hot, cold) analysis and makes transfer decisions in the runtime on the host side. We also compare against a mechanism which uses asynchronous pinned-memory based data transfers that enables GPU virtualization [87]. The transfer decisions for the pinned approach are also implemented on the host side. Figure 5.11a shows the overall speedup obtained by the 3 mechanisms over the sequential context execution. The benefits of a software approach decrease as the number of concurrent contexts increases. The pinned-memory approach struggles to achieve any significant speedup over sequential execution for multiple concurrent contexts. TMM shows an improvement in speedup by 22% and 40.7% over the software-managed and pinned-memory approaches, respectively. The software approach communicates each transfer decision to the GPU over the PCIe bus before initiating the transfer. It also issues queries to the GPU regarding the state of buffers and contexts. The communication delay adds latency to the software-managed mechanism. The pinned-memory approach introduces high overhead due to increased PCIe traffic and decision communication delay, especially as we increase the number of concurrent contexts. Figure 5.11b compares the number of queries from host to the GPU for each management mechanism. TMM helps in reducing the number of query traffic by initiating transfer control using CP. A CP with good performance is essential for efficient decision control and transfer initialization, reducing the overhead incurred due processing.
CHAPTER 5. MULTI-CONTEXT EXECUTION

Figure 5.11: Evaluation of different global memory management mechanisms. (a) Overall Speedup and (b) Number of context status and data buffer status queries from host-to-GPU.

Figure 5.12: Performance of pinned memory updates over transfer stalling (blank space indicates no pinned memory update).
CHAPTER 5. MULTI-CONTEXT EXECUTION

The TMM uses pinned memory for new buffers when GPU memory is fully utilized by hot-buffers. All the updates to those buffers involve accesses to pinned memory. The software-managed mechanism provides an alternate approach to stall transfers when the GPU memory is full. This can result in stalled contexts, which are waiting for data buffers. Figure 5.12 shows the total number of accesses to pinned buffers as a proportion of total buffer accesses when using TMM. It also shows the relative speedup obtained by relying on pinned buffers over the transfer stalling implemented in the software-managed approach. Applications that rely on pinned memory versus stalling when GPU memory is full, perform 1.12X better compared to waiting for memory to become available. For cases such as SHR-4 and CHR-1, the global memory gets released soon after the decision of using pinned memory is taken. In these cases, the software-managed stalling approach performs better than pinned memory buffer, if the stall time is low. A side-effect of software-managed approach is the simultaneous stalling of all the executing contexts leading to a deadlock on the GPU. The pinned buffer approach of TMM helps in avoiding such deadlocks by continuing execution of the contexts.

Figure 5.13: Timeline analysis of the compute unit remapping mechanism.
5.6.4 Evaluation of Compute-Unit Remapping Mechanism

We analyze the efficiency of the compute unit (CU) transfer mechanism described in Section 5.3.2. Figure 5.13 presents a timeline-based analysis of the number of CUs controlled by each context, and clearly illustrates the CU remapping mechanism. All contexts are assigned the number of CUs according to their specified sharing-allocation. The SC context is allotted 33 CUs in the SHR-2 mix and is allotted 22 CUs in the CHR-2 mix, as seen in Figure 5.13. Thus, the execution time for SC in the CHR-2 mix is 21% longer than in the SHR-2 mix. The CP remaps the mappable CUs to other contexts which have exhausted their allocated CUs and have additional work to complete. The reclamation procedure of CUs can also be observed for most of the contexts in mixes SHR-2, CHR-2 and CHR-3. The contexts in SHR-4 benefit from the early completion of the SS context, which allows for aggressive remapping of CUs. The remapping mechanism is critical for application throughput and must be managed using an efficient CP.

5.6.5 L2 cache Efficiency for Multi-Context Execution

![Figure 5.14: L2 cache hit rate for multi-context execution.](image)

The address space isolation mechanism proposed in Section 5.2.2 allows for the effective sharing of an L2 cache between concurrently executing contexts. The contexts using large portions of L2 cache memory may suffer due to interference from other concurrent contexts. This may result in lower cache hit rates and a slowdown of the context. The L2 cache hit rates for the contexts belonging to different mixes are shown in Figure 5.14. The contexts in memory bound mixes (CHR-1 and CHR4) exhibit thrashing in the L2 cache. Thrashing leads to the eviction of cache blocks by another context in the mix. The compute-bound mixes (CHR-2 and CHR-3) have hit rates comparable to sequential execution. The contexts from these mixes do not stress the cache and show efficient cache sharing for multi-context execution. Sharing-allocation mixes show the impact of scaling the number of concurrent contexts on the hit rate of the L2 cache. The reduction in hit rate for SHR-4...
CHAPTER 5. MULTI-CONTEXT EXECUTION

(with 4 concurrent contexts) and SHR-1 (2 concurrent contexts) when compared to their sequential execution is 25% and 8%, respectively. The effective hit rate falls as the number of concurrent contexts increases.

Figure 5.15: Conflict miss in L2 cache as % of total misses.

Figure 5.16: TLB hit rate for multi-context execution.

Figure 5.15 shows the number of conflict misses occurring in the L2 cache, where blocks of one context are evicting blocks of another context. For the memory-bound mixes, the conflict misses make up a majority of the total L2 cache misses (avg. 51%). For the compute-bound mixes, we see that L2 conflicts are less of an issue (avg. 27%). The number of conflict misses is higher when concurrently executing 4 contexts (avg. 49% in SHR-4), versus SHR-1 (avg. 29%) which executes 2 concurrent contexts. Figure 5.16 shows the L1 and L2 TLB hit rates for contexts from all of the mixes. The interference due to multi-context execution results in a reduced hit rate for individual contexts on both the L1 and L2 TLBs. The average reduction in hit rate for all contexts is only 8.4% on the L1 TLB and 6.9% on the L2 TLB for multi-context execution. The proposed mechanism for address space isolation in the L2 cache allows for efficient execution of multiple contexts. This is evident from the impressive speedup in execution performance for all the mixes using multi-context execution, with only a small degradation (less than 9%) in L2 cache hit rates for individual contexts.
5.6.6 Firmware characterization and CP selection

To ensure that the performance of the CP does not become a bottleneck, we characterize the execution of the firmware tasks needed to support multi-context execution. Figure 5.17a shows the proportion of each firmware task that runs on the CP. We observe that more than 60% of the time is spent in TMM control, kernel scheduling and resource management while handling multiple contexts. The design of the CP core must be aimed at reducing latencies incurred for executing these three tasks. Figure 5.17b shows the operation-based breakdown of the firmware tasks, averaged over all of the application mixes. The firmware tasks are fairly compute-bound, with more than 75% of the time spent on integer and float-point operations. The small percentage of memory operations in every firmware task indicates less data usage. Thus, these tasks will not require a complex processor core to hide memory latencies. We considered three designs for the CP core: (1) A Simple thin core with 5 stage pipeline comparable to the current CP designs, (2) An in-order core with 8 stage pipeline, comparable to modern ARM-like embedded processors, and (3) A complex 2-way-out-of-order core (ROB size:8, ld/st queue size : 8/8), which resembles modern low-power CPU.

<table>
<thead>
<tr>
<th>Core Type</th>
<th>Simple</th>
<th>In-order</th>
<th>2-way-OoO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology(nm)</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>Core Area(mm²)</td>
<td>1.678</td>
<td>1.075</td>
<td>9.84</td>
</tr>
<tr>
<td>Peak Power(W)</td>
<td>1.169</td>
<td>0.1591</td>
<td>26.114</td>
</tr>
<tr>
<td>Subthreshold Leakage(W)</td>
<td>0.796</td>
<td>0.059</td>
<td>4.57</td>
</tr>
<tr>
<td>Area of AMD R9-290X GPU(mm²)</td>
<td>438</td>
<td>438</td>
<td>438</td>
</tr>
<tr>
<td>% increase in GPU area due to CP</td>
<td>0.38</td>
<td>0.25</td>
<td>2.25</td>
</tr>
</tbody>
</table>

Table 5.5: Area and power overheads of the CP alternatives.
CHAPTER 5. MULTI-CONTEXT EXECUTION

Figure 5.18: Execution performance of firmware on CP cores.

Figure 5.18 shows the firmware execution performance of the cores as normalized to the simple core. We observe that the in-order core shows an 1.21X increase in performance over the simple core, whereas the out-of-order shows 1.28X improvement in performance over the simple core. Table 5.5 shows the peak power and area of the CP core configurations calculated using the McPAT power and area modeling tool, integrated with Multi2Sim [85, 148]. The ARM-like in-order core is the most efficient design considering the execution performance, peak power, and die area. The in-order core provides a good balance between performance and core complexity and is used for the evaluation presented in previous sections.

5.6.7 Overhead in Multi-Context Execution

Figure 5.19: Context handling delay added by the CP.

The CP controls the context creation and deletion on the GPU. For every context dispatched to the GPU, the CP performs initialization steps such as resource checking, resource allocation, and setting up global memory and page tables. This initialization adds execution overhead for each context. Figure 5.19 shows the context creation and context deletion delay as a percentage of
CHAPTER 5. MULTI-CONTEXT EXECUTION

The total execution time for each context across all evaluated mixes. The same application contexts, when executed using different mixes and sharing-levels, exhibit variations in context creation and deletion delay. The change in sharing-allocation results in a change in execution time of the context and impacts the context management overhead. We observe an average context creation delay of 3.63% and an average context deletion delay of 1.79% of total execution time.

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>L1 TLB</th>
<th>L2 TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB Entries</td>
<td>2048</td>
<td>8192</td>
</tr>
<tr>
<td>Tags (52 bit VPN * L1/L2 entries)</td>
<td>13KB</td>
<td>52KB</td>
</tr>
<tr>
<td>Total CID overhead (2bit/entry * L1/L2 entries)</td>
<td>512B</td>
<td>2KB</td>
</tr>
<tr>
<td>Total TLB Area (20 Bit PPN * L1/L2 entries + Tag)</td>
<td>18KB</td>
<td>72KB</td>
</tr>
<tr>
<td>% increase in TLB area L1:(512B/18KB), L2:(2KB/72KB)</td>
<td>2.77</td>
<td>2.77</td>
</tr>
</tbody>
</table>

Table 5.7: TLB area overhead.

The context-id (CID) bits add storage overhead to the L2 cache tags, and TLB tags. Table 5.6 and Table 5.7 detail the storage overhead due to the additional CID bits in the L2 cache and TLB, respectively. The CID bits in the L2 cache add 4KB of storage overhead (less than 0.4% of the area of baseline 1MB L2 cache). The CID tag adds less than 3% storage overhead for L1 TLB (excess 512B) and L2 TLB (excess 2KB). The CID registers for CUs and page table registers add a 120B storage overhead. The low additional overhead for the L2 cache, TLB, and control registers provide for a cost-effective and performance-effective address space isolation mechanism using the CID.
5.7 Discussion

5.7.1 Static L2 Cache Partitioning for Multiple Contexts

Our proposed mechanism for address space isolation using a context-id (CID) is discussed in Section 5.2.2. Another possible scheme for address space isolation is to use static cache partitioning. Such schemes have been discussed in previous work on cache management on GPUs [83]. The L2 cache on the AMD R9-290X GPU is split across 8 memory-controllers [13]. We can implement static partitioning by assigning L2 units to different contexts. Figure 5.20 shows the hit rates for static partitioning and our CID-based cache design. Static partitioning of the L2 cache provided limited benefits in terms of the cache hit rates for multi-context execution. Static partitioning restricts the contexts to use only limited cache resources, and does not tolerate large variations in data access patterns which can lead to thrashing in the L2 cache. In our approach, the entire L2 cache is visible and accessible to all contexts by using their unique CIDs, and can adapt for contexts with varying data requirements. To summarize, in any combination of contexts, a context can exhibit L2 cache thrashing using static cache partitioning, if that context uses more than \( P\% \) (where \( P \) is the sharing-allocation, e.g. 25% or 50%) of the cache space in a CID based-cache.

5.8 Summary

In this chapter, we proposed a hardware/software approach to support multi-context execution on a GPU. Our design increases the utilization of the GPU resources, which can significantly benefit workloads targeted for data centers and cloud engines. We describe new architectural and runtime enhancements needed to support multi-context execution on a GPU. We add a context-management-layer (CML) in the runtime to handle memory transfers for contexts and manage the
CHAPTER 5. MULTI-CONTEXT EXECUTION

dispatch of contexts to the GPU. We also describe a new *Transparent Memory Management* (TMM) facility and evaluate the benefits of this combined host/GPU-based control mechanism versus other host-side approaches. Similarly, we enable the GPU to handle complex firmware tasks using an upgraded command processor (CP). We also implement address space isolation in shared L2 cache and TLBs with less than 0.4% increase in the area for caches.
Chapter 6

Application Behavior-Aware Resource Management for Multi-tasked GPUs

Popular compute environments, such as mobile SoCs (Systems-on-Chip), datacenters, and cloud engines, allow the use of GPUs for offloading computations. The growing demand for these compute platforms enhances the need for GPUs that can support execution of multiple applications that concurrently execute on the same device.

We have presented our design to implement multi-tasking on GPUs in Chapters 4 and 5. Our approach provides a mechanism to seamlessly host multiple application contexts on the same GPU. The adaptive spatial partitioning mechanism allows for each of these contexts to execute multiple concurrent kernels, thereby enabling multi-level concurrency. While our approach for multi-tasking is efficient, it may also introduce further challenges, such as added contention on multiple hardware resources. GPUs consist of many compute units (CUs), which are be shared among multiple applications. The contention for CUs may affect the amount of parallelism that can be achieved by an application. Changing the number of CUs allocated to an application can have a significant impact on the resulting execution performance. Another source for contention on a multi-tasked GPU is the added pressure on the shared memory hierarchy. In modern GPU designs, all CUs share the L2 cache, the L3 cache (if present) and the global DRAM memory. The overall performance of a multi-tasked GPU is observed to degrade when the working sets of co-executing applications exceed the L2/L3 cache sizes or the global memory size [144]. Similarly, applications executing on a multi-tasked GPU can present disparate compute behaviors. The performance of these applications may be dependent on more than one resource. We enhance our multi-tasking framework on the GPU to reconsider the existing resource allocation for applications, and make changes to reduce resource contention and improve throughput. In addition, the co-executing applications may
have different priority levels or may require strict Quality-of-Service (QoS) guarantees (e.g., graphics applications). The QoS requirements can be service level agreements in the form of min/max performance or a dedicated resource allocation.

Our design described in previous chapters primarily focuses on managing the contention across CUs, and provides limited adaptability to match the compute behavior of application. In this chapter, we enhance our current design by introducing Virtuoso – a dynamic resource management framework to provide contention free, QoS-aware, and application behavior-aware allocation of compute and memory resources to applications on a multi-tasked GPU. Virtuoso is implemented by modifying software runtimes and hardware-level architecture of the GPU. Virtuoso provides priority-based resource allocation mechanisms to achieve QoS targets for applications. The QoS targets for applications are set using QoS policies. A wide range of QoS policies are supported, including performance targets, throughput targets, or resource utilization targets. At the software-level, Virtuoso provides an interface which allows the host OS to define QoS policies that match the requirements of the system.

The rest of the chapter is organized as follows. Section 6.1 describes the need for QoS aware resource management on multi-tasked GPUs. The Section 6.2 describes the structure of the QoS policies for GPUs, and the metrics used for measuring the QoS. We then describe the software and hardware based architecture of Virtuoso in Section 6.3. The Section 6.4 describes our evaluation methodology, while Section 6.5 presents the evaluation of our Virtuoso framework across multiple application mixes and QoS policies.

6.1 Need and Opportunity for QoS

Fig. 6.1 presents an example to illustrate the need for QoS on GPU platforms that implement spatial multi-tasking. The graph compares the execution performance (IPC), and system throughput (STP) of applications executing in multi-tasked mode against dedicated (DED) mode execution. STP is calculated as suggested by Eyerman et al. [45]. STP measures the overall system performance and expresses the amount of work done during a unit of time. In this example, we execute 20 distinct pairs of compute-bound (COM) and memory-bound (MEM) applications selected from Table 5.2 on a detailed GPU simulator supporting spatial multi-tasking [148]. The system throughput of the multi-tasked GPU improves on average by 25.7% when applications execute simultaneously. But this improvement in overall throughput comes at the cost of the performance (IPC) of each individual application. The COM and MEM applications show an average 34% and 29% reduction in IPC over their DED performance, respectively. To improve the performance of
CHAPTER 6. APPLICATION-AWARE RESOURCE MANAGEMENT FOR GPUS

Figure 6.1: System throughput improves, but at the cost of application performance for co-executing applications on multi-tasked GPUs. This illustrates the need to support applications with QoS requirements when sharing GPUs.

Figure 6.2: Large improvements in IPC obtained with small variation in STP, for co-executing COM/MEM application pairs by (a) varying the number of allocated CUs for the COM application, and (b) varying the L2 cache allocation for MEM applications. This demonstrates the opportunity for achieving QoS through reallocation of resources. The geomean of IPC and System Throughput (STP) is provided.

applications executing on shared GPUs, we need to develop robust policies which can enforce QoS guarantees for certain high-priority applications. This goal can be achieved by improving the IPC of high-priority applications, while maintaining a near-constant system throughput.

We now explore the opportunity to achieve QoS guarantees for applications on a multi-tasked GPU through resource scaling. We observe the performance of the 20 COM/MEM application pairs by varying the number of resources allocated to an application. Each application in the pair is assigned a distinct priority level. We scale the number of compute units and L2 cache space allocated to the high-priority application in the pair. The evaluation is done for both cases: (1)
CHAPTER 6. APPLICATION-AWARE RESOURCE MANAGEMENT FOR GPUS

COM has higher priority (CUs varied), and (2) when MEM has higher priority (L2 cache space is varied), for each of the pairs. When one resource is varied, the other resource is distributed equally across the applications. As observed in Fig.6.2, when a compute-bound application (COM) has high priority, the IPC can be improved by 14x over an equal distribution, experiencing less than a 5% variation in the STP. Moreover, the memory-bound applications are sensitive to variations in L2 cache space allocation and can be improved by 6x over an equal distribution, experiencing only a 3% variation in STP. We can clearly see an opportunity to improve the individual performance (IPC) of each application through resource scaling, with little impact on overall system throughput. This behavior can be exploited to achieve QoS targets for various types of applications (COM, MEM) on multi-tasked GPUs. This motivates the need for a framework which can achieve the QoS targets of prioritized applications through allocation of multiple compute and memory resources.

6.2 Virtuoso Defined QoS

6.2.1 QoS policies

QoS policies for a multi-tasked GPU can vary greatly, based on the requirements of the host environment (e.g., SoCs or cloud engines). The host OS in such environments should be able to implement a wide range of QoS policies for the GPU. The policy should be able to support the QoS targets for multiple priority levels. The selection of QoS targets has to be flexible and must allow for resource utilization targets, and execution performance targets. The host OS should also have the ability to control the system throughput, along with QoS targets, for applications. The system should further be able to define QoS policies for time-sensitive applications such as graphics applications. Virtuoso provides a flexible microarchitecture-agnostic interface which allows the host OS to define the QoS policy, along with the control parameters which can suit the needs of the system. This interface is described in Section 6.3.

A range of policies can be defined by specifying targets (resource or performance) for different priorities using the Virtuoso interface. To evaluate the performance and robustness of Virtuoso, we implement three priority-based QoS policies. We specify the QoS targets for the applications in terms of performance level constraints. The performance constraint is defined as the level below which the performance of an application must not degrade when compared to its dedicated mode execution (DED). The policies used in our design for dynamic QoS are:

(a) Low Priority Constraint – a goal that any/all low-priority applications should not degrade below a certain level of baseline performance.

(b) High Priority Constraint – a performance goal for high-priority applications relative to their
DED performance, while ignoring the low-priority constraint.

(c) Overall Constraint – a threshold for the system throughput (STP) of the multi-tasked GPU is defined. There might be cases where the degradation of a low-priority application far outweighs the performance improvement from a high-priority application. In such scenarios, maintaining a constant STP is beneficial for multi-tasked GPUs.

6.2.2 QoS Metrics

Multi-tasked GPUs have to efficiently manage compute and memory resources to achieve the QoS targets. For our Virtuoso implementation, we focus on managing the allocation of compute units (CUs), L2 cache, DRAM cache and off-chip DRAM memory. The utilization of compute resources is measured using the number of CUs allocated to the applications. The memory resource utilization considers the L2 cache space, and the DRAM cache space used by each application. For global memory utilization, we consider the total usage of off-chip DRAM memory.

As described in Section 6.2.1, the QoS targets may not only be related to resource utilization, but also with execution performance. In such cases, resource usage of an application is not sufficient to measure QoS and make allocation decisions because allocating additional resources may not necessarily improve the performance of an application. For example, allocating more memory to a compute-bound application may not impact its performance at all. Therefore, we use performance metrics such as Instructions Per Cycle (IPC) to measure CU allocation efficiency, Misses Per Interval (MPI) for L2 cache effectiveness, and global memory throughput (GMT) in kilobytes/cycle for the DRAM cache performance. System throughput (STP) measures the overall progress of the system with multiple co-executing applications.

6.3 Virtuoso Architecture

In this section, we describe the implementation of the Virtuoso architecture. Virtuoso is a layered architecture, consisting of three primary layers: (i) QoS interface, (ii) QoS assignment, and (iii) QoS enforcement. Virtuoso is integrated into the GPU architecture, as shown in Fig. 6.3.

6.3.1 QoS Interface: Settings Register (QSR)

The QoS interface layer is used to initialize the QoS hardware on the GPU. The layer consists of a memory-mapped QoS settings register (QSR) which provides an interface between the host OS and the GPU. The OS can set the QSR to indicate the number of applications executing on the GPU, the QoS policy type (i.e., High Priority Constraint, Overall Constraint, or Low Priority Constraint).
CHAPTER 6. APPLICATION-AWARE RESOURCE MANAGEMENT FOR GPUS

Figure 6.3: Virtuoso, as implemented in a current GPU architecture (left). The three layers of the Virtuoso implementation (right). The interface layer provides an interface to the host OS, the assignment layer allocates resources and the enforcement layer implements QoS-aware resource allocation/reallocation.

<table>
<thead>
<tr>
<th>(a) QoS Settings Register (QSR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Apps (4 bits)</td>
</tr>
<tr>
<td>-------------------------------</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Priority Level</th>
<th>Resource Type</th>
<th>Performance Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits to indicate priority, -3 levels of priority: 00: Low 01: Medium 10: High</td>
<td>24 bits for the resource usage - 8 bits for Compute Units (CU) - 8 bits for L2 cache usage - 8 bits DRAM cache usage</td>
<td>50 bits to capture the performance metrics - 2 bits for QoS type (Rsrc alloc./Constraint) - 16 bits for target IPC and current interval (8 bits each) - 16 bits for MPI (Miss per instruction) - 16 bits for DRAM cache throughput</td>
</tr>
</tbody>
</table>

Figure 6.4: Information stored in the (a) QoS setting register and the (b) QoS information table.

Constraint) and the QoS targets (i.e., objectives and constraints). For example, the high-priority constraint policy can be set with a constraint value of 20%. This would direct Virtuoso to reallocate resources in such a way that the performance of the high-priority application is only 20% less than its dedicated (DED) execution. A resource allocation objective can also be set if an objective-based QoS policy is selected. The QSR register contents are shown in Fig. 6.4a.

The dynamic QoS policy implemented by Virtuoso uses three priority levels (High, Medium, Low). For effective management of resources across these priorities, we consider two additional parameters: (a) the partition level and (b) the split ratio. All applications that are assigned a priority lower than the partition level will be candidates to release resources. The split ratio dictates the ratio in which the released resources will be distributed across the higher priority
applications. For our evaluation, we set the partition level to medium priority (i.e., resources are only released by low-priority applications). Both the partition level and the split ratio can be modified by host OS by writing to the QSR. This protocol provides the flexibility to implement multiple levels of priority without any changes to the hardware.

### 6.3.2 QoS Assignment: Performance Analyzer

Once the QSR is initialized, the next step is to determine how to adjust the resources assigned to every application. The QoS assignment layer implements a QoS information table (QIT), which stores the QoS information for each executing application. There is one table entry for each application in the QIT. Dynamically monitored system performance metrics are also stored in the QIT. The contents of a QIT table entry are shown in Fig. 6.4b.

#### Algorithm 3 Performance Analyzer

```plaintext
1: for all Co-Executing Applications do
2:     // Calculate error between current interval and target for each metric
3:     IPC_{err} = (IPC_{targ} - IPC_{curr}) / IPC_{targ}
4:     MPI_{err} = (MPI_{targ} - MPI_{curr}) / MPI_{targ}
5:     GMT_{err} = (GMT_{targ} - GMT_{curr}) / GMT_{targ}
6:     // Sort the metric errors in descending order
7:     Sort(IPC_{err}, MPI_{err}, GMT_{err})
8:     AssignScore(IPC_{err}, MPI_{err}, GMT_{err})
10: end for
```

The performance metrics for each application are recorded in the QIT using GPU performance monitors. This recording is done periodically using a driver-defined QoS interval. The performance analyzer module decides the resources to be assigned to each application in the next QoS interval using Alg. 3. The algorithm calculates the normalized error between the target value and the current interval value for each metric (line 3–5). It sorts the error values in descending order using a merge-sort operation (line 8). The analyzer then assigns a score to each metric according to their position in the order. This process is repeated for each of the co-executing applications. As described in Section 6.2.2, each metric is associated with a particular provisioned resource. The sorted order of performance metrics represents the order in which resources which have to be provisioned for the next QoS interval for each application. For \( n \) co-executing applications and \( m \) provisioned resources, the algorithm shows a complexity of \( O(nm\log(m)) \). As the number of provisioned resources for any GPU remains constant, the time complexity becomes linear \( O(n) \). This analyzer uses the command processor on the GPU for calculations. The command processor executes the
CHAPTER 6. APPLICATION-AWARE RESOURCE MANAGEMENT FOR GPUs

6.3.3 QoS Enforcement: Resource Manager

The resource manager represents each resource in the form of a distinct token (or share) \([152]\). The token for each resource represents the minimum amount of resource which can be assigned to an application (i.e., the number of CUs, the amount of L2 cache, and the amount of DRAM cache). The applications below the partition-level (defined by the QoS policy) release tokens for each resource. These tokens can be reallocated to applications running above the defined partition-level. The relative performance impact of releasing tokens for a particular resource is based on the information from the performance analyzer. The following steps are implemented by the resource manager for dynamic allocation of resources:

1. A score for each resource of a high-priority application is used to calculate the number of tokens...
CHAPTER 6. APPLICATION-AWARE RESOURCE MANAGEMENT FOR GPUS

to be released by the low-priority application (See Fig. 6.5).

2. The number of released tokens for each resource is stored in the Token Release Register (TRR).

3. The contents of the TRR are used to modify the token count for each application in the resource table. Modifications are done according to the \textit{split ratio} defined through the host OS.

4. Tokenizers start the allocation mechanism by converting the token amount for each application to the appropriate level for each resource.

5. The reallocation steps are repeated for every QoS interval.

\textit{Virtuoso} resets this dynamic allocation process whenever a new high-priority application arrives, or an old high-priority application completes execution.

Next, we describe the allocation mechanism for each of the controlled resources.

\subsection*{6.3.3.1 Compute Unit Reassignment:}

The number of CU tokens assigned to each application in the resource token table represents the CU budget of each application. The CU tokenizer translates the number of tokens to the number of CUs using a token-to-CU mapping. An increase in the token count relates to an increase in CUs assigned to an application. Similarly, a decrease in the token count corresponds to a decrease in the number of CUs. The tokenizer triggers the preemption mechanism to initiate a reassignment of CUs to the applications. The CU allocation for higher priority applications stops if the required QoS objective is achieved, or if the application has no pending work. The token-based allocation is valid for all priority levels except for \textit{time-sensitive} applications. In such cases, the preemption mechanism triggers a device-wide context-switch, which saves the execution state of all running applications and hands over all CUs and cache space to the \textit{time-sensitive} application. Upon completion, the execution state and the resource allocation of the preempted applications is restored.

\subsection*{6.3.3.2 Partitioning L2 cache and DRAM cache:}

The L2 cache and DRAM cache utilization of each application is monitored for each QoS interval. The partitioning of the caches is established by imposing a cache allocation threshold on the application. A token-based partitioning for cache defines the granularity at which the resource quantity increases/decreases for a given application. For example, one token for the L2 cache translates to 10\% of the cache space to be allocated or deallocated from an application. The QoS target is achieved by controlling the cache allocation through a modified cache replacement policy during each interval. The utilization of the cache by each priority level is assessed through performance monitoring. If the present utilization is lower than the allocation threshold, then the replacement
follows a standard LRU algorithm. When the utilization of the constrained priority level reaches a threshold, our replacement policy overrides the default LRU to ensure that it finds the victim within the same priority level. There may be cases where the set does not contain a line from the same priority level, even though the utilization threshold is reached. In such cases, the victim is selected from a lower priority application, or at random if none is available. The same modifications apply to DRAM caches, where the victim is an entire cached page within the set, and not a cache line.

6.3.3.3 Management of Multi-priority Structures:

There may exist situations where multiple applications from the same priority execute concurrently. In such cases, Virtuoso allocates an equal number of resources to each application to maintain fairness. For cases where co-executing applications are only from two priority levels, the QoS policy adapts to a two-level priority structure. For example, if one high-priority application and two medium-priority applications execute concurrently, the system changes to a two-level priority structure, and classifies the high-priority application as the partition level (applications below the partition level can release resources). In such cases, the resources are released by the medium-priority applications in every other QoS interval. This effectively halves the rate at which medium-priority applications release resources as compared to low-priority applications, and prevents the medium-priority application from behaving as a low-priority workload. Virtuoso knows the priority levels of applications since they are stored in the QSR and QIT. The decision to release resources from an application is made by setting a bit in the QIT entry for the application.

Implementing various QoS policies may lead to scenarios where low-priority applications may lose ownership of some of the Virtuoso controlled resources. In such cases, Virtuoso waits for the completion of a higher-priority application, and then reinstates the low-priority task with an upgraded priority (i.e., the low-priority task is upgraded to a medium-priority task). This mechanism prevents starvation of the low-priority application.

6.4 Evaluation Methodology

6.4.1 Platform for Evaluation

We evaluate our QoS-aware resource allocation mechanism using the same GPU model, and simulation framework as described in Chapter 5 Section 5.5. We have integrated the hardware implementation of Virtuoso into the existing simulation infrastructure. The profiling counters used for performance monitoring are similar to those provided in today’s GPU architectures. No additional performance counters were added to the GPU architecture. The baseline dedicated (DED)
CHAPTER 6. APPLICATION-AWARE RESOURCE MANAGEMENT FOR GPUS

### QoS Configuration

<table>
<thead>
<tr>
<th>QoS Settings</th>
<th># of Priorities</th>
<th>Partition Level</th>
<th>Medium Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>1 CU token</td>
<td>1 CU</td>
</tr>
</tbody>
</table>

**QoS token settings (1 token = amount of resource)**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 L2 cache token</td>
<td>5% of L2 cache space</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 DRAM cache token</td>
<td>5% DRAM cache space</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CU</td>
<td>67 : 33</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L2 cache</td>
<td>75 : 25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DRAM cache</td>
<td>75 : 25</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: QoS parameters and resource token settings used in *Virtuoso* evaluation.

Performance is predicted by applying a linear estimation of the performance metrics obtained for each application in their first two QoS intervals. Leveraging linear estimation has been used previously by Aguilera et al. [11], and is also adopted in this work. The DRAM cache model and its latencies are modeled as described in previous work by Jevdjić et al. [66]. The configuration of the simulated GPU and the QoS settings are provided in Table 6.1.

<table>
<thead>
<tr>
<th>Mix name</th>
<th>Applications</th>
<th>Priority Levels</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mix-1</td>
<td>SC</td>
<td>High</td>
<td>Compute Bound</td>
</tr>
<tr>
<td></td>
<td>LC</td>
<td>Med</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BP</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>Mix-2</td>
<td>LUD</td>
<td>High</td>
<td>Memory Bound</td>
</tr>
<tr>
<td></td>
<td>CFD</td>
<td>Med</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BFS</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>Mix-3</td>
<td>GE</td>
<td>High</td>
<td>Memory + Compute (Compute Heavy)</td>
</tr>
<tr>
<td></td>
<td>MRI</td>
<td>Med</td>
<td></td>
</tr>
<tr>
<td></td>
<td>KM</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>Mix-4</td>
<td>BP</td>
<td>High</td>
<td>Memory + Compute (Memory Heavy)</td>
</tr>
<tr>
<td></td>
<td>BFS</td>
<td>Med</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GE</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>Mix-5</td>
<td>SC</td>
<td>High</td>
<td>Compute + Memory (Balanced)</td>
</tr>
<tr>
<td></td>
<td>LUD</td>
<td>Med</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GE</td>
<td>Med</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Workload mixes used to evaluate our *Virtuoso* to support QoS goals.

### 6.4.2 Applications for Evaluation

To properly evaluate *Virtuoso*, we build application mixes using the same multi-kernel OpenCL applications as described in Chapter 5 Section 5.5 (See Table 5.2). We build application mixes such that each application is assigned a different priority level. The application mixes are formed in such a manner that each mix displays a distinct execution behavior (*memory bound* or *compute bound*). The application mixes used for our evaluation are presented in Table 6.2.
6.4.3 Choosing the QoS Reallocation Interval

An important decision for dynamic resource allocation is the choice of the time interval between two allocation decisions. Using a shorter interval may not amortize the resource reallocation latency, and can be prone to sampling anomalies in our performance counts. Similarly, a longer interval may not capture the fine-grained behavior of an application, and may not allocate enough resources required to achieve the QoS targets. The QoS interval for *Virtuoso* is chosen by considering: (1) the time taken to reallocate compute resources, and (2) the average time taken to capture the behavior of an application when using the reallocated resource. The first quantity is defined as the average time taken by an application to reassign one CU to another application ($T_{cuassgn}$). The CU preemption mechanism implemented in the simulator is used to calculate this value for each application [143]. The L2 cache and DRAM cache allocation scheme does not impose any latency, as it is controlled through the cache replacement policy. The second quantity used to determine the interval is defined as the average time taken by an application to complete work allocated to one CU. This is calculated as the time taken to execute all wavefronts (warps in CUDA) scheduled on one CU for a given application. $WF_{perCU}$ defines the average number of wavefronts on each CU, and $WF_{exec}$ defines the average execution time for each wavefront for an application. The QoS interval is calculated using Equation 6.1, which amortizes the quantities described above.

$$QoS_{interval} = \frac{1}{N} \sum_{i=0}^{N} [(T_{cuassgn})_i + (WF_{perCU})_i \times (WF_{exec})_i]$$  

(6.1)

Where $N$ is the number of applications used for calculating the QoS interval value.

The values for $WF_{perCU}$ and $WF_{exec}$ are obtained by profiling the execution of our selected application benchmarks (Table 5.2) on a physical AMD Firepro GPU. Values for $T_{cuassgn}$ and $WF_{exec}$ for all of the applications in this thesis are presented in Fig 6.7. Using Equation 6.1
and the data obtained from the profiling runs, the value for the QoS interval is calculated as $35K$ GPU cycles. This value is used as the QoS interval for further evaluations.

**Figure 6.8:** *Virtuoso* based QoS enforcement leads to sharp gains in IPC (normalized to DED) for applications with higher STP. Restricting the degradation of low priority applications leads to higher throughput with modest gains in IPC for each application.

### 6.5 *Virtuoso* Evaluation

#### 6.5.1 Evaluation of QoS Policies

We evaluate the effects of prioritization and dynamic reallocation for the QoS policies proposed in Section 6.2. The performance constraints for each policy are specified as a multiplier of their baseline *dedicated* (DED) mode execution performance (e.g., 0.9x, 0.8x etc.). Fig. 6.8 shows the performance of each application in the mix as compared to its dedicated execution performance. We observe that the *Overall Constraint* policy leads to 0.8x of DED performance for high-priority applications, even while achieving a constant STP. The resulting performance for the *Overall Constraint* policy can be attributed to the reassignment of low-priority resources to high and medium-priority applications. The low-priority applications experience a $\sim70\%$ performance degradation as compared to DED mode performance. Significant degradation of low-priority applications can be avoided using the *Low Priority Constraint* policy. The average performance of high-priority applications is lower when executing under the *Low Priority Constraint* policy, as compared to other policies. The applications from *Mix-5* do not contain a low-priority application. In such cases, *Virtuoso* uses a two-level priority approach, and the medium-priority application releases resources that are acquired by the high-priority application. The rate of resource reduction for the medium-priority application is at half the rate of the resource reduction for the low-priority applications. This approach ensures fair management of medium-priority applications in such scenarios. As observed in Fig. 6.8, regardless of the policy, the degradation experienced in medium-priority applications in Mix-5 is less than the degradation experienced for the low-priority applications.
CHAPTER 6. APPLICATION-AWARE RESOURCE MANAGEMENT FOR GPUS

Figure 6.9: Improvement seen in performance (IPC) and system throughput (STP) of applications using Virtuoso policies as compared to Naive-MT. The geomean of IPC (normalized to DED) for all priority levels is considered.

We also compare the performance of Virtuoso against a naive spatial multi-tasking (Naive-MT) strategy. The Naive-MT technique establishes spatial allocation on the basis of the number of workitems (threads) launched by each application [10]. The Naive-MT technique does not implement any QoS-aware partitioning. Fig. 6.9 illustrates the performance of applications at each priority level versus their dedicated execution (DED). The comparison is done across different Virtuoso policies and the Naive-MT technique. The aggressive reallocation of resources to high and medium-priority applications using Virtuoso policies results in enhanced performance over the Naive-MT scheme for high and medium-priority levels. The low-priority applications suffer more degradation using Virtuoso, but can be controlled by using a Low Priority Constraint policy. Overall, Virtuoso shows an average improvement of 17.8% in IPC and 37% improvement in STP over Naive-MT.

6.5.2 Impact of Compute Resource Allocation

The QoS-aware reallocation of compute units (CUs) helps high-priority applications achieve significant performance advantages, at the cost of performance degradation of low-priority applications. Virtuoso does not allow one CU to be shared by two applications simultaneously. If the split-ratio for distributing CUs across priorities is non-integral, then the number of CUs assigned to the high-priority applications is rounded up, while the number of CUs assigned to the medium-priority applications is rounded down. We study the impact of CU allocation on Mix-1 (compute bound), and Mix-2 (memory bound) using the Overall Constraint policy. The evaluation is presented in Fig. 6.10. The CU allocation is found to have a direct impact on the system throughput (STP) for the compute-bound mix. In contrast, the STP for the memory-bound mix is less sensitive
to CU allocation and is governed by the allocation of memory resources. The performance analyzer determines that the cache is the most sensitive resource for memory-bound applications (Mix-2), and reduces the rate of CU deallocation for low-priority applications. Hence, the rate at which CUs are deallocated from low-priority applications is slower for Mix-2 as compared to Mix-1.

Figure 6.10: Improvement in STP with CU allocation for compute-bound and memory-bound application mixes. The Overall Constraint policy is used with STP constraint of 1.75.

### 6.5.3 Effects of QoS-aware Memory Management

1. **L2 cache management:** Fig. 6.11a shows the change in L2 cache space for every application, versus DED mode execution, for different QoS policies. The average reduction in utilization of the cache for all applications, as compared to their DED mode performance, is 38%. The high-priority constraint and overall constraint policy show improvement in cache space allocation for high and medium-priority applications, as compared to the Naive-MT technique.

   We analyze the impact of cache space allocation using the Misses-Per-Instruction (MPI) metric. Fig. 6.11b illustrates the advantages of using QoS-aware cache allocation by comparing cache space allocation and MPI for each application mix against the Naive-MT technique. The Overall Constraint policy does not limit the degradation of low-priority applications. The loss of cache space for low-priority applications can be larger than 60% when operating under the Overall Constraint policy. Such degradation increases the MPI of the platform versus the Naive-MT execution. The high-priority and medium-priority applications experience significant benefits using Virtuoso defined policies, and exhibit an average 38% increase in cache space over the Naive-MT execution. The performance analyzer identifies the L2-cache and the DRAM cache as the primary factors influencing the performance of the high-priority application in memory-bound mixes. Therefore, the change in L2 cache space for memory-bound mixes (Mix-2 and Mix-4) is much higher than that of compute-bound mixes (Mix-1, Mix-3 and Mix-5).
CHAPTER 6. APPLICATION-AWARE RESOURCE MANAGEMENT FOR GPUs

Figure 6.11: (a) Virtuoso based policies show less reduction in cache space (compared to DED) versus Naive-MT, and (b) Reduction in MPI by assigning more cache space to applications using Virtuoso. Performance is compared versus Naive-MT.

2. Benefits of die-stacked DRAM cache: Inclusion of a die-stacked DRAM cache proves beneficial for applications since they can increase their global memory throughput [66]. Providing more space in the DRAM cache for higher-priority applications not only improves the global memory throughput, but also improves the IPC of the application. We compare the global memory throughput of our QoS policies versus Naive-MT mode execution. We include a DRAM cache in our Naive-MT executions for a fair comparison, and also to highlight the benefits of QoS-aware allocation. Fig 6.12 shows the improvement in global memory throughput for the Overall Constraint policy and the Low Priority Constraint policy. The high-priority applications experience more than a 2x improvement in global memory throughput over Naive-MT. Memory-bound applications (Mix-2, Mix-4) are more sensitive to changes in DRAM cache allocation and show large improvements in IPC as we increase the DRAM cache space.
CHAPTER 6. APPLICATION-AWARE RESOURCE MANAGEMENT FOR GPUS

Figure 6.12: QoS-aware allocation of DRAM cache space using Virtuoso leads to improvements in global memory throughput (GMT) over Naive-MT.

Figure 6.13: Timeline highlighting the behavior-aware resource allocation along with performance enhancements obtained with Virtuoso to achieve QoS targets. Mix-4 (BP/BFS/GE) execution for first 1M cycles is shown.

6.5.4 Case Study: Overall Impact of Virtuoso

As a case study, we show an execution timeline of Mix-4 (BP/BFS/GE) on the multi-tasked GPU with Virtuoso. High Priority Constraint policy is used with Constraint = 0.8x of DED performance for evaluation. We focus on the first 1M cycles of execution for the executing mix. As observed from Fig. 6.13, Virtuoso is invoked after 35K cycles. The performance of BP improves (more than 60%) as more CUs are allocated from cycles 150K to 500K. At 450K cycles, 8 CUs are transferred to the compute-bound BP applications, which results in large performance gains. The
Figure 6.14: Predictive data buffer movement between GPU and host memory using TMM allows mixes to oversubscribe the GPU memory. The total memory subscribed by applications to GPU DRAM memory is shown on the secondary axis.

L2 cache reallocated from GE does not result in huge benefits for compute-bound BP, but improves the performance of the memory-bound BFS application. The same effect is seen when DRAM cache space is reallocated from GE, and given to BP and BFS. We see a performance increase of greater than 20% increase for BP and BFS, which is due to added space in DRAM cache (after 750K cycles). The STP shows a standard deviation of only 5.3% throughout the execution of the mix. As observed from Fig. 6.9, the Mix-4 also shows an average performance improvement of 14% using Virtuoso compared to Naive-MT.

### 6.5.5 Evaluation of Memory Oversubscription

We analyze the efficiency of TMM mechanism by monitoring the usage of the off-chip DRAM memory (global memory) for each application in the concurrently executing application mix. At the same time, we track the total amount of memory subscribed by the executing mix. We examine this in Fig. 6.14 by showing the memory usage of applications belonging to Mix-1 (compute bound), and Mix-2 (memory bound) for the first 10 QoS intervals. The mixes execute under the Overall Constraint policy. The impact of TMM can be clearly observed for the LC (Mix-1) and LUD (Mix-2) mixes, where the memory usage increases for some intervals, and then reduces during others. TMM actively transfers the cold data buffers to the host memory, and creates more space for new buffers to be allocated. The total memory subscribed by the mix continues to increase as applications create more data buffers. The average amount of memory requested by any mix is at least 1.4x greater than the DRAM capacity of the simulated GPU. Some application mixes, such as Mix-1 and Mix-2, show regions of free memory in the DRAM during the execution of the mix. Both mixes consist of cold buffers which are transferred back to the host memory, thereby creating
6.5.6 Sensitivity Analysis of High-Priority Constraints

The High Priority Constraint policy is used to secure a majority of the resources for a majority of the time for the high-priority applications. As we select a constraint closer to the DED performance, the High Priority Constraint policy adopts a more aggressive nature to achieve the QoS targets. This results in an acute degradation for the low-priority application and can also lower the STP. We analyze the performance sensitivity of the High Priority Constraint policy by increasing the value of the constraint from 0.4x to 0.95x of dedicated performance in Fig. 6.15. As we increase the constraint value, the low-priority applications experience extreme performance degradation. The shaded portion of the figure indicates the regions where the low-priority application loses ownership of some of the Virtuoso controlled resources and the high-priority application fails to achieve the QoS target. Fig 6.16 highlights this behavior for the low-priority applications from compute-bound Mix-1 and memory-bound Mix-2. Compute-bound mixes (Mix-1) cause the low-priority application to lose control of CUs, while the low-priority applications in memory-bound mixes (Mix-2) lose control of the L2 cache.
CHAPTER 6. APPLICATION-AWARE RESOURCE MANAGEMENT FOR GPUS

Figure 6.16: Degradation of low-priority applications from (a) Mix-1 (compute bound) and (b) Mix-2 (memory bound), leading to loss of control of one or more resources, and an acute decrease in IPC. High Priority Constraint policy with a constraint of 0.9x of DED performance being used.

6.5.7 Measuring QoS Success rate

The QoS success rate shows the ability of the framework to achieve QoS targets for applications. It is measured using Equation 6.2:

\[
QoS\text{-}Success\text{-}rate = \frac{QoS\text{-}achieved\text{-}intervals}{Total\text{-}num\text{-}QoS\text{-}intervals} \times 100
\]

(6.2)

Where \(QoS\text{-}achieved\text{-}intervals\) is the number of QoS intervals in which the QoS target for the application was met or surpassed. Table 6.3 shows an average QoS success rate of 91.1% across the evaluated Virtuoso policies for all of the application mixes and pairs.

<table>
<thead>
<tr>
<th></th>
<th>High Priority Constraint (Constraint = 0.85x of DED)</th>
<th>Low Priority Constraint (Constraint = 0.5x of DED)</th>
<th>Overall Priority Constraint (STP = 1.75)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mixes (Mix-1 to Mix-5)</td>
<td>91</td>
<td>95</td>
<td>87.5</td>
</tr>
<tr>
<td>COM/MEM pairs (COM-as-High)</td>
<td>89</td>
<td>97.3</td>
<td>90.1</td>
</tr>
<tr>
<td>COM/MEM pairs (MEM-as-High)</td>
<td>86.3</td>
<td>98.1</td>
<td>86.2</td>
</tr>
</tbody>
</table>

Table 6.3: High success rate for Virtuoso based QoS policies for the evaluated application mixes and combinations.

6.6 Discussion

Virtuoso enables a multi-tasked GPU to control performance-critical compute and memory resources. The QoS-aware architecture can be extended to control other performance-critical
elements such as the on-chip interconnect network fabric. Co-executing applications on a multi-
tasked GPU may tend to oversubscribe the interconnect fabric while trying to access the L1 cache,
L2-cache, and DRAM cache from the CUs. Enforcing QoS-aware policies for interconnect uti-
лизация can reduce access latencies for high-priority applications and help us achieve QoS targets
quicker.

To understand the potential of controlling an interconnect for QoS, we implement the
prototype of a scheme to accelerate high-priority accesses between the L2-cache and DRAM cache.
We modify the structure of the DRAM cache read/write buffer by assigning slots for high, medium
and low-priority tasks. For DRAM cache, we implement a 2KB (32 entries X 64 byte entry size)
read/write buffer [66]. We reserve 16 entries for high-priority, 12 entries for medium-priority and 4
entries for low-priority tasks. The unused entries of lower priority levels can be used by the accesses
from higher priority levels. An evaluation of our application mixes using this configuration is shown
in Fig. 6.17 using Virtuoso equipped with a QoS-aware network versus without. We observe an 8.3%
improvement in overall global-memory throughput with use of QoS-aware networks in Virtuoso . In
future work, we plan to extend this design to provide QoS-aware traffic management from L1-to-L2
cache.

6.7 Summary

In this chapter we developed Virtuoso , which extends the concurrent context execu-
tion mechanism described in Chapter 5. Virtuoso realizes a dynamic, QoS-aware, and application
behavior-aware resource manager for multi-tasked GPUs. Our design for Virtuoso uses both the
software runtime and hardware architecture on the GPUs. The hardware level of Virtuoso monitors
the performance of the executing applications periodically, and performs a priority-guided reallo-
CHAPTER 6. APPLICATION-AWARE RESOURCE MANAGEMENT FOR GPUS

cation of CUs, L2-cache and DRAM cache. The software-level component of Virtuoso provides a flexible interface to define QoS policies for co-executing applications. It also uses the Transparent Memory Manager, to manage oversubscription of GPU memory for concurrent applications. Virtuoso is able to improve application throughput by 37% versus the current state-of-the-art spatial multi-tasking system, and achieves the QoS for 91.1% for all of the application mixes and pairs across all evaluated QoS policies.
Chapter 7

Predictive Scheduling for GPU based Cloud Servers using Machine Learning

Cloud and server infrastructures routinely use GPUs for servicing compute tasks from various domains such as signal processing, data mining, biomedical simulations, search, and gaming [61, 106, 110]. A number of large-scale cloud providers and data center operators such as Amazon EC2 [6], Nimbitx [7], and Peer1 Hosting [8] offer GPU services.

Given that application kernels may not always fully utilize a GPU, we developed architectural techniques to support multi-tasking on GPUs, along with QoS-aware resource provisioning. Our work proposed in previous chapters helps achieve a higher throughput using multi-tasked GPU, and provides an interface to define flexible QoS policies for the GPU. The Virtuoso framework ensures that applications achieve their QoS through priority-guided management of multiple resources. In addition, the support for multi-tasking on GPUs has advanced over the years. Modern NVIDIA GPUs can support concurrent execution of data independent computations using HyperQ technology [109]. Techniques such as NVIDIA GRID allow co-execution of up to 16 applications on a cloud platform [4].

However, it can be difficult to effectively schedule concurrent applications on a GPU since the operator does not have any a priori knowledge of the characteristics of the workloads scheduled for execution on the server or cloud system. Hence, scheduling multiple applications on a multi-tasked GPU may lead to workload imbalance and resource contention, caused due to application interference. The Virtuoso framework provides aid in interference detection by providing a linear estimation of the dedicated performance of the scheduled application. It uses this estimate to initiate the resource management. However, linear estimation models can be limited due to their inability to predict different behavior phases of an application. Machine learning (ML) based techniques
have been suggested for estimating application performance on GPUs [161]. But, realizing these technique on GPU hardware leads to heavy compute, storage, and hardware overheads. An efficient, low-overhead runtime mechanism to detect interference among scheduled applications is required for GPU cloud servers and datacenters.

We introduce Mystic, a framework enabling interference-aware scheduling for GPU workloads. Our work targets servers and cloud schedulers by utilizing machine learning algorithms. Mystic utilizes the concurrency features of modern GPUs exposed by programming frameworks such as CUDA 7.0. The major novelty of Mystic is its ability to accurately characterize an unknown application to predict the interference it would cause when co-scheduled with other executing applications. The characterization process of Mystic uses a collaborative filtering approach. Collaborative filtering is commonly used in state-of-the-art recommendation systems deployed for Netflix [26], and Amazon [88]. The characterization predicts the execution behavior of incoming applications, and their similarity to previously scheduled applications. It exploits the data obtained from previously executed applications, coupled with an offline trained application corpus, and a minimal signature obtained from the incoming application.

The rest of the chapter is organized as follows. Section 7.1 describes the challenges for application scheduling on GPU cloud servers. Section 7.2 and Section 7.3 provides the knowledge of GPU remoting and Collaborative filtering based prediction, respectively. These techniques are used to realize the Mystic framework. Section 7.4 describes the Mystic architecture. Section 7.5 describes the evaluation methodology, while Section 7.6 presents the evaluation results.

7.1 Scheduling Challenges for GPU Cloud Servers and Datacenters

The major scheduling goals for such a server or cloud service operator is to achieve high resource utilization, fast execution (low turnaround time), and high system throughput. However, the effective use of GPUs in the current multi-tenant servers and cloud engines is limited by static provisioning of GPU resources. Applications are statically assigned to GPUs and are provided dedicated access to the GPU for the duration of their computation. This dedicated access (i.e., GPU pass-through) often leads to cases where long latency applications can block an entire device, thereby reducing the overall system throughput [90]. Additionally, current static provisioning policies can also underutilize the GPUs, thereby resulting in idle compute resources. Scheduling applications in such cases becomes more difficult since cloud/server operators have to accommodate a diverse set of GPU workloads in terms of their resource needs and performance requirements. Moreover, these applications may interfere with each other while co-execution, and result in an
Figure 7.1: Degradation in performance for 45 applications run on an 8-GPU cluster using interference-oblivious scheduling. The baseline is the execution performance of applications when executed alone (dedicated mode). Applications are arranged according to performance.

lower in system throughput and also degrade their individual performance.

We define interference as any performance degradation of an application on a multi-tasked GPU, caused by resource contention between co-executing applications. The co-executing applications share the Stream Multiprocessors (SMs), caches, memory channels, storage, and the interconnect network. To motivate our work here, we conduct a study to measure the impact of interference (see Figure 7.1). We launch 45 applications selected from the Rodinia (17 apps.) [32], Parboil (9 apps.) [136], and Polybench (10 apps.) [53] suites, as well as the OpenFOAM solvers (9 apps.) [142] on four nodes of a private cluster, with 2 GPUs on each node. We experiment with 2 different interference-oblivious scheduling schemes: Round Robin (RR), and Least Loaded (LL). We observe that the interference-oblivious scheduler introduce resource contention, which generates an average slowdown of 41%, with 8 applications showing more than a 2.5x degradation. We define the Quality of Service (QoS) goals for co-executing applications to be 80% of their dedicated performance. Only 9% applications achieve this QoS target using LL or RR scheduling. To address this problem, our work focuses on resolving workload interference through informed scheduling for a range of workloads run on GPU servers and cloud engines.

Interference-aware scheduling on GPU servers has been attempted in the past [117, 131]. However, the techniques described in these approaches implement time-sharing on GPUs and do not effectively use the advanced support for concurrency on modern GPUs. Interference-aware scheduling for CPU workloads on servers and data centers has been studied in great detail over the past few years [96, 104]. The approaches presented in these studies use empirical analysis to develop
CHAPTER 7. MACHINE LEARNING BASED SCHEDULING FOR GPU CLOUD SERVERS

Figure 7.2: GPU Remoting: (a) architecture of GPU remoting, and (b) implementation of GPU remoting with frontend applications and backend threads (BTs).

models which can help predict interference and guide resource management. In our work, we do not have any a priori empirical, nor static analysis, of the application characteristics. We develop an interference-aware scheduler that can learn the behavior of applications which have already been executed on the system.

*Mystic* tracks the applications executing on each GPU of the server and monitors their execution performance. Once the incoming application is characterized, it is scheduled to run on the GPU that will produce the least amount of interference with already executing applications. Since collaborative filtering allows for highly efficient characterization of potential interference, system throughput and application performance can benefit in a timely manner. *Mystic* leverages analytical techniques and machine learning, and hence, provides a robust framework which can scale to the number of GPUs and nodes in a server.

7.2 GPU Remoting for Cloud Servers and Datacenters

GPU-Remoting is an efficient techniques to schedule multiple applications on multiple GPUs in a cluster or cloud servers. Previous work such as GVim [57], rCUDA [44], vCUDA [132], Pegasus [58] and gVirtus [50], each propose a slightly different approach for GPU Remoting. GPU remoting proposes API-driven separation of the CPU component and GPU component of the applications. Figure 7.2a shows an architecture that supports GPU remoting. The *frontend* of the implementation presents an interposer library that intercepts CUDA calls, while the *backend* executes as a daemon responsible for receiving GPU requests. Requests from various *frontend* user applications can be aggregated into *backend* threads, which can be handled as a single GPU context (see Figure 7.2b). GPU components of all the frontend applications co-executing on the GPU are assigned to separate backend threads. The backend threads map to the same device on a per-GPU
CHAPTER 7. MACHINE LEARNING BASED SCHEDULING FOR GPU CLOUD SERVERS

class basis. This design enables GPU operations from different applications to be executed concurrently, which enables a single GPU to be shared in both space and time [44, 131]. Our work does not propose a new GPU remoting technique, but instead leverages the GPU remoting technique proposed by rCUDA [44] for managing a GPU cluster. We add the Mystic framework on top of rCUDA to provide support for machine learning based application scheduling.

7.3 Collaborative Filtering for Application Performance Prediction

Our work aims to predict the potential interference between GPU applications. The Mystic framework does not try to analyze every application in detail, but leverages information about the applications that are currently running on the system. We try to characterize the incoming application according to its similarity with currently running applications. Applications with similar characteristics will contend for the same resources, and thus, potentially interfere with one another if scheduled on the same GPU. To detect potential interference, we use a recommender system which is built on knowledge of resource usage from a corpus of offline-profiled applications. Note that our training corpus workloads are a different set of applications than those used during the evaluation of our framework. The recommender system is used to guide the scheduler on where to run the incoming applications. As each incoming application is about to be scheduled, we profile the application for a short amount of time to generate a limited profile signature. We then use collaborative filtering to predict the full characteristics of the incoming application based on the limited profile.

Collaborative filtering (CF) is a technique from machine learning that is widely used in recommender systems, an enabling framework made popular by e-commerce companies such as Amazon [88] and Netflix [26]. In an e-commerce environment, the recommender system analyzes the buying history of a user and provides personalized recommendations based on the user’s interests. There are two common methods used in collaborative filtering – neighborhood based methods, and latent factor models [79].

Using a neighborhood based method, the system explores the relationships between users (user-based collaborative filtering) or items (item-based collaborative filtering). To recommend an item to user \( u \), the system can either find similar users and recommend the item they preferred to \( u \), or the system can find the items that are similar to what \( u \) has rated positively, and recommend those items to \( u \). An alternative approach is to use a latent factor model. Instead of characterizing the relationships between users or items alone, a latent factor model describes both users and items in a latent factor domain. Unlike the human produced domains, where a product is categorized into factors (e.g., genre for a movie), the latent factors are concepts that can uncover complicated rela-
CHAPTER 7. MACHINE LEARNING BASED SCHEDULING FOR GPU CLOUD SERVERS

Figure 7.3: Single Value Decomposition (SVD) on a rating matrix.

tionships such as a group of movies with product placements, or two products with totally different user bases. In the case of scheduling applications on GPUs, we use the latent factor model where the incoming application represents the user \( u \), and the missing values in the limited profile represent the products to be recommended.

**Single Value Decomposition** (SVD) is a popular approach for decomposing user-products relations into latent factor domains. Figure 7.3 shows an example of how SVD is applied in a recommender system. The input to the recommender is a matrix \( R_{m \times n} \) (i.e., the rating matrix), where each row represents a user, each column represents a product, and each cell \( r_{ui} \) represents the rating that the user \( u \) gives to product \( i \). The goal of the recommender is to predict the missing values in the matrix (in our case, the recommender predicts the characteristics of the incoming application).

The latent factor model first applies SVD, which is a well-studied matrix decomposition method, to the rating matrix. SVD generates three matrices: \( U_{n \times r} \), \( \Sigma_{r \times r} \), and \( V_{n \times r} \). Matrix \( \Sigma \) is a diagonal matrix whose dimension \( r \) is the number of latent factors and each element in the diagonal represents the confidence of each factor in descending order. A row \( u \) in matrix \( U \) describes to what extent the user \( u \) likes each factor, such as how much \( u \) prefers one movie concept over another concept, or how much \( u \) dislikes a movie with a particular concept. A row \( i \) in matrix \( V \), on the other hand, captures to what extent the product \( i \) possesses those corresponding concepts. To predict the missing value \( r_{ui} \) using SVD, we use:

\[
    r_{ui} = \frac{\sum_{j=1}^{n} u_{ij} \cdot \text{sim}(V_i, V_j)}{\sum_{j=1}^{n} \text{sim}(V_i, V_j)}
\]

Where \( \text{sim()} \) measures the similarity of vector \( V_i \) and \( V_j \). The \( V_i \) represents the \( i \)th row of the matrix \( V \), which identifies the latent factors of the product \( i \). We use \( V_i \) and \( V_j \) to identify the latent factors of various performance characteristics.

In a product/movie dataset, the training rating matrix is always sparse. Thus, **Stochastic Gradient Descent** (SGD) is commonly used to reconstruct the matrix and recover the missing values.
first before SVD. In our implementation of the Mystic framework, we have a fully populated training rating matrix with rows representing training applications from our corpus and columns representing the performance characteristics. Thus, we can use SVD directly to predict the missing performance metrics for an incoming application provided with a limited profile. The performance predictions are then used for determining application interference.

7.4 Mystic Architecture

Mystic is implemented as a 3-stage control layer, hosted in the head node of a GPU cloud server or cluster. The framework is capable of predicting the interference between an incoming application and currently running applications on the server. Mystic guides the scheduler to optimize co-execution of applications on a GPU using the predicted interference. The architecture of Mystic framework shown in Figure 7.4.

7.4.1 The Causes of Interference (CoI)

Contention for resources on a multi-context GPU can lead to interference between multiple co-executing applications. The applications contend for compute resources such as stream multi-processors (SMs), memory resources (e.g., L1 cache, L2 cache, texture caches, global DRAM memory), and the interconnect network. These six resources form the major cause of interference (CoIs) between co-executing applications. To understand the resource usage of each application, we profile the applications using the NVIDIA profiler tools [5]. Table 7.1 presents details on the metrics acquired during profiling for each contended resource. The performance of applications is subject to dynamic factors such as number of kernel launches, and execution grid size on the GPU. To establish a fair comparison among applications, we normalize the profiled performance metrics against the grid size of kernel, and number of kernel launches. When comparing two applications, we consider the difference in performance metrics of the applications for all the CoIs. If the percentage variation (difference) is less than 40% for three or more contended resources, then the two applications are marked as similar, and thereby noted to be interfering.

7.4.2 Stage-1: Initializer and Profile Generator

The initializer starts Mystic, and invokes different modules of the framework and creates the book-keeping mechanism to track the progress and status of each application. The initializer queries the cluster status table. The framework obtains the IP address, the number of CPU cores, the number of GPUs, and the amount of system memory present on each compute node. The
The architecture of Mystic. The Mystic framework is implemented as a 3-stage controller hosted in the head node of the GPU cluster or cloud server. The initializer uses this information to create a status entry for each incoming application in the Mystic Application Status Table (MAST).

Mystic detects the interference between applications by classifying applications according to their similarity. The framework needs to obtain profiler metrics for each incoming application to aid the process of similarity detection. Obtaining a full profile for each incoming application...
leads to redundant computation and introduces delay, which cannot be tolerated in service-oriented cloud servers and clusters. Instead, *Mystic* initiates two short profiling runs for each incoming application to obtain metrics for two randomly selected CoIs (out of 6 identified CoIs). The profiler run needs be long enough to profile each distinct kernel in the application at least once. We analyzed 37 applications (from Rodinia [32], Parboil [136] and the NVIDIA SDK [111]). We found that a profiling time of 5 seconds captures profile metrics of every kernel in our applications. The head node maintains two separate GPUs for initiating the profile runs. The short-profiles (~5 seconds) for incoming applications are collected and stored in the Profile Information Table (PIT) in form of sparse rows, as metrics for only 2 random CoIs out of 6 are captured. The PIT is indexed by the application process ID (pid).

To fill the missing values in the PIT entry for each application, we use collaborative filtering (CF). As described in Section 7.3, CF requires a rating matrix which contains the history of the previously observed values (metrics in our case). Our rating matrix maintains full profiles of several applications which are profiled offline. The rating matrix is referred to as the Training Matrix (TRM) in the *Mystic* framework. The next stage of *Mystic* performs the collaborative filtering using the TRM and PIT.

### 7.4.3 Stage-2: Collaborative Filtering (CF) based Prediction

The CF predictor takes the PIT and TRM as inputs. When a new application $A_0$ is enqueued for execution on the system, the predictor identifies $A_0$’s profile information by searching the PIT using the process-id (pid) of the application. The PIT returns a sparse vector $\vec{v}$ with the metrics obtained from the short profiles collected in Stage-1. The sparse PIT entry does not contain the metrics of all CoIs. To fill out the missing entries using collaborative filtering we append the

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Resource</th>
<th>Metric Name</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM Efficiency</td>
<td>SM</td>
<td>L2 reads</td>
<td>L2 cache</td>
</tr>
<tr>
<td>IPC</td>
<td>SM</td>
<td>L2 writes</td>
<td>L2 cache</td>
</tr>
<tr>
<td>DRAM read throughput</td>
<td>DRAM</td>
<td>Texture cache read throughput</td>
<td>Texture cache</td>
</tr>
<tr>
<td>DRAM write throughput</td>
<td>DRAM</td>
<td>Texture cache hit rate</td>
<td>Texture cache</td>
</tr>
<tr>
<td>DRAM reads</td>
<td>DRAM</td>
<td>Texture cache reads</td>
<td>Texture cache</td>
</tr>
<tr>
<td>L1 reads</td>
<td>L1 cache</td>
<td>Global Loads</td>
<td>Interconnect</td>
</tr>
<tr>
<td>L1 hit rate</td>
<td>L1 cache</td>
<td>Global Load throughput</td>
<td>Interconnect</td>
</tr>
<tr>
<td>L1 read throughput</td>
<td>L1 cache</td>
<td>Global Store throughput</td>
<td>Interconnect</td>
</tr>
</tbody>
</table>

Table 7.1: The metrics used for profiling resource usage.
vector $\vec{v}$ to the TRM (Training Matrix), as shown in Figure 7.5. We then perform SVD-based collaborative filtering (as described in Section 7.3) to fill out all of the missing values in the TRM. The rating matrix (TRM in our case) has only one row of missing incoming values, since we only append one new application vector $\vec{v}$ to the TRM at a time. This allows predictions to be independent of other incoming applications, which increases the prediction accuracy. The prediction generated by the predictor is the vector $\vec{v}'$, for which all the CoI metrics have been predicted. The vector $\vec{v}'$ is written to the Prediction Table (PRT), which holds the CF predicted performance values of all CoIs for the incoming applications and current running applications. The PRT is used to guide the interference-aware scheduler in Stage-3 of the Mystic framework.

### 7.4.4 Stage-3: Interference Aware Scheduler

To guide the scheduling of incoming applications on a GPU node, while introducing the minimum interference, we implement an Interference-Aware Scheduler. The scheduler includes a similarity detector, which takes the Prediction Table (PRT) and the MAST as the inputs, and generates the similarity between pairs of applications. The scheduling algorithm decides whether the incoming application $A_0$ will not be executed concurrently with an executing application $A_1$ on the same GPU if:

1. There are other idle GPUs in the system, or
2. If there is at least one currently running application $A_2$ that is less similar to $A_0$ than $A_1$.

More specifically, whenever a new application arrives, the scheduler first checks for an idle GPU. If more than one idle GPU is found, then the application is assigned to the GPU nearest to the head node. In the absence of an idle GPU, the similarity detector finds the entry for incoming application in the PRT using the application $pid$. The detector extracts the predicted metrics of the new application from the PRT and arranges them in a vector $\vec{v}_{\text{new}}$. 
7.4.5 Validation

We validate our CF-based prediction model by comparing clustering results for the predicted performance (CF-based prediction) against the actual performance profiles of the applications. During validation, the Training Matrix (TRM) consists of full profiles of applications from Rodinia \[32\], and Parboil \[136\] suites. We use 16 applications taken from the NVIDIA SDK as the testing set \[111\].

As a first step, we capture full profiles of the applications in the testing set. As a next step, we capture the limited-profiles (\(~5\) second runs) and capture 2 out of 6 CoIs for all the testing set applications. Later, we fill out the missing metrics in the limited profiles using our CF-based predictor described in Section 7.4.3. Mystic creates the predicted profile for each testing set application. As a final step, we apply agglomerative clustering \[23\] to both the full profiles and predicted profiles of the validation set applications. Figure 7.6 shows the clustering results of using both the full-profile and predicted-profile. Mystic relies on accurate predictions of similarity between applications to calculate the interference scores. Hence, the clustering similarity of both the profiles provides an indication of the accuracy of the CF predictor. Clustering similarity can be compared using the Normalized Mutual Information (NMI) metric described by Strehl et al. \[138\]. The NMI of the the clustering results for the 3 clusters is 0.89.

7.5 Evaluation Methodology

7.5.1 Server Setup

We evaluate the performance and efficiency of the Mystic framework using a private GPU cluster. The cluster system includes a head node housing two GPUs, and 16 distinct compute nodes, each with two GPUs per node. Table 7.2 provides the details of the configuration of our evaluation

\[^1\text{NMI is a higher-is-better metric, ranging from zero to one} \]
platform. We have deployed the rCUDA framework for GPU remoting on each of the compute nodes [44]. The Mystic framework is hosted on the head node of the cluster, and uses node queues to dispatch work to compute nodes. Mystic implements routines to communicate with the rCUDA framework on each of the compute nodes to update the MAST (Mystic Application Status Table).

We use four configurations of 2 nodes, 4 nodes, 8 nodes, and 16 nodes to evaluate the scalability of Mystic.

### 7.5.2 Workloads for Evaluation

We select 55 distinct workloads taken from the Polybench [53], Lonestar [81], NU-PAR [151], and SHOC [36] benchmark suites. We also include workloads from OpenFOAM GPU solvers [142], and ASC proxy applications used for exascale computing research [156]. In addition, we leverage tuned CUDA libraries such as cuDNN (deep learning libraries) [33] and MAGMA (lin-
Table 7.3: Multi-application workloads from 8 application suites.

<table>
<thead>
<tr>
<th>Suite</th>
<th># Apps.</th>
<th>Name of Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolyBench-GPU</td>
<td>14</td>
<td>2d Conv, 3d Conv, 3mm, Atax, Bicg, Gemm, Gesummv, Gramschmidt, Mvt, Syr2k, Syrk, Correlation, Covariance, Fdtd-2d</td>
</tr>
<tr>
<td>SHOC-GPU</td>
<td>12</td>
<td>Bfs-shoc, FFT, MD, MD5Hash, Neuralnet, Scan, Sort, Reduction, Spmv, Stencil-2D, triad, qt-Clustering</td>
</tr>
<tr>
<td>Lonestar-GPU</td>
<td>6</td>
<td>BHN-body, BFS, LS, MST, DMR, SP, Sssp</td>
</tr>
<tr>
<td>NUPAR-Bench</td>
<td>5</td>
<td>CCL, LSS, HMM, LoKDR, IIR</td>
</tr>
<tr>
<td>OpenFOAM-GPU</td>
<td>8</td>
<td>dsmcFoam, PDRFoam, thermoFoam, buoy-antSFoam, mhdFoam, simpleFoam, sprayFoam, driftFluxFoam</td>
</tr>
<tr>
<td>ASC Proxy Apps</td>
<td>4</td>
<td>CoMD, LULESH, miniFE, miniMD</td>
</tr>
<tr>
<td>Caffe-cuDNN</td>
<td>3</td>
<td>leNet, logReg, fineTune</td>
</tr>
<tr>
<td>MAGMA</td>
<td>3</td>
<td>dasaxpycp, strsm, dgesv</td>
</tr>
</tbody>
</table>

Ear algebra solvers) [146]. Table 7.3 lists all of the workloads used in our evaluation. Apart from the evaluation workloads, we use full-profiles of 42 distinct applications from from Rodinia [32], Parboil [136] and NVIDIA SDK [111] as Training Matrix in our CF predictor (see Section 7.4). The 55 applications can be scheduled on the cluster using $55! \approx 1.269\times 10^{73}$ launch schedules. We select 100 random launch sequences for evaluating our scheduler. We vary the load on the server by randomly selecting the application arrival times. We consider light, medium and heavy application loads. Applications are scheduled with an inter-arrival time of 8 seconds, 5 seconds, and 2 seconds for light load, medium load, and heavy load scenarios, respectively. Most of our evaluations are done using medium load scenarios.

7.5.3 Metrics

We use the performance metrics suggested by Eyerman et al. for evaluating multi-programmed workloads and systems [45]. The metrics compare the performance of application run in a dedicated mode versus in co-execution mode. **Average Normalized Turnaround Time (ANTT)** is the arithmetic average of the turnaround times for the applications in a workload, normalized to their single application execution. **System Throughput (STP)** measures the overall system performance and expresses work completed in unit time.
Figure 7.7: Performance impact using Mystic medium load. Applications are arranged in order of increasing performance.

**Fairness** measures equal progress of applications in a multiprogrammed workload. We use Jain’s fairness index which is expressed as a number between zero and one [64]. One indicates perfect fairness (co-executing processes experience equal slowdown) and zero indicates no fairness at all. Mystic is also evaluated for its efficiency in improving GPU utilization.

### 7.6 Evaluation Results

#### 7.6.1 Scheduling Performance

The performance of Mystic is compared to the Round Robin (RR) and Least Loaded (LL) scheduler in Figure 7.8. The performance is evaluated as overall speedup achieved, normalized to the baseline. The baseline for all our evaluations is the performance of applications when run in dedicated mode (executing alone on the GPU). Overall, Mystic based scheduling outperforms the interference-oblivious schedulers. We observe that over 90% of applications achieve their QoS objectives using Mystic scheduling. Only 21% of the applications achieve their QoS objectives using the RR scheduler, while none of the applications achieve their QoS goal when using a LL scheduler.
CHAPTER 7. MACHINE LEARNING BASED SCHEDULING FOR GPU CLOUD SERVERS

Figure 7.8: Performance of Mystic when scaling the number of compute nodes.

Figure 7.9: Performance of our three schedulers when the GPU server is under (a) Light Load, and (b) Heavy Load.

We allow co-execution of at most 3 applications on the same GPU. This hard bound on the number of applications per GPU helps to meter the amount of performance degradation. The LL scheduler only focuses on the load on the GPU, and unknowingly co-schedules applications to run on the same GPU that significantly interfere with each other. Many short-running applications with high interference get scheduled to the same GPU using LL. This results in the LL scheduler experiencing a linear performance degradation as we increase the number of applications. On comparison, Mystic bounds the performance degradation, as more than 75% applications have less than 15% degradation. As shown in Figure 7.8, the performance of Mystic is scales effectively as we double the number of compute nodes on the server, and shows . For fewer compute nodes, LL scheduler utilizes the GPUs more effectively than the RR scheduler. Thus LL shows 19% less degradation in performance as compared to RR for 2 node execution.

We also evaluate the behavior of Mystic while we vary the load on the server. Figure 7.9a shows results for a lightly loaded server, and Figure 7.9b shows results for a heavily loaded scenario. For light loads, all the schedulers show little degradation in performance versus the baseline. Mys-
CHAPTER 7. MACHINE LEARNING BASED SCHEDULING FOR GPU CLOUD SERVERS

7.6.2 System Performance Using Mystic

In this section, we analyze the overall performance of the system when using the Mystic framework. Figure 7.10 shows the average normalized turnaround time (ANTT) and the system throughput (STP) for each of 100 launch sequences. Both ANTT and STP are system level metrics which analyze the performance of a workload mix, which is the launch sequence in our case. From Figure 7.10, we observe that Mystic enjoys the lowest ANTT among all of the schedulers (ANTT is a lower-is-better metric). The average ANTT for the LL scheduler is 9.1, which indicates an average slowdown of 9.1x per application across all the launch sequences. Interference-oblivious scheduling of applications for co-execution can heavily impact execution performance of individual applications. In contrast, Mystic achieves an ANTT value of 4.7, which is 48% lower than the the ANTT for LL, and 41% lower than the ANTT found for RR.

System throughput (STP) is an important metric to assess the overall efficiency of the
system. We plot the STP for each scheduler in Figure 7.10b. The STP obtained using Mystic is 40% higher than the LL scheduler, and 16% higher than the RR scheduler. LL scheduler many times selects the same GPU repeatedly in the case of launching small, short-running, applications. This leads to cases where only few GPUs are used to execute many, small, interfering applications, while the larger applications occupy other GPUs. This results in the slowdown of smaller applications and impacts system throughput significantly, reducing the STP for LL. Alternatively, the RR scheduler inherently uses all the GPUs to launch applications. But, interference between co-execution applications results to a lower STP. Moreover, 63% (avg.) co-executing application pairs/triads using RR and LL scheduler are characterized as interfering by Mystic.

7.6.3 GPU Utilization

GPUs are the prime resources in our evaluation. We analyze the utilization of the 32 GPUs across all nodes for a medium load scenario. Figure 7.11 shows the average utilization of each GPU for the entire execution over all launch sequences. We observe that Mystic is able to balance the distribution of applications on the server and enables an average utilization of 90.5% for all of the GPUs. The Mystic scheduler is interference-aware and utilization sensitive. As described in Section 7.4, the Mystic scheduler tries to assign application to GPUs that results in the lowest interference possible, while also considering device utilization. The LL scheduler oversubscribes GPUs by co-executing interfering applications on the same GPUs. As described in previous section, the LL scheduler can increase loads on the GPU with short-running applications. Figure 7.11 shows that, for the LL scheduler, many GPUs are underutilized, while others are oversubscribed. The
CHAPTER 7. MACHINE LEARNING BASED SCHEDULING FOR GPU CLOUD SERVERS

Figure 7.12: Coefficient of variation (COV) computed for the runtime of each application across the 100 launch sequences. Lower COV indicates higher quality of the selected launch sequence.

Figure 7.13: Scheduling decision quality evaluated, using: (a) Performance degradation observed for each scheduling scheme, and (b) The fairness achieved for each of the co-executing applications.

average GPU utilization for the RR scheduler is 79.3%.

7.6.4 Quality of Launch Sequence Selection

As described in Section 7.5, we use 55 distinct applications for evaluating the Mystic framework. The launch sequence describes the order of arrival of the applications on the server. As we have chosen 100 random sequences (from a pool of fact(55)) to conduct experiments, we need to evaluate the quality of our selected sequences. To analyze the quality, we determine the execution time of each of the 55 application when executed in each of the 100 sequences. Next, we calculate the coefficient of variation (COV) for applications across 100 launch sequences. The COV describes how spread-out the data is in the given set of execution times for applications in each sequence. Figure 7.12 shows the COV of each application across 100 launch sequences. The selected launch sequences show a low average COV of 6.3%. The low COV suggests that our selected launch sequences are representative of all possible sequences.
CHAPTER 7. MACHINE LEARNING BASED SCHEDULING FOR GPU CLOUD SERVERS

Figure 7.14: Performance variation between collaborative filtering based scheduling, and full-profile guided scheduling using Mystic. Lower variation values suggest higher accuracy.

7.6.5 Scheduling Decision Quality

We evaluate the scheduling decisions made by Mystic, analyzing the performance degradation suffered by each application over its dedicated execution. From Figure 7.13, we observe that more than 90% of the applications achieve less than 20% degradation using Mystic. In contrast, the interference-oblivious schedulers experience more than 30% degradation for a large fraction of their applications. The interference introduced by the LL and RR schedulers leads to significant slowdown. More than 80% of the applications scheduled using Mystic show no negative-interference. This is evident from the fairness index of the Mystic scheduler, as observed in Figure 7.13. A fairness index represents the per-application interference when two or more applications share GPU resources. A fairness index closer to one represents less interference between applications. Mystic achieves an average fairness improvement of 34% and 81% over the RR scheduler and the LL scheduler, respectively.

Another important factor which affects the scheduling decisions with Mystic is the accuracy of the collaborative filtering (CF) based prediction. As described in Section 7.4, the CF predictor uses short execution profiles (∼5 seconds) to predict the overall performance of the application. The short profile runs do not capture the entire dynamic behavior of the application, and may provide misleading predictions. These predictions may cause Mystic to incorrectly schedule the applications. We evaluate the accuracy of CF by scheduling applications using the CF predictor, and using the full-profile of the application. We compare the execution performance for each application when scheduling a medium load on the GPU server. The average performance variation is calculated for each of the 55 applications across the 100 launch sequences. The Figure 7.14 shows as low as 7.8% difference (avg.) in performance between full-profiled scheduling, and CF prediction-based scheduling.

7.6.6 Scheduling Overheads Using Mystic

We evaluate the overhead imposed by each of the evaluated schedulers. This includes the overhead for scheduling and training (profiling and recommender based prediction). We measured
CHAPTER 7. MACHINE LEARNING BASED SCHEDULING FOR GPU CLOUD SERVERS

Figure 7.15: Overheads incurred with the Mystic scheduler for twenty representative applications across 100 launch sequences.

the overhead of launching 20 representative applications from our application mix. The applications are executed as part of the medium load scenario on the server. The Figure 7.15 presents the application performance normalized to the baseline, and shows the breakdown of each overhead. The training phase of profiling and prediction does not exist for interference-oblivious schedulers, whereas the training phase constitutes an average of 13.4% of the execution time for Mystic scheduler. The profiling stage (of ~5 seconds) of training makes up 89.3% of the total training time. Effectively, the overall overhead of the recommender is less than 1.5% of the total execution time of the application.

7.7 Summary

In this chapter, we have presented Mystic, an interference aware scheduler for GPU clusters and cloud servers. Mystic uses collaborative filtering to predict the performance of incoming applications and determines the interference with running applications. Mystic relies on information from previously executing applications, as well a limited profile of the incoming application. The Mystic scheduler assigns the incoming application to a GPU for co-execution such that it minimizes interference and improves system throughput. We perform extensive evaluation of the Mystic framework using 55 real world applications, with 100 distinct launch sequences and varying load scenarios. We observe that Mystic achieves the QoS goals for over 90% of our applications when scheduling a medium load on a GPU server. For 35 of the 55 applications evaluated, Mystic was able to maintain co-execution performance within 15% of dedicated performance. Mystic improves system throughput by 27.5% and GPU utilization by 16.3% when compared to the best performing interference-oblivious schedulers.
Chapter 8

Conclusion

We have analyzed the requirements of applications ported to the GPU, across different computing domains such as cloud engines and data centers. Our analysis identifies that modern applications require flexible resource allocation, strict timing constraints, and quality-of-service (QoS) guarantees. The platforms hosting these applications, desire for an increased resource utilization and sustained system throughput when using a GPU. We develop a hardware and runtime level solution for GPUs to realize the demands of future GPU computing.

In this thesis, we performed an in-depth exploration of the challenges associated with adding multi-level concurrency support on modern GPUs. We showed the potential benefits of mapping multiple command queues on a GPU. We implemented an adaptive spatial partitioning mechanism on the GPU, which partitions the GPU compute units according to the size of the applications. We provided a runtime interface to expose the kernel concurrency to users. Our study explored a range of scheduling policy trade-offs related to partitioning of the GPU for concurrent execution of multiple kernels, and also explored different workgroup scheduling mechanisms. The adaptive partitioning of compute units results in performance gains and better utilization of device resources (up to 3x over a single command queue implementation).

To increase the levels of concurrency support on the GPU, we provided the architectural and runtime design for concurrently executing multiple application contexts on the GPU. Our design increases the utilization of the GPU resources, which significantly benefits workloads targeted for data centers and cloud engines. We describe new architectural and runtime enhancements needed to support multi-context execution on a GPU. We added a context-management-layer (CML) in the runtime to handle memory transfers for contexts and manage the dispatch of contexts to the GPU. We also describe a new Transparent Memory Management (TMM) facility and evaluate the benefits of this combined host/GPU-based control mechanism versus other host-side approaches. Similarly,
we enabled the GPU to handle complex firmware tasks using an upgraded command processor (CP). Our implementation for address space isolation in a shared L2 cache and TLBs incurs less than 0.4% overhead in cache area. Our overall design for multi-context execution achieves an average speedup of 1.56X across application mixes executing concurrently on the GPU.

We have enhanced our multi-level concurrency framework by providing support for QoS-aware resource management, and an interface to change QoS policies on the GPU. We implemented the Virtuoso framework, which provides dynamic, QoS-aware, and application behavior-aware resource management on the multi-tasked GPUs. Virtuoso monitors the performance of the executing applications periodically, to perform a priority-guided reallocation of CUs, L2-cache and DRAM cache. We also provided a memory mapped register based interface to define QoS policies for co-executing applications. Virtuoso is able to improve application throughput by 37% versus the current state-of-the-art spatial multi-tasking system. Using the TMM mechanism, concurrent applications are observed to efficiently access all of their DRAM memory data, even if the GPU memory is oversubscribed up to 1.4x its capacity.

We implemented the Mystic framework, which provides an interference aware scheduler for GPU clusters and cloud servers. Mystic is designed as a runtime solution as an aid to the multi-level concurrency architecture on the GPU. Mystic uses collaborative filtering to predict the performance of incoming applications and determines the interference with running applications. Virtuoso relies on information from previously executing applications, as well a limited profile of the incoming application. The Virtuoso scheduler assigns the incoming application to a GPU for co-execution such that it minimizes interference and improves system throughput. We performed extensive evaluation of the Virtuoso framework using 55 real world applications, with 100 distinct launch sequences and varying load scenarios. We observed that Virtuoso achieves the QoS goals for over 90% of our applications when scheduling a medium load on a GPU server.

This thesis delivers an in-depth design of multi-level concurrency for GPUs. We provide sophisticated, yet low-overhead, mechanisms for compute resources orchestration, memory sub-system management, QoS support, and interference detection. Our solution encompasses the different design aspects and helps fulfill the demands of the newer class of GPU applications, while improving performance of the existing applications.

8.1 Future Work

The multi-level concurrency for GPUs presented in this thesis can lead to new avenues for research in the future.
CHAPTER 8. CONCLUSION

The adaptive spatial partitioning can be extended to support compute unit affinity, to preserve cache locality. Also, the scheduler can be extended to support power-aware throttling of certain kernels, to realize the energy goals of the system. Our implementation of multi-context execution uses tag extensions for identifying contexts at the L2-cache and DRAM-cache level. Future research directions can target sophisticated mechanisms to partition the L2-cache and DRAM-cache based on utility or reuse distance. As GPU designs continue to evolve, our next steps include support for multi-context execution on the same compute-unit and L1 cache partitioning for the multiple contexts.

In our work, we provided a partial implementation of a QoS-aware interconnect network. As a part of immediate future work, we plan to extend Virtuoso to provide full support for managing interconnect networks as a part of the QoS framework. We also plan to enhance the Mystic framework to be GPU architecture-aware, for efficient co-execution on diverse GPU devices on a dense cloud server. The next path of research would be an effective GPU workload migration, from one device to another with the help of checkpointing schemes.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


[100] A. Munshi. The opencl specification 2.0.


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


