ARCHITECTURAL SUPPORT FOR IRREGULAR PROGRAMS AND PERFORMANCE MONITORING FOR HETEROGENEOUS SYSTEMS

A Thesis Presented
by
Perhaad Mistry
to
The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the field of
Computer Engineering
Northeastern University
Boston, Massachusetts
April 2014
Abstract

Since the advent of heterogeneous computing a large number of applications have been ported to utilize heterogeneous systems. Data parallel applications have mapped their computation to the large numbers of cores in heterogeneous systems and reported large performance improvements. However, the programming APIs targeting heterogeneous systems require explicit data movement and thread management by the application developer. Low level programming models such as OpenCL complicate the performance optimization of closely coupled applications executing on heterogeneous systems.

Closely coupled applications refers to computational problems with frequent communication between the host and the device, where computation carried out on the host and device affect each other. Developing architectural support for efficient host device interaction in closely coupled applications is an open problem in heterogeneous systems which if addressed can enable new application spaces for heterogeneous systems, simplify development and improve performance.

This thesis proposes architectural enhancements to the profiling and workgroup scheduling subsystems of heterogeneous devices. The profiling and workgroup scheduling subsystems have been augmented with a resource known as the Offload Control Unit (OCU). The OCU enables performance monitoring of compute units with throughput counters. Throughput counters provide utilization information of compute units and the performance knowledge generated is utilized to improve execution performance for priority and data-driven workloads. Throughput counters and the software profiling subsystems result in a runtime that allows performance monitoring, profiling and specializations of applications built using heterogeneous computational pipelines. The scheduling capabilities proposed enable utilization of heterogeneous systems for workloads with QOS and non-homogeneous workgroup distributions.

This thesis also proposes a benchmark suite for heterogeneous systems where flexibility in behavior is a primary guiding design choice. The benchmark suite has led to the construction of application features to model benchmark behavior. The features have been applied to study application behavior on different heterogeneous systems at different layers of abstraction. The benchmarks and the application classification methodology enables study of behavior of heterogeneous architectures across applications or different behaviors seen within the same application.
Acknowledgements

Frodo: I wish the ring had never come to me. I wish none of this had happened.
Gandalf: So do all who live to see such times. But that is not for them to decide. All we have to decide is what to do with the time that is given to us. There are other forces at work in this world Frodo, besides the will of evil. Bilbo was meant to find the Ring. In which case, you were also meant to have it. And that is an encouraging thought.

I have often been angry about why I started the PhD and I see myself retreating to the above quotations more often than I’d like. I think the only reason I can safely offer as to why I got a PhD degree is that there was something more than a force of evil at work. Even Gandalf would agree, it was more like the force of Kaeli’s will rather than any evil out of middle-earth. For some reason he trusted me even though I had no previous research experience, and let me have a desk in his lab. His trust in me with awesome projects and his high-level guidance whenever I was stuck is the only reason why this thesis was completed. His willingness to run through brick-walls for his graduate students has motivated me to do the same for him over the past 5 years.

My committe of Norm Rubin and Prof. Schirner have been an extremely important guiding light of this thesis. This thesis essentially started when Norm let me join his group in AMD for 3 months and work on OpenCL applications. Working under him at AMD led to my full time job with the awesome AMD Boston Devtools group. A number of past and present NUCAR members (Byung, Rafa, Dana, Yash etc.) have also helped so much with applications, debugging and results. I do not know where I’d be without them.

I would like to acknowledge a bunch of friends outside NUCAR, collectively known as the mandal for all their support and dragging me out of the lab with persistent phone calls on Fridays and Saturdays.

I would also like to acknowledge my parents for allowing me the freedom to pursue the PhD without any constraints. My sister Sanobar for the continuous encouragement when I was bored beyond belief and needed a non-engineer to talk to. Last but not remotely the least, I would like to thank my awesome wife Khushnama, who has made my life more amazing than she will ever know.
## Contents

### Abstract

### Acknowledgements

<table>
<thead>
<tr>
<th>1 Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Introduction</td>
</tr>
<tr>
<td>1.2 Performance Analysis History</td>
</tr>
<tr>
<td>1.3 Heterogeneous Applications</td>
</tr>
<tr>
<td>1.4 Application Performance Optimization for Heterogeneous Systems</td>
</tr>
<tr>
<td>1.5 The Offload Control Unit</td>
</tr>
<tr>
<td>1.6 Contributions of this Thesis</td>
</tr>
<tr>
<td>1.7 Organization of this Thesis</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2 Background and Related Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Introduction</td>
</tr>
<tr>
<td>2.2 OpenCL</td>
</tr>
<tr>
<td>2.3 Current Heterogeneous Architectures</td>
</tr>
<tr>
<td>2.4 Profiling Approaches and Support for Performance Analysis</td>
</tr>
<tr>
<td>2.5 Profiling Interfaces in Programming Models</td>
</tr>
<tr>
<td>2.6 Architectural Support to Generate Performance Information</td>
</tr>
<tr>
<td>2.7 Benchmark Suites and Workload Characterization</td>
</tr>
<tr>
<td>2.8 Summary</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3 Feature Based Heterogeneous Application Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
</tr>
<tr>
<td>3.2 Multi-Layer Workload Models</td>
</tr>
<tr>
<td>3.3 AL0 Features</td>
</tr>
<tr>
<td>3.4 AL1 Features</td>
</tr>
<tr>
<td>3.5 AL2 Features</td>
</tr>
<tr>
<td>3.6 AL3 Features</td>
</tr>
</tbody>
</table>
8 Performance Results
8.1 Tested Heterogeneous Platforms ........................................... 85
8.2 Profiling Experiments Using Haptic ..................................... 86
8.3 Valar Characterization .......................................................... 89
8.4 Additional Throughput Counter Performance Results ................. 94
8.5 OCU Subsystem Evaluations ................................................. 97
8.6 Summary of Performance Results ......................................... 103

9 Summary and Conclusions .................................................. 104
9.1 Completed Work ................................................................. 104
9.2 Future Research Directions ................................................ 105
9.3 Publication List ................................................................. 108

Bibliography ................................................................. 110
# List of Figures

1.1 Moore’s Law Scaling .................................................. 5  
1.2 Persistent Kernel Application ..................................... 8  
1.3 High level OCU model ............................................... 9  

2.1 OpenCL Architecture .................................................. 14  
2.2 Presently available OpenCL compliant heterogeneous systems ........................................ 17  
2.3 Profile driven analysis of heterogeneous applications ................................................. 23  

3.1 Abstraction layers for Studying Heterogeneous Applications .................................... 28  
3.2 Abstraction levels AL0 and AL1 in heterogeneous applications ........................................ 29  
3.3 AL2 Level Features .................................................... 30  
3.4 AL3 Level Features .................................................... 31  

4.1 The dynamic execution patterns of applications considered for Valar ............................ 37  
4.2 The SURF Algorithm .................................................. 40  
4.3 Adaptive FIR Filter .................................................. 40  
4.4 Collision Detection Application ..................................... 41  
4.5 Traffic Simulation ..................................................... 41  
4.6 Search benchmark .................................................... 42  
4.7 Imbalance Microbenchmarks .......................................... 45  
4.8 Storm Topologies and their mapping .................................. 47  

5.1 Haptic Architecture based on OpenCL .................................. 51  
5.2 Programming API for Haptic ........................................ 54  
5.3 FIR Application usage in Haptic ........................................ 55  
5.4 SURF usage in Haptic .................................................. 56  

6.1 AMD Evergreen and Southern Island Compute Units ................................................. 61  
6.2 Imbalanced vector addition microbenchmark ...................................................... 62  
6.3 High Level Profiling Model ............................................ 63  
6.4 Simulation methodology ................................................ 64  
6.5 Throughput counter data example ........................................ 64
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.6</td>
<td>Clustering of Throughput Counters</td>
<td>67</td>
</tr>
<tr>
<td>6.7</td>
<td>Distribution of clusters into workloads</td>
<td>68</td>
</tr>
<tr>
<td>7.1</td>
<td>FSM describing interaction within heterogeneous systems</td>
<td>70</td>
</tr>
<tr>
<td>7.2</td>
<td>FSM describing interaction within an OCU-based heterogeneous system</td>
<td>71</td>
</tr>
<tr>
<td>7.3</td>
<td>Functional model of the OCU system</td>
<td>72</td>
</tr>
<tr>
<td>7.4</td>
<td>Motivating WG Scheduling</td>
<td>73</td>
</tr>
<tr>
<td>7.5</td>
<td>Concurrent Command Queue Model</td>
<td>75</td>
</tr>
<tr>
<td>7.6</td>
<td>OpenCL Pipes Model</td>
<td>77</td>
</tr>
<tr>
<td>7.7</td>
<td>Memory hierarchy of modeled Southern Islands GPU device</td>
<td>80</td>
</tr>
<tr>
<td>7.8</td>
<td>Evaluation of execution scalability with a communication channel</td>
<td>81</td>
</tr>
<tr>
<td>7.9</td>
<td>Evaluating restrictions on scheduling iterations</td>
<td>82</td>
</tr>
<tr>
<td>7.10</td>
<td>Evaluating atomics overhead in pipe based applications</td>
<td>83</td>
</tr>
<tr>
<td>8.1</td>
<td>FIR execution results with Haptic</td>
<td>87</td>
</tr>
<tr>
<td>8.2</td>
<td>Effect of Haptic’s Analysis Devices on SURF’s performance</td>
<td>88</td>
</tr>
<tr>
<td>8.3</td>
<td>Feature count tracking for (a) detect scene change (b) regulate feature count</td>
<td>89</td>
</tr>
<tr>
<td>8.4</td>
<td>Variation in execution time for the Adaptive FIR application</td>
<td>91</td>
</tr>
<tr>
<td>8.5</td>
<td>Search execution behavior</td>
<td>92</td>
</tr>
<tr>
<td>8.6</td>
<td>Throughput of Physics application</td>
<td>93</td>
</tr>
<tr>
<td>8.7</td>
<td>Throughput of large particles in Physics</td>
<td>94</td>
</tr>
<tr>
<td>8.8</td>
<td>SI GPU Workgroup Distribution</td>
<td>95</td>
</tr>
<tr>
<td>8.9</td>
<td>Variation in compute unit utilization over time</td>
<td>96</td>
</tr>
<tr>
<td>8.10</td>
<td>Variation in compute unit utilization over time</td>
<td>97</td>
</tr>
<tr>
<td>8.11</td>
<td>SI GPU Concurrent Command Queue execution</td>
<td>99</td>
</tr>
<tr>
<td>8.12</td>
<td>GPU Concurrent queue utilization with load imbalance</td>
<td>99</td>
</tr>
<tr>
<td>8.13</td>
<td>Performance improvement when using multiple command queues</td>
<td>100</td>
</tr>
<tr>
<td>8.14</td>
<td>Memory performance when using multiple command queues</td>
<td>101</td>
</tr>
<tr>
<td>8.15</td>
<td>Application usage scenarios for pipes</td>
<td>102</td>
</tr>
<tr>
<td>8.16</td>
<td>Execution performance of multiple command queues using a pipe object</td>
<td>102</td>
</tr>
<tr>
<td>8.17</td>
<td>L2 cache performance for applications using an OpenCL pipe</td>
<td>102</td>
</tr>
<tr>
<td>9.1</td>
<td>A Sample Storm topology</td>
<td>107</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Workload characteristics of existing GPGPU benchmarks 36
4.2 Application domains, behavior and implementation characteristics of Valar 39
4.3 Input arguments for Valar’s benchmarks 39

5.1 OpenCL event states 50
5.2 The main components of Haptic and their role in heterogeneous applications 52

6.1 Configuration parameters for K-means clustering 67

7.1 Device configuration of the Southern Islands GPU modeled in this thesis 80

8.1 Target platforms utilized for Haptic and Valar evaluations 86
Chapter 1

Introduction

1.1 Introduction

The recent trend in processor architectures has aggressively moved from homogeneous multicore systems to heterogeneous many-core systems. Heterogeneous systems in this context refers to a system with different types of computational resources [40, 41, 111]. A heterogeneous system could consist of a CPU (commonly known as a host device) and a graphics processing unit (GPU) or other computation accelerator such as field programmable gate arrays (FPGA). The rate of introduction of new heterogeneous systems and applications has increased rapidly across a range of different application domains such as high performance computing (HPC), data mining and image processing.

Applications targeting heterogeneous systems are commonly programmed using Nvidia’s Compute Unified Device Architecture (CUDA) [2] or the Open Compute Language (OpenCL) [41]. OpenCL and CUDA are programming models built using C and C++. In OpenCL and CUDA, the data parallel and computationally intensive portions of an application are offloaded to the accelerator device. OpenCL and CUDA provide language and runtime support that allows an application developer to rewrite the data parallel portion of his application to take advantage of the accelerator device. OpenCL and CUDA lets the application developer to decompose the parallel portions of his computation into a set of kernels. Kernels consist of independent units of work known as workgroups. Workgroups are executed in parallel on the different compute units of the accelerator device. Thus, CUDA and OpenCL follow a master-slave style of programming model, where the host-device (i.e., the CPU) can be considered to be the master since it controls data movement and the programs executed by the accelerator device (i.e., the GPU) can be treated as a slave. Popular applications in scientific computing [90, 41], molecular dynamics [83], graph analytics [53] and medical imaging [93] have been redesigned to take advantage of heterogeneous systems.

This offload-based or master-slave programming model allows an application developer to focus on performance optimization of the computationally intensive kernels.
The master-slave programming model used by CUDA and OpenCL benefits applications where the performance critical code is restricted to well-defined regions of code possessing a high degree of parallelism. The performance of such applications is not impacted by data movement and host-device communication, since a majority of the application typically executes without interruption on the accelerator device [45, 41, 90]. The master-slave programming model can target heterogeneous systems using presently available system-software since the low level control of the device is abstracted from developers by the graphics driver [2]. Abstraction provided by the graphics driver enables application developers, to view the process of launching of a kernel on an accelerator (slave device) similar to issuing a subroutine call.

In presently available heterogeneous systems host-device interaction can only be controlled by application developers at a kernel granularity. Interaction between a host and device is handled by the graphics device driver and can only be controlled by an application developer at a kernel granularity. This limitation is imposed by programming models such as OpenCL to simplify application development by allowing the programmer to abstract the computationally intensive portions of his algorithm into kernels. This limitation is also beneficial since it reduces driver overhead. However, a number of modern applications such as data-mining, social networking analytics and web-search are data driven and communication intensive [38, 56]. This makes workloads challenging to express in CUDA or OpenCL without extensive additional language and runtime support [13, 25, 23, 120]. When utilizing a master-slave model on heterogeneous devices, it may be difficult to obtain benefits from a heterogeneous architecture if there is no computationally intensive bottleneck in the application. A number of applications could impose timing and/or throughput requirements.

Due to the limitations of the master-slave programming model, some of the applications discussed above target heterogeneous systems using an implementation technique called “persistent kernels”. A persistent kernel implements computation within a single kernel containing an infinite loop [108]. The infinite loop executes on the accelerator device choosing a new unit of work by atomically reading the head of a global memory list on each iteration. The infinite loop exits only when no more work is available. A persistent kernel implementation is illustrated in Figure 1.2. As shown, In Figure 1.2 a fixed number of workgroups (equal to the number of compute units on the device) atomically extract computation from a queue. This software based implementation methodology does not require modifications of the architecture or the existing software stack of the accelerator [108]. However, this method is only scalable if there is no frequent host-device communication needed or any variation in offloaded computation. Software only persistent kernel designs methodologies are also highly dependent on device parameters like compute unit counts,

\[\text{1}\text{Data driven refers to data dependent performance and continuous behavior where input is continuously streamed from external sources such as databases and sensors to the device}\]
atomic operation performance and workgroup scheduling policies. These factors can greatly restrict application portability. Software-only persistent kernel implementations are additionally complicated if the application involves data movement between external sources (e.g., DBMS systems, sensors etc.) and the compute device. The limitations have been summarized in Figure 1.2. Software only implementations of persistent kernels on presently available heterogeneous devices are also oblivious to QOS and are unable to support priority between different workgroups running on the accelerator device. The absence of priority in scheduling has been a barrier in the utilization of heterogeneous devices such as GPUs in online systems where multiple streams of computation have different bandwidth and latency requirements [61, 89, 108].

This thesis was motivated by the evaluation of novel workloads targeting heterogeneous systems. The workloads addressed (based in analytics and data mining) have a high degree of host-device communication. The host-device interaction in our studied workloads is dependent on the algorithm of the workload, its mapping to heterogeneous devices and its usage scenarios within our applications. The workloads have been implemented with the flexibility to vary the interaction between the host and compute devices present on heterogeneous systems. Some of the workloads have been implemented using the persistent kernel methodology (Figure 1.2). In order to evaluate the workloads, multi-layered application models targeting heterogeneous devices have been developed. These application models are an important contribution of this thesis. Application models allow us to evaluate similar / differing behavior patterns across applications. Application models also allow us to evaluate similar / differing behavior patterns across different architectures. This is also applicable for heterogeneous applications targeting different accelerator devices whose behavior can be studied at different layers of abstraction.

Our study of the behavior of different heterogeneous applications has motivated us to propose architectural enhancements to the host-device interface for CPU-GPU heterogeneous systems. These architectural enhancements have then been leveraged in runtimes targetted by application developers. By proposing architectural enhancements to the host-device interface for CPU-GPU based systems, the limitations discussed for applications based on persistent kernel implementation can be alleviated. The architectural innovation proposed in this thesis consists of the development of a resource known as the Offload Control Unit (OCU). The OCU provides architectural support for profiling and workgroup scheduling on future heterogeneous platforms. The OCU can be implemented as an hardware or software resource. The OCU provides architectural support that will enable new classes of applications to take advantage of heterogeneous systems.

The Offload Control Unit (OCU) is an architectural resource that decouples the host offload of computation to the accelerator and the execution of compute kernel on device. This decoupling is beneficial for applications where work is generated at a granularity that
is smaller than a compute kernel which would utilize the complete device. Such applications are commonly called continuous applications. Continuous applications demonstrate streaming behavior where work is continuously offloaded to a compute device. Sample continuous applications in data mining and social networking include 1) tracking trending topics 2) reporting the frequency of words or metadata such as hyperlinks [68, 69, 110]. Such applications can only be implemented using the persistent kernel methodologies with the limitations shown in Figure 1.2. The OCU and the proposed runtime support simplifies development of such applications and increases portability without losses in performance.

The OCU provides architectural support for scheduling non-homogeneous distributions of workgroups on compute units of accelerator devices. The OCU also enables online performance monitoring of compute devices in heterogeneous systems. Leveraging the scheduling and performance monitoring capabilities of the OCU in software runtimes enables more applications to take advantage of heterogeneous devices.

To summarize, the Offload Control Unit (OCU) expands the applicability of heterogeneous devices by increasing the performance awareness and control over scheduling execution to compute units. The OCU provides a graceful interface between computation on the host and accelerator devices.

This introductory chapter discusses the evolution of the science of performance analysis from single threaded CPUs, to multicore CPUs with shared memory and finally to heterogeneous systems (Section 1.2). Applications targeting heterogeneous devices are discussed in Section 1.3. The OCU is described briefly in Section 1.5. Finally, the main contributions of this thesis are covered (Section 1.6).

1.2 Performance Analysis History

We initially provide a brief history of performance analysis due its importance in our evaluation of heterogeneous systems. Performance optimization of an application has been a challenging long-term research goal in computer architecture [34]. As shown in Figure 1.1, due to Moore’s law, the number of transistors on a chip has been increasing. However, on chip clock frequencies have remained static due to increasing power density [95]. Since, the frequency of transistors has not substantially increased, there is no free application speedup as applications migrate to modern CPUs. Optimization of software to efficiently utilize the transistors available on the chip is critical to scalable performance.

Performance analysis of a single threaded application involves the process of reducing the execution time of a single thread executing on a CPU core. Optimization of a single threaded application on a processor has leveraged improved execution frequency, deeper pipelines and speculative execution techniques. Single threaded performance optimization for out of order and superscalar processors has usually concentrated on quantifying the

\(^2\)Free refers to application speedup caused by increasing chip frequency
loss in performance due to the finite capacity of architectural resources [36]. Based on the performance bottlenecks detected, architectural improvements have been suggested [36].

On modern CPUs, performance improvement of a single-threaded application is restricted due to architectural design complexity and limited increases in clock frequency [95]. This has led to the advent of multicore platforms where multiple CPU cores are placed on a single die of silicon. These cores commonly share a memory address space and are targeted with multithreaded workloads [18, 118]. Multithreaded programs have become prevalent in a number of application domains such as scientific computing, computational fluid dynamics (CFD), data mining, and artificial intelligence. Performance analysis of applications executing on multicore systems extends single-threaded performance optimization, since the optimization process now requires performance analysis of a single thread on a core and also resource contention between multiple CPU cores [48, 49]. In the context of performance optimization on multicore CPUs, resource contention refers to the effect of shared resources between CPU cores, such as the cache hierarchy. However, multicore CPU performance analysis is specific to a single ISA and platform since all the CPUs in a multicore system are the same [46].

![Intel CPU Trends](image)

Figure 1.1: Scaling of transistor count, clock speed and the TDP of Intel x86 processors. While transistor counts have continued increasing, the power and clock frequency have stabilized since 2000 [49].

Thus, performance analysis for applications on heterogeneous systems is additionally
complicated due to the variety of ISAs used and data movement strategies for accelerator devices. The accelerator resource such as a GPU can be connected to the CPU by either a PCIe bus (with separate address spaces), or the accelerator resource could be integrated onto the same die as the CPU and share system memory. A wide variety of accelerator devices such as FPGAs or DSPs can be targeted with the same application.

The CUDA programming model to target discrete GPUs is a low level programming model where data movement between the host and accelerator has to be explicitly specified by the application developer. CUDA has been widely accepted due to low cost of discrete GPUs and the high performance that can be obtained from optimized applications. However, since CUDA-C is specific only on Nvidia GPUs, performance analysis of CUDA applications targeting heterogeneous systems is restricted to a single platform combination (CPU + Nvidia GPU) [58, 66, 65, 90].

1.3 Heterogeneous Applications

Popular applications in domains such as scientific computing [90, 41], molecular dynamics [83], graph analytics [53] and medical imaging [93] have been redesigned to take advantage of heterogeneous systems. A majority of these applications have been implemented for discrete GPUs using Nvidia’s CUDA programming environment [2].

The CUDA programming model introduced by Nvidia has enabled application developers in a number of software domains to take advantage of discrete GPUs to accelerate the compute intensive portions of their applications. Targeting an application to utilize a discrete GPU as an accelerator device entails parallelizing the compute intensive portions of the algorithm. The goal of the parallelization effort is to utilize the Single Instruction Multiple Threaded (SIMT) execution units on the GPU. CUDA applications targeting discrete GPUs have demonstrated substantial performance improvements in different domains due to the large numbers of compute units and memory bandwidth [53, 58, 90].

Motivated by the success of CUDA, the OpenCL standard proposed by the Khronos group has been released [41]. OpenCL is a vendor neutral low level programming model similar to CUDA. However, OpenCL is supported across vendors and can target multiple different computational accelerators such as GPUs from within the same applications. This standardization of a programming API like OpenCL for heterogeneous computing has increased interest among a wider range of developers. A vendor neutral programming standard is commonly favored by application developers. A number of popular CUDA applications have been ported to OpenCL to take advantage of the portability provided by the standard [24]. The ability of OpenCL to target the host-CPU as another accelerator present in the system, has led to additional performance benefits for selected applications [9, 75].
1.4 Application Performance Optimization for Heterogeneous Systems

Improving application performance is the primary motivation of using heterogeneous devices. CUDA and OpenCL are low-level programming models where the programmer has to explicitly specify data movement and manage thread granularity. Optimizing applications on heterogeneous devices is a challenging task. The goal of optimizations is to effectively utilize the memory bandwidth and large number of processing elements available on the GPU.

Performance analysis of applications executing on heterogeneous systems is complex due to the architecture of the GPU and the low level programming models. The performance analysis tools designed for heterogeneous systems also require an understanding of performance information specific to a single architecture (e.g., an AMD discrete GPU or an Nvidia discrete GPU). This complicates the development of performance-portable software. This highly architecture specific performance analysis methodology contrasts sharply with CPU platforms where performance optimization can be carried out across vendors using tool specific interfaces such as PAPI [109]. Such interfaces for GPUs from different vendors do not exist.

Since the introduction of CUDA and OpenCL, the computationally intensive portions of an algorithm has been offloaded to the GPU [57, 58, 93]. In discrete GPU applications large portions of the algorithm execute on the GPU compute device. The large benefits provided by the GPU device and low involvement of the CPU has caused a lack of interest in the performance of the CPU device. However, OpenCL allows kernels to be targeted to the CPU thus providing systems with a multicore CPU, an extra compute device. Treating extra CPU cores as another compute device has demonstrated performance benefits for applications with low communication costs [9, 103].

Modern heterogeneous systems released by AMD and Intel where the CPU and GPU device share a single die has lead to an increased interest in the effect of the host CPU on overall application performance. Such systems are commonly known as APUs (Accelerated Processing Units). On an APU device the CPU and GPU device have a shared address space without the PCIe data-transfer bottleneck. Placing devices on the same die has reduced the cost of data movement and motivated recent research to improve application performance by focusing on improving CPU utilization [8, 121]. The compute capability of the CPU and GPU devices in APUs is comparable to each other. This fact has complicated performance optimization since offloading large chunks of computation to a single device as done in discrete platforms could lead to lower utilization of the system as a whole [102].

Due to the complexity of performance optimization, OpenCL kernels and the compute device are being abstracted away from application developers using domain specific languages (DSL) [23] and libraries [15, 74]. Compilation of a DSL or linking with external
CHAPTER 1. INTRODUCTION

Figure 1.2: Comparing the master-slave model for heterogeneous processing (Case 1) with persistent kernel approaches (Case 2). In Case 1, an OpenCL workgroup executes a task and terminates since a known number of workgroups can be launched by the host device. Case 2 shows a software implementation of a continuous kernel in OpenCL where an unbounded number of work can be continuously executed on the device in a single kernel call.

libraries allows application developers to invoke code optimized for the compute device. However, modularization makes tuning applications harder [62]. This issue is especially relevant for cross-platform specifications such as OpenCL, which target different classes of devices using the same program. Without language support, once an application has been implemented, specializing it usually requires source code access. When calling an application interfaced with a library through an API, it is difficult for the caller to specialize the underlying library by tuning parameters or modifying underlying functionality. Also, due to device specific optimizations, specialization usually requires deep understanding of hardware and source code access [15].

1.5 The Offload Control Unit

The OCU introduced in Section 1.1 has been motivated by continuously streaming workloads and the capability of recent GPUs to concurrently execute workgroups from different
compute kernels. This capability has enabled parallel applications where smaller units of work are continuously offloaded asynchronously to operate on data from a range of different sources such as databases. These changes in accelerator architectures and application behaviors have increased the importance of scheduling workgroups of on compute units. However, GPU architectures have not provided additional support for scheduling workgroups with varying resource requirements and priorities.

Management of compute devices requires low overhead profiling subsystems to gauge the utilization of a compute unit’s resources (e.g., registers, wavefront pool entries, local memory etc.) at runtime. Architectural support for scheduling and resource allocation has not been evaluated for throughput device such as GPUs when executing multiple NDranges. This gap in the state of the art research has been present due to the nature of HPC applications of GPUs. In HPC applications, compute kernels offloaded usually contain enough work groups to efficiently utilize all machine resources \[24, 28\].

Initially, the role of the OCU is shown in Figure 1.3. The host offloads execution and data movement commands to the OCU. The proposed architectural counters on the compute units provide utilization information to the OCU. The timing and partitioning constraints imposed on the kernel can be used to guide the OCU’s mapping of workgroups to different compute units. The usage scenarios of the OCU studied in this thesis includes architectural support for continuous applications and support for performance aware middleware. The contributions have been further elaborated in Section 1.6.
1.6 Contributions of this Thesis

The key contribution of this thesis consists of the architectural innovation within profiling and scheduling subsystems, leading to the OCU. Another key contribution includes the workload modeling methodologies for heterogeneous platforms.

These contributions improve performance monitoring capabilities and provide architectural support for irregular and continuous applications targeted to heterogeneous systems. The main contributions of this thesis are described below in more detail.

1.6.1 OCU: Architectural Enhancements for Profiling subsystems

The OCU includes architectural extensions to profiling subsystems of heterogeneous platforms. The OCU profiling extensions can be utilized by application runtimes to be aware of their performance and usage.

Support for Irregular and Continuous Computation Applications

The OCU provides architectural support for irregular and continuous applications targeting heterogeneous systems. By decoupling the offload of computation from the start of computation and by adding support for scheduling workgroups, irregular applications with different kernels can be continuously executed without consuming a host’s CPU resources for compute device management.

Latency and Throughput Aware Queue Management

A number of applications studied in this thesis have time constraints have components with different timing constraints. A batch component where larger chunks of data are updated and a latency component with smaller high priority updates to data \[68, 69\]. Such applications have usually targeted heterogeneous devices by time slicing the usage of the compute device \[61\]. Other real time computation efforts targeting heterogeneous systems support batch execution behavior as seen in encryption systems \[115\].

The usage of the OCU, the request signatures and compute unit capacity identification system can be demonstrated on heterogeneous systems.

Architectural Support for Performance Aware Middleware for Heterogeneous Systems

Support for continuous computation enabled by the OCU could be provided by software and a dedicated CPU device whose sole task is offloading computation. However, a CPU based implementation would prevent the utilization of profiling capabilities added to the OpenCL compute devices. The information provided by counters on compute devices can be used
as request signatures and have a range of applications such as capacity identification, anomaly detection and scheduling guidance.

As an initial example, the Haptic OpenCL extensions have been presented as an implementation of performance aware middleware for heterogeneous systems. Architectural support for Haptic would utilize performance information stored in the OCU and generated by the hardware counters proposed. The architectural support would improve the software only support available in Haptic for runtime specialization of heterogeneous compute pipelines.

Request signatures provide indicators of an application’s properties but cannot be built online and do not provide utility during program execution. To enable online profiling for heterogeneous systems, we extend the usage of heartbeats to heterogeneous systems. Timestamps have served as heartbeats to track application performance. By providing architectural and middleware support for feeding the utilization statistics stored in the OCU, metrics like heartbeats can be developed to track progress or performance of a heterogeneous application.

1.6.2 Applications of Feature-Classification on Heterogeneous Applications

Benchmarks for heterogeneous systems can have a wide range of possible behaviors ranging from traditional offload intensive GPGPU benchmarks to closely coupled CPU-GPU interaction. The profiling architectures and the application classification proposed have been applied to workload characterization. The Valar benchmarks have been designed from a set of heterogeneous applications which exhibit a range of behavior on heterogeneous systems based on input parameters, datasets and targeted compute devices.

Host-Device Interaction Optimization Studies

GPGPU optimization has focused on improving memory performance. However optimizing a GPU kernel and its memory access patterns is sufficient only for applications with a limited number of kernels where a bottleneck is clearly visible. The benchmark suite developed as part of this thesis enables the study of the effect of host optimizations and data preprocessing steps (e.g, sorting) on OpenCL kernel performance across different applications.

Benchmark Development methodology and Targeted Benchmarking for Workload exploration

Heterogeneous applications can be classified at multiple layers of abstraction. This classification of benchmarks allows for generation of test cases based on desired features rather
than specifying input sets (e.g., ref, test and train inputs in SPEC [50]). The features allow us to develop an API for the generation of benchmarks that generate specific behaviors on their target platform.

Our workload characterization efforts extends input selection research [17, 33] for CPUs, where the behavior of a fixed number of inputs for a benchmark is examined with respect to behavior of inputs used in real world scenarios. Input selection is important in characterizing heterogeneous applications, where there are multiple real world scenarios of a heterogeneous application. Valar allows flexibility w.r.t. input cases and the application’s execution space. Thus, classification of workloads at different abstraction levels assists benchmark developers when converting a complicated real world application into a benchmark.

1.7 Organization of this Thesis

The thesis is organized as follows. Chapter 2 discusses related work and provides the necessary background information. In Chapter 3 we discuss the architectural features we use for classification of heterogeneous applications. In Chapter 4 we discuss the Valar benchmark suite and other motivating benchmarks such as the Storm continuous computation framework. In Chapter 5 we discuss the profiling architectures that we use to generate features and their application scenarios in online systems. Chapter 6 discusses the ongoing work of architectural support for profiling proposed by this thesis and the its application in heterogeneous computing. Chapter 8 discusses our proposed experiments and preliminary results. Chapter 9 discusses the proposed work for the completion of this thesis.
Chapter 2

Background and Related Work

2.1 Introduction

The high peak performance of heterogeneous systems has enabled scientists to examine computational problems and evaluate design spaces previously thought to be intractable \[41, 1, 83\]. As a result, applications leveraging heterogeneous architectures are greatly accelerating the pace of innovation in both, industry and research settings \[83\]. This adoption of heterogeneous computing among application developers has encouraged a large amount of research in improving performance and power characteristics of heterogeneous system \[42\]. The challenges in developing applications targeting heterogeneous systems has also led to research in programming languages and runtimes to improve developer productivity \[32, 40\]. This chapter covers background and related work in heterogeneous systems research. This chapter provides a brief overview of the following topics.

- the OpenCL programming model,
- currently-available commercial heterogeneous architectures,
- profiling approaches and support for performance analysis,
- languages and runtime for heterogeneous computing,
- architectural support for profiling, and
- benchmark suites and workload characterization for heterogeneous system execution.

2.2 OpenCL

OpenCL is an open standard maintained by the Khronos group which has received the backing of major graphics hardware vendors. An OpenCL program is comprised of two elements:
1. host code that executes on a CPU and is responsible for setting up data and scheduling execution on a compute device (such as a GPU), and

2. the code that executes on a compute device, called a kernel.

This architecture is shown in Figure 2.1. In-depth information on the details of implementing heterogeneous applications in OpenCL are provided in [41] and [77].

2.2.1 OpenCL Command Queues and Compute Units

Heterogeneous applications implemented using OpenCL commonly create a single command queue to access each compute device. Commands to enqueue an NDRRange or move data to device memory are passed to the device through the command queue parameter as shown below.

```c
cl_int clEnqueueReadBuffer( cl_command_queue command_queue,
                           cl_mem buffer, cl_bool blocking_read,
                           size_t offset, size_t cb, void *ptr,
                           cl_uint num_events_in_wait_list,
                           const cl_event *event_wait_list, cl_event *event)
```

OpenCL command queues can be managed as either in-order or out-of-order. An out of order OpenCL command queue does not guarantee any ordering between commands enqueued. In an OpenCL in order command queue, each command enqueued to the compute devices is completed before executing any subsequent commands. An in order command queue can be blocking or non-blocking with respect to the host. The blocking_read flag in the command above indicates if a read operation is blocking or non-blocking. If blocking_read is CL_TRUE (i.e., the read command is blocking), clEnqueueReadBuffer does not return until the buffer data has been read and copied into memory pointed to by ptr. If
blocking_read is CL_FALSE (i.e., the read command is non-blocking), clEnqueueReadBuffer queues a non-blocking read command and returns. The contents of the buffer that ptr points to cannot be used until the read command has completed. The event argument returns an event object which can be used to query the execution status of the read command. When the read command has completed, the contents of the buffer that ptr points to can be used by the application.

OpenCL platform model is shown in Figure 2.1 where a single command queue is mapped to a compute device. Modern CPU and GPU platforms have extended the platform model by adding the capability of mapping multiple command queues to the same physical device [1, 80]. Applications targeting discrete GPUs commonly utilize this capability to overlap data movement over the PCIe bus and kernel computation. The application developer is responsible for managing concurrency between data management and computation. Careful tuning can deliver significant increases in system utilization [1].

The ability to overlap independent computation across command queues is an extension of overlapping data movement. OpenCL version 1.1 introduced a subdevice extension which allows spatial partitioning of the compute units available on a compute device [41]. This extension was moved to the core specification in OpenCL 1.2. The benefit of mapping multiple command queues to a single device allows all of the data resident within the global memory address space of the same device to be addressable by all of the kernels. Also, mapping multiple queues to a single device provides tighter control over resource scheduling, which allows us to support priority-based command queues.

Nvidia Fermi GPUs were the first publicly available platform utilizing multiple queues for computation on a single device. The scheduling policy used by Fermi GPUs is similar to a left-over policy which schedules a complete kernel at a time [80]. A left over policy only schedules the next kernel if any compute units are available. Such a policy delays scheduling of smaller kernels [80]. Thus, it is inadequate for workloads with priorities between command queues and uneven workgroup distributions between the NDRanges. Nvidia’s Kepler GPUs improve concurrent behavior using their Hyper-Q technology [1]. However, workgroup scheduling heuristics in modern GPUs are not public and controlled by a complicated driver software stack. The CUDA stream API does not provide support to control allocation of compute units to streams or allow us to express priorities between different streams.

2.2.2 Domain Specific Languages

Due to the complexity of parallel programming, the OpenCL kernel and the compute device are rapidly being abstracted away from the application developer. Domain specific languages [23] and algorithm libraries [15, 97] for GPU computing have gained traction within their respective developer communities.

Domain-specific languages are gaining popularity for heterogeneous computing as a
means to simplify programming [23]. Combined runtime and language frameworks, such as Petabricks [7], allow us to express a range of different algorithms to solve a problem. The Petabricks API allows a programmer to write the same problem using different implementations. The auto-tuning component of Petabricks shows excellent promise for multi-core CPU performance, but has not discussed generating code for GPUs. Other domain specific languages include methods such as Teleport Messaging [110], a language construct in StreamIT (a stream programming framework) that enables sending control messages in a distributed memory environment. By using a synchronous data flow model, StreamIT programs benefit from powerful compiler optimizations. Our work in this thesis (described in Chapter 5) has a similar aim of injecting control information, but caters to a challenging environment of heterogeneous computing.

DSLs and libraries allow an application developer to simply call the library which runs the optimized code for the platform. However, it is well known that such library-based modularization makes application tuning more challenging [62]. This thesis aims to provide a framework that can be used to tune or specialize libraries targeting heterogeneous devices using domain specific knowledge.

### 2.2.3 Runtimes for Heterogeneous Systems

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 [37]. Sun et al. demonstrate software based support for task scheduling across CPU and GPU devices [107].

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 [9, 103] which are similar to equivalent CPU based frameworks like Intel TBB [87]. Profiling and specialization extensions discussed in this thesis are complimentary to such frameworks.

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 [89]. This software architecture prevents the GPU from being utilized for lightweight tasks and operating system services such as encryption [89]. Runtimes such as Bothnia allow accessing compute devices like GPUs without the overhead of the driver stack [26]. 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 kernel and user space [89].

One of the common goals of runtimes targeting heterogeneous systems is implementing data transformations to user provided data in order to simplify programming. The goal of data transformations is usually to improve data locality for memory performance or manage
host-device IO. Related work in runtime data transformations on GPUs is usually specific to molecular dynamics [120]. However, recent novel approaches in data transformations and application runtimes have begun utilize the CPU as a symbiotic resource to facilitate the execution of GPGPU programs on fused CPU-GPU architectures [121].

### 2.3 Current Heterogeneous Architectures

A summary of presently available heterogeneous architectures is provided below. The heterogeneous architectures discussed below include devices which can be targeted using the OpenCL programming model. Some of the devices listed have been used as evaluation platforms in this thesis. The classes of devices that support OpenCL execution are listed in Figure 2.2.

![Diagram of heterogeneous architectures](image)

**Multicore CPU**: Commonly available multicore CPUs can be targeted for OpenCL execution [41, 84]. Multicore CPUs can further be spatially partitioned into heterogeneous sub-devices. In this case, a subset of the total CPU cores available can be referred to as an "accelerator".

**Discrete Architectures**: These systems consist of a multicore CPU (the host) and a discrete GPU (the device). The CPU and the GPU have distinct address spaces, which usually necessitates explicit data movement between the host and the device (and vice versa). Data movement can be managed by the application developer or through a runtime layer [9, 43, 41].
Shared Memory Platforms: Recent heterogeneous platforms containing a CPU and a GPU device on the same chip have been released by AMD and Intel. Such devices usually share an address space and are commonly known as Accelerated Processing Units (APUs) \[33, 41\]. Bridge and AMD APUs

Manycore Platforms: Platforms containing a large number of smaller CPU cores have been announced. An good example of such a platform is the Intel Xenon Phi system which combines up to 64 x86-based processing cores. These systems do not require a separate host CPU device to act as a heterogeneous platform.

Reconfigurable and DSP Platforms: FPGA vendors such as Altera support OpenCL compilation on their reconfigurable platforms. DSP vendors such as Texas Instruments have also announced OpenCL support on their upcoming platforms. In this thesis, we have not studied reconfigurable and DSP platforms, mainly due to the fact that these system are just arriving on the market.

2.4 Profiling Approaches and Support for Performance Analysis

We can utilize observations of platform performance for a number of purposes. We begin by defining common terminology when using performance information. The goal is to clarify the key differences between monitoring, profiling and tracing as applicable to architectural research.

Monitoring refers to an ongoing and preferably automated process. Monitoring tools can be used to trigger alerts when a monitored resource falls above or below a trigger level \[3\]. Monitoring tools can have a number of applications in heterogeneous environments.

Profiling is usually done on a program to see which portion of the code is using the most resources. The resource metric commonly used is CPU time, but can also include memory usage, I/O bandwidth, and execution (wall) time. Profiling is commonly used to identify candidate code for optimization. Profiling tools tend to be language and/or platform dependent. Profiling can also be done offline using logs and/or monitoring data \[3\].

Program tracing involves collecting information that tracks the execution of a program. Tracing is generally performed in debugging scenarios. Example tracing tools include, system call tracing tools such as strace. A system call trace can help to determine why a program is failing or not responding as expected.

A more generic procedure call trace can also be applied to study code coverage and is commonly used in live debugging to view the previously called functions. However, tracing commonly requires a debug version of a program to be built \[3\].

This thesis develops monitoring and profiling capabilities specifically tailored to heterogeneous architectures.
2.4.1 Uses of Profiling

The benefits of profiling frameworks can be described with respect to the requirements of programmers, runtime developers and compiler designers. The main benefits of profiling have been covered in detail by Bone et al. [20].

**Application Developers:** Programmers need tools to help identify and understand the causes behind performance problems [20]. For example, communication between tasks can cause the task to be blocked while waiting for information. One task that blocks may delay other tasks that depend on data it produces. Chains of tasks that produce and consume values from one another are common on heterogeneous systems.

**Runtime Systems:** Profiling tools can also be used by the developers of a runtime system [20]. Some examples of the potential benefits of profiling for runtimes include: a.) when a piece of work is made available, how quickly can a sleeping worker-thread respond to this request, and b.) when a task becomes runnable after being blocked, how often will it be executed on the same CPU where it was previously executing, exploiting any caching effects. Information obtained through profiling parallel programs can be used to improve the runtime system, which can help improve the performance of all parallel programs [20].

**Compilers:** Profiling tools are also valuable to compiler designers when developing parallelization tools [20]. For the compiler to do a good job of parallelizing a program requires a cost-benefit analysis for each parallelism opportunity. Profiling tools can help researchers calibrate the algorithms they use to generate estimates of both costs and benefits.

2.4.2 Motivating Research in Profiling Architectures

Designing a profiling architecture requires a mechanism to gather power/performance statistics. Statistics collected by the profiling architecture can be leveraged to guide performance optimizations during application development. Performance statistics can also be leveraged at runtime as part of an autotuning framework. The present of a quality profiling architecture is fundamental to performance optimization since information fidelity is key to beneficial performance optimization.

Prior research in profiling architectures is limited due to requirements of architectural enhancements or hardware changes. The addition of profiling capabilities can be justified based on the performance optimizations enabled by profiling execution [91]. While hardware resources can benefit performance optimization, these resources have also been applied in a number of online scenarios such as anomaly detection, request signature identification [52], capacity measurement [67] and debugging support [41].

Utilizing profiling information at runtime to optimize performance of an application requires high fidelity performance data with minimal delay. A number of heterogeneous

---

1Online denotes application usage or deployment scenarios witnessed at runtime after development
implementations of algorithms are embedded to create computational pipelines. Specialization of computational pipelines targeting heterogeneous systems requires performance monitoring capabilities specific to devices such as GPUs. As discussed, the host device interaction affects performance.

2.5 Profiling Interfaces in Programming Models

To enable reuse of tool development efforts, a number of programming models have interfaces that provide profiling and performance information. Profiling interfaces also simplify performance analysis by application developers since the interface provides performance information in a consistent format across different target devices. For example, the OpenCL profiling interface dictates that all conforming devices provide timestamps with at least nanosecond resolution. A powerful profiling interface embedded in a runtime system can positively impact our effort to accelerate heterogeneous applications. In this thesis, we leverage the OpenCL profiling interface to demonstrate this potential. We discuss the class of profiling in more detail in Chapter 5.

2.5.1 OpenMP Profiling Interfaces

OpenMP programs present a challenge for performance measurement tools because of the OpenMP specification. The user-level execution behavior of OpenMP applications is described in the OpenMP specification. However, the OpenMP specification is similar to OpenCL in the fact that it does not specify implementation details. While this ambiguity provides flexibility to implementation vendors, it complicates the development of performance measurement tools, since tools record data based on the implementation-model.

The OpenMP Runtime Tracing API is designed to provide data for performance tools to build traces and present measurements for OpenMP programs. The recorded performance trace is mapped back to the user model to enable an application developer to optimize his/her program [55]. Many different tracing methodologies targeting OpenMP applications have been developed, along with a rich set of data-reduction and visualization tools. However, traces scale poorly with increased runtime if there is no compaction mechanism or throttling strategy managing data collection. If records are missed due to overruns, the dynamic program graph can not be generated. OpenMP tracing tools include Paraver, K0JAK, CATCH, TAU, and VAMPIR [39]. One tracing methodology, POMP, was proposed as a standard for OpenMP performance measurement [39].

OpenMP API's extensions have also been proposed towards statistical profiling. The goal of statistical profiling is building a dynamic call graph of an OpenMP program [55]. Statistical profiles have the advantage that they can scale very nicely for long program runs, simply by throttling the profiling rate. By capturing complete callstacks with each event, the dynamic callgraph behavior can be reconstructed [55].
2.5.2 MPI Profiling Interfaces

The extensive adoption of MPI as parallel programming model for applications targeting HPC clusters has lead to the development of a standardized profiling interface [99]. The MPI profiling interface uses the linker to substitute a user-written “wrapper” function for the MPI function called by the application. In order to do this, the wrapper must have the same name. Since the author of the profiling library does not have access to the MPI source code, the profiling version of the MPI function must have a way to call the original MPI function to carry out the required task. Every MPI function is callable by a different name. The MPI definition requires that every MPI function MPI\text{Xxx} also be callable by the name PMPI\text{Xxx}. For example, the profiling version of MPI\text{Bcast} can itself call PMPI\text{Bcast} to actually do the broadcast, thus performing whatever profiling work is desired before and after the call to PMPI\text{Bcast}.

The MPI standard specifies a mechanism by which all calls may be intercepted by the user, who can define a profiling library containing his own versions of the MPI functions. For example, distributed with the MPICH portable implementation of MPI is a profiling library to write records, logging the start time and end time of all MPI calls. This state can be viewed using visualization tools. The profiling version of MPI\text{Finalize} causes the local logs to be merged and the log file to be written. To enable profiling, one need only link the profiling library in front of the MPI library in use.

2.5.3 Other Profiling Tool Interfaces

The notion of custom counters is also present in managed runtime systems including the Microsoft .NET environment [119]. The Java Virtual machine provides counters known as “Java virtual machine counters” [4] within the virtual machine. The Java programming language also provides a Java Performance Counter Monitor (PCM). The Java Performance Counter Monitor’s goal is to extend performance counter monitoring code for Intel platforms and expose it’s functionality in Java via JMX. Unlike OpenCL, these counters are not vendor independent, and cannot be utilized in an online application.

2.6 Architectural Support to Generate Performance Information

Research in architectural support for performance analysis has a rich history that started with the addition of performance counters on a CPU [31]. The goal of profiling is to generate resource usage information that when represented visually can guide the user towards the resource bottleneck [31]. Performance information generated by profiling can also be utilized by compilers and continuous optimization systems [62]. Architectural support for profiling can include a variety of schemes based on hardware, software and hybrid extensions. The
goal of any of these methodologies is a tradeoff between overhead and the quality of the generated performance information.

**Hardware Schemes:** The goal of hardware based profiling schemes is to minimize profiling overhead and avoid instrumentation and recompilation of the target application. A coprocessor scheme where a simple CPU handles the profiling of the main computational CPU has been proposed [123]. Hardware based profiling extensions could also include ISA extensions where a bit is added to an ISA instruction to enable profiling [16]. Other proposed hardware extensions include, the profile buffer for the collection of profile information. The slowdown for the profile buffer is low and fulfills the profiling demands of an optimizing compiler by providing high accuracy [27]. Hardware support for profiling has also been proposed for modern multicore platforms in the form of monitoring resources that are distributed throughout a multicore system [117].

**Software Schemes:** Software support for profiling can be provided using instrumentation systems [64, 113]. The instrumentation framework inserts profiling instructions (e.g: increment instruction to count number of loop traversals). Purely software based methods introduce overhead and also perturb the system under test. For e.g: writing performance information to a memory location in cache frequently could affect the application’s miss rate. A number of software algorithm such as variational path profiling [82] and edge profiling exploit program structure to reduce overhead and interference. Variational path profiling deals with the different possible paths a program can take through a binary. It samples path executions, measures performance variations and ranks top varying paths in program.

**Hybrid Schemes:** Hybrid profiling schemes aim to provide the low overhead of hardware based profiling and the flexibility of software based methods. Hybrid profilers include programmable coprocessors that interrupt the main computational processors [123]. Hybrid schemes also include a combination of hardware counters and analysis tools [6, 22, 109]. Other hybrid profiling schemes are based on a stream compression profiling model [92]. In this model, the processor generates a stream of profiled events whose type is selected by software. The stream is a sequence of data tuples: for example, a value-profiling tuple contains the PC of a load instruction and the loaded value. Dedicated hardware compresses this stream before passing it on to software.

### 2.6.1 GPGPU Profiling

Due to the low level programming model for GPGPU applications where the application developer is responsible for parallelization and data movement, accurate performance information is required to guide optimizations.

To allow developers to optimize kernel performance, GPU vendors such as AMD and Nvidia provide profilers that return performance information. Device-specific performance of OpenCL kernels can be studied using these vendor provided profilers such as the AMD APP profiler [41]. These profilers query hardware performance counters on the device [41].
Kernel optimization studies utilize such profilers [24, 41, 58, 90] since the area of interest of such studies is optimization within a single compute kernel.

Commonly used HPC performance tools like TAU and PAPI have also been enhanced [65] by integrating GPU performance measurement into their measurement and instrumentation frameworks. These tools allow application developers to study performance analysis using traces built during program execution.

![Figure 2.3: Profile driven analysis of heterogeneous applications. Profiling subsystems could refer to the vendor provided profiler, OpenCL events or basic timing routines.](image)

Other profiling research for GPU devices consists of software based methods that focus on instrumenting activity in the memory hierarchy [11]. Performance information is generated using sparse compiler inserted instrumentation [11]. Software based profiling architectures for heterogeneous devices can also be based on binary instrumentation techniques to understand performance of the multiple components in the software stack such as the GPU driver [113].

Due to the closer integration of the CPU and the GPU device in platforms such as APUs, the computational capacity of the CPU and the GPU is more comparable than in discrete platforms. The optimization of the compute kernel is still important as in discrete platforms, however data management and scheduling is now an additional challenge for APU platforms. However, studying host-device interaction only requires high level performance information such as time and data size. When studying this behavior, this thesis uses OpenCL’s event cl\_event interface which allows querying statistics regarding OpenCL commands off the critical path with lower overhead [41, 73].

### 2.7 Benchmark Suites and Workload Characterization

One of the primary contributions of this thesis (see Section 1.6) is the design of a rich benchmark suite targeting heterogeneous architectures. We provide a brief overview of existing CPU benchmark suites which have been designed to evaluate a range of architectures from single core CPUs (SPECint & SPECfp) to multicore (PARSEC) and multithreaded
platforms (SPLASH2).

### 2.7.1 Single Threaded CPU Benchmarks - SPEC

Benchmarking on parallel systems has a rich history. SPECint and SPECfp \[50\] are two suites from the SPEC organization, an industry consortium focused on developing evaluation standards for benchmarking commercial systems. These two suites focus on single-core performance, specifically single-core integer and floating point performance.

### 2.7.2 Multithreaded Benchmarks - SPLASH2

SPLASH2 is a multithreaded benchmark suite designed to study multithreaded systems \[118\]. SPLASH2 caters to the study of centralized and distributed shared address-space multiprocessors. For homogeneous multiprocessor systems, SPLASH2 allows understanding of the interplay of problem size, number of processors, and working sets in designing experiments and interpreting their results.

### 2.7.3 CMP Benchmarks - PARSEC

A popular benchmark suite for Chip MultiProcessor (CMP) research is the PARSEC suite \[18\]. However PARSEC does not provide benchmarks that can be used for heterogeneous computing. The algorithms of the benchmarks implemented in PARSEC were not intended to scale beyond conventional CMP core counts higher than 16. PARSEC has 4 benchmarks whose performance is data dependent (dedup, freqmine, raytrace, x264) \[19\].

### 2.7.4 LMBench Suite

Lmbench is a micro-benchmark suite designed to focus attention on the basic building blocks of many common system applications, such as databases, simulations, software development and networking \[70\]. Lmbench can be used to compare different system implementations from different vendors.

### 2.7.5 Workload Characterization

Workload characterization is the study of the temporal and spatial characteristics when executing a program on a computational resource. The time varying behavior of a serial application can be studied using Simpoints. Simpoints provides automatic techniques capable of finding and exploiting the Large Scale Behavior of programs (i.e., behavior seen over billions of instructions). For single-threaded execution, Simpoints provide a hardware independent metric (basic block vector) that can concisely summarize the behavior of an arbitrary section of execution in a program \[100\]. Workload characterization research on
multicore systems uses architectural counters to study the performance of multicore applications [18]. Workload characterization studies usually focus on a single characteristic of an application, such as the synchronization for critical sections [35].

GPGPU workload characterization [12] studies examine the effects of architectural modifications on OpenCL / CUDA kernels. Past workload characterization studies for discrete devices have used low-level counter information that can be obtained using simulators or native platforms [12, 112]. Characterization of a GPU memory system can be carried out by sparse instrumentation [11] and subsequent modeling. These previously proposed workload characterization methods for discrete GPUs complement our analysis for heterogeneous workloads. Gregg and Hazelwood [45] examine how the location of data affects performance in heterogeneous platforms. Due to the cost-efficiency of GPU accelerated clusters prior work has characterized workloads that utilize multiple discrete GPUs present in many modern HPC clusters [94].

Unlike workloads on discrete platforms, workload performance targeted for a shared memory platform such as a APUs is impacted to a larger degree by the CPU device. The importance of interaction between heterogeneous devices is highlighted in recent research, where the role of the host CPU was evaluated [8]. Spafford et al. have recently studied the low-level architecture of APU devices using the SHOC benchmark suite [102].

2.8 Summary

In this chapter we have presented a range of background discussion. We have discussed OpenCL and heterogeneous applications. To gain an understanding of the possible targets of this thesis, we have presented an overview of publicly available heterogeneous architectures. Since this thesis extends the OpenCL profiling model, we have discussed tool-interfaces in other popular programming models such as OpenMP and MPI. One of the fundamental contributions to this thesis is a benchmark suite and workload characterization. To provide the appropriate background, we have evaluated CPU benchmarks and discussed the state of the art in workload characterization for heterogeneous architectures.
Chapter 3

Feature Based Heterogeneous Application Models

3.1 Introduction

Applications targeting heterogeneous devices are commonly executed using complex software stacks. Software stacks for compute devices such as GPUs commonly include multiple usermode and kernel-mode components, such as runtimes implementing a programming API and device drivers. Device drivers for GPU handle a number of low-level device management tasks such as initialization of the device, transfer of programs to the device and initialization of the kernel to be executed. In closely-coupled applications that require frequent host-device communication, such software stacks can affect performance and complicate end-to-end performance analysis [89, 108, 113].

One of the fundamental goals of this thesis is proposing architecture-level features that allow irregular applications with frequent host-device communication to leverage the benefits of a GPU. This necessitates developing a new model that can be used to describe the behavior of an heterogeneous application. The goal of designing our application model is to define the behavior of an heterogeneous application in order to describe behavior and study performance from an end to end perspective\[1\]. An application model can be used to study performance. Our model will capture behavior that can be described at multiple layers of abstraction. For example, behavior can be expressed in terms of high-level application throughput (e.g, records processed per second) or low-level memory statistics (e.g, cache hit percentage or memory bandwidth).

This chapter presents our contribution to building performance models for heterogeneous applications at different layers of abstraction. The behavior of heterogeneous applications

\[1\]In this work, the behavior of heterogeneous applications is not restricted to only offloaded compute kernels
is discussed at multiple layers due to the number of similarities and differences between applications and architectures. For example, a number of heterogeneous applications (e.g., linear algebra solvers) could have similar behavior with respect to host-device communication. On the other hand, a number of heterogeneous architectures are similar and simply contain different numbers of the same architectural components (e.g., different Nvidia Fermi GPUs with varying memory clock rates and core counts). Such a large design space necessitates models that can be used to express behavior of target applications. The performance analysis of host-device interaction discussed in this thesis is complementary to analysis of the compute kernel enqueued on the device [57, 58, 90].

3.1.1 Role of Workload Models

Modeling workload behavior has a range of applications in architectural research and software development. The common uses of workload models are described below.

**Architecture Independent Application Analysis:** Workload models are commonly designed to study application properties independent of the target architecture. For example, the number of iterations of computation needed to achieve a fixed error in a linear algebra solver is independent of the target architecture. In a parallel application, data reuse and inter-thread interaction is a function of the software implementation and should be similar across different architectures [63, 71]. An example of architecture independent characteristics of an application targeting heterogeneous devices would be the number of kernel calls for a fixed input size [74].

**Improving Simulation Performance:** Workload models can be used to improve simulation efficiency in architectural design space exploration. The slowdown of cycle-based architectural simulation versus native execution can be as high as 10000x. This level of slowdown severely limits design space exploration of new architectural designs.

Modeling methodologies applied to single-threaded workload have shown that there exists repetitive periodic behavior in programs [100]. These repeating patterns are commonly known as *application phases*. Characterization of these repetitive phases in applications has been used to improve simulation performance [100]. The results of characterizing benchmark execution using phases are shared among architects in the form of tools such as Simpoints. Simpoints are used by architects to speed up simulation across repeated regions of benchmark execution [100].

**Guiding Optimizations and Scheduling:** Workload models in combination with runtime support have been used to manage execution and guide runtime optimizations [79]. Application runtimes have leveraged the periodic nature of application behavior phases to implement online optimizations such as data-path specialization and voltage/frequency scaling [13]. Application models and phase awareness has also enabled remote monitoring and optimization of deployed applications [79].
3.2 Multi-Layer Workload Models

Performance analysis of an application can be carried out using multiple layers of abstraction. The simplest method to gain insight into performance of different sections of an application would be to instrument an application with timer code and discover the “hot” code [113]. At the microarchitectural layer of abstraction, performance information is correlated with the utilization of architectural resources (e.g., memory bandwidth utilization, cache hits, TLB misses etc.). The resultant counter statistics are compared with respect to their ideal values (e.g., 0% cache miss rate) to understand resource utilization. This analysis of hardware counter statistics can guide an application developer to decide the bottleneck resource in his application.

Performance analysis methodologies based on architectural counters have been used to optimize GPU applications. Performance information is generated by querying hardware counters [2]. Profilers such as the Nvidia Nsight and AMD’s Code XL are provided by their respective vendors. The profilers sample performance counters values using the device driver. For a discrete GPU, performance analysis of CUDA applications has been complicated by the SIMT nature of the GPU and the low-level programming models. An HPC application usually transfers data to the GPU device memory and then launches long running kernels which operate on the input data. The large degree of available data parallelism allows for uninterrupted execution on the compute device. Applications targeting discrete GPUs, especially in the HPC domain, minimize host-device interaction. Thus, optimizing compute kernel performance in such applications would usually lead to the maximum possible application-level speedup [24, 58, 90].

![Figure 3.1: Abstraction layers for Studying Heterogeneous Applications. An application can be studied based on its input arguments, host-device interaction, platform independent metrics and hardware counters.](image)

Discrete GPU applications are usually targeted to run on a single-device architecture. The CUDA programming model is used in HPC to run applications on Nvidia GPUs. As mentioned in Chapter [1] the applications discussed in this thesis have behavior where the host and multiple compute devices can communicate frequently. Such applications can be
run on a large range of devices if we move our coding to OpenCL. Due to using a tiered software stack and the presence of a range of possible behaviors, heterogeneous application now need to be studied at different layers, as shown in Figure 3.1. This thesis describes a novel mechanism that uses multiple layers of application behavior measurements specifically chosen to capture host-device interaction, as well as high throughput kernel computation on the compute device. The layers are briefly listed below:

- **AL0 Features**: Input arguments
- **AL1 Features**: Host device interaction metrics
- **AL2 Features**: OpenCL kernel invariant metrics
- **AL3 Features**: Hardware performance counter statistics
- **AL4 Features**: Compute Unit statistics

Figure 3.2: **AL0 and AL1 abstraction levels for the study of Heterogeneous applications.** Performance of diverse applications can be compared with respect to their host-device interaction.

### 3.3 AL0 Features

The **AL0 features** simply capture the input parameters of an application. The **AL0 features** of an application represents the higher level of abstraction since **AL0 features** are based solely on the application. This level was chosen as 0 since it provides a baseline layer for studying behavior.

The **AL0 features** provide us little room for flexibility with respect to performance optimization. For the applications considered in this thesis, the input arguments are shown in Table 4.3.
CHAPTER 3. FEATURE BASED HETEROGENEOUS APPLICATION MODELS

3.4 AL1 Features

The AL1 features describe data movement frequency and the time taken by a kernel to execute on an OpenCL-compatible compute device. The AL1 features (shown in Figure 3.2) have been used to build a model of the host-device interaction.

For programs targeting heterogeneous devices using OpenCL, host-device interaction can occur in the form of kernel launches, data transfer calls and data mapping calls. The AL1 feature set consists of IO and kernel call frequency, which can be used to build a model of the host-device interaction and compare the host device behavior of different OpenCL applications. The study of AL1 features enables the exploration of an optimization space based on data movement and kernel launch frequencies. For example, write combining of host-to-device data transfer and grouping OpenCL kernel launches are optimizations implemented using AL1 features. The benefits of AL1 optimizations can be compared across different OpenCL capable architectures (CPU compute devices vs GPU compute devices).

The Valar benchmarks described in Chapter 4 provide flexibility in exercising host-device behavior. AL1 features can be controlled to a large extent by choosing the appropriate input arguments to the benchmarks. The OpenCL level profiling framework Haptic discussed in Chapter 5 allows us to explore the host-device interaction space.

3.5 AL2 Features

The AL2 feature set is the first abstraction layer that discusses the offloaded kernel execution. AL2 features denote OpenCL kernel metrics that are invariant to execution on different compute devices. “OpenCL kernel metrics“ refers to OpenCL kernel statistics that can be expressed using terminology based on the OpenCL specification.

![Figure 3.3: OpenCL Compute Device Model showing the AL2 Level Features. These features are metrics that are the same for any OpenCL kernel irrespective of the actual compute device chosen for execution.](image)

The AL2 model statistics proposed in this thesis are gathered using simulation-driven studies [112]. AL2 level statistics can be gathered using functional simulation where abstract behavior, such as the ALU:memory access ratio and instruction mix, can be calculated.
Gathering AL2 level statistics on a native platform can be considered to be similar to the preset events in the PAPI hardware counters [109]. Preset events are populated by post-processing native performance counters. CPU architectures supported by PAPI’s preset events target x86, which simplifies generating performance information across different CPU platforms.

Performance modeling can build an abstract representation of a compute kernel [10]. Thus, the AL2 features can be described as a functional model or an abstract representation of an OpenCL kernel. The AL2 features can be explained with the help of Figure 3.3.

3.6 AL3 Features

The AL3 abstraction layer is the first abstraction that discusses OpenCL kernel performance in terms of its execution a specific architecture (e.g: Nvidia Fermi GPUs or AMD’s Southern Island GPUs). The AL3 features shown in Figure 3.4 display the execution results of the OpenCL kernels and the host-device transfer time. AL3 metrics of an OpenCL kernel include kernel execution time, but include architectural metrics provided by the hardware performance counter.

<table>
<thead>
<tr>
<th></th>
<th>AMD GCN Counters</th>
<th>Nvidia Fermi Counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector ALU Busy %</td>
<td>ALU Busy %</td>
<td>Coalesced Load %</td>
</tr>
<tr>
<td>Scalar ALU Busy %</td>
<td>Cache Hit %</td>
<td>Registers Used</td>
</tr>
<tr>
<td>Mem-Unit Busy %</td>
<td></td>
<td>Local Memory Used</td>
</tr>
<tr>
<td>Registers Used</td>
<td></td>
<td>Memory Bandwidth</td>
</tr>
<tr>
<td>Local Memory Used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.4: Hardware performance counters for AMD and Nvidia discrete GPUs. These counters are unique to an architecture and cannot be used to optimize applications in a cross-platform manner.

OpenCL and CUDA programs are usually optimized by targeting the architecture-specific AL3 metrics shown in Figure 3.3. Vendor-provided tools, such as Nvidia Nsight [2] and AMD Code XL, provide AL3 level features. Performance analysis of applications using only AL3 metrics has been well studied [90, 57] and an overview of performance optimization studies of GPU applications is discussed in Chapter 2.
3.7 AL4 Features

The AL4 abstraction layer has been proposed to study performance of an heterogeneous application within a compute unit on a heterogeneous device. The AL4 features have been defined in this thesis to provide performance statistics within a time interval for the OCU enhancements proposed in this thesis. The low-level and compute device-specific nature of the AL4 statistics makes them unsuitable for application-level optimizations.

The AL4 features (throughput counters) are highly specific to each architecture, since they are generated on each compute unit of a compute device. The AL4 metrics are discussed in more detail in Section 6.3. The AL3-level metrics differ from the AL4 metrics in the fact that AL3 metrics are calculated on a per-device basis over a complete OpenCL kernel, versus AL4 metrics which are calculated on a per time interval for each compute unit of a device.

3.8 Using Features to Build Heterogeneous Performance Models

The multi-layered workload modeling methodology discussed above can be used as a hierarchical model to study applications. As discussed in Section 2.2, the OpenCL programming model provides an abstraction of the underlying compute device’s architecture. When a OpenCL kernel is mapped to a compute device, local memory and global memory are stored in different memory spaces of the compute device.

Generating consistent performance information across heterogeneous devices the include different ISAs is part of our future goals. Metrics such as size of local memory accesses can be defined in a consistent manner across devices. Such high-level performance models can be used for performance optimization. At compile time, performance information can guide resource usage. At runtime, performance information can be used to schedule workgroups to avoid conflicts between resources such as cache memory.

In this thesis, we have proposed using workload models that capture metrics across multiple levels of abstraction. We have leveraged the workload modeling developed in Chapter 4 while designing our benchmark data sets. We have also applied our proposed models to optimize host-device interaction when using the Haptic framework in Chapter 5. We have also defined AL4 level metrics for the compute units of two AMD GPU architectures.

3.9 Summary

This chapter discussed the multiple layers of abstraction that can be used to study performance in heterogeneous systems. The features can be used to study performance trends in
OpenCL-compliant devices, with respect to host-device interaction and kernel characteristics. One of our future goals is to define AL2 level statistics which would serve as “virtual” counter interface \cite{4,109} such as PAPI for GPU devices.

Further chapters discuss our motivating benchmarks and the performance information generated. The applications of these features are discussed in the context of benchmarking in Chapter 4 and applied to profiling architectures in Chapters 5 and 6.
Chapter 4

Valar - Benchmarks to study the Dynamic Behavior of Heterogeneous Applications

Benchmark suites are commonly designed with the goal of exploring specific design spaces such as compiler optimizations or microarchitectural enhancements. The resulting performance impact of the microarchitectural or compiler design feature on the benchmark is used to establish the merit of the respective modification. Benchmark suites for multicore and GPU research can be used to explore these design spaces. To quantitatively evaluate the value of each design feature, computer architects rely on the representativeness of each benchmark. Representativeness is a measure of how close the benchmark suite represents real world applications.

Compiler and architectural research has motivated the development of benchmark suites to equip researchers with appropriate workloads to evaluate GPU-based systems [105, 42]. The advent of CUDA saw a number of benchmarks introduced to study the performance of Nvidia discrete GPUs [24, 28, 105]. Many of these same benchmarks have been ported to OpenCL. This allows the benchmark suites to target a larger range devices consisting of CPUs and AMD GPUs [24, 28, 105]. Designing benchmarks to target different heterogeneous systems is more complicated than designing benchmarks for multicore CPUs or a single discrete GPU platform. This is due to the large differences in system organization, including the memory system, core architecture, and communication interface.

Modern applications targeting heterogeneous devices have a range of possible host-device behaviors based on their design and usage. For example, an application based on its data set size and Quality of Service (QOS), could either be a latency-oriented application, with smaller independent units of work, or a more throughput-oriented application that work primarily on large data sets. Given this range of application behaviors, the characterization
task become more challenging. To deal with this range of behaviors, benchmark suites usually limit data-driven problems to a set of known inputs (e.g., small, medium and large). However, input size does not fully capture the dimensions of heterogeneity present. Fixing application inputs prevents a proper investigation of the data-driven characteristics of a benchmark and its effect on the architecture being studied.

To address limitations with previous benchmark suites, we propose a new set of benchmarks called Valar. The goal of Valar is to provide a set of applications that exhibit a wide range of host-device behaviors and data-sharing patterns. Recent application studies on key value stores [51] and databases [13] show the importance of studying host-device behavior on closely-coupled heterogeneous systems.

The applications in Valar are full-blown, real-world, applications that execute on OpenCL-compliant devices. To be compatible with available devices, these applications are based on the host-device programming model of OpenCL. As mentioned in Section 1.6, one of the goals of our research is to design architectural support to enable heterogeneous devices for newer families of applications. This chapter also presents a subset of the future applications that motivate our architectural research (discussed in Chapter 6).

In this chapter we discuss existing benchmark suites targeting heterogeneous computing in Section 4.1. We describe the characteristics of the Valar benchmarks in Section 4.2 and provide detailed discussion of these benchmarks in Section 4.3. We present some execution characteristics of the workloads on selected target systems in Sections 4.6 and 4.5. We then discuss future applications that should be added to Valar in Section 4.7.

4.1 GPGPU Benchmarks

Popular open-source benchmark suites targeting heterogeneous devices include Parboil [105], Rodinia [24] and the Scalable Heterogeneous Computing (SHOC) benchmark suite [28]. These benchmark suites fulfill roles similar to PARSEC [19]. SPEC [50] is another popular benchmark suite managed by an industry consortium.

Parboil, Rodinia and SHOC support OpenCL and also provide applications implemented in different programming models such as CUDA and OpenMP [24, 28, 105]. These benchmarks assist architects with the study of programming models and applications.

**SHOC**: SHOC [28] provides a range of low-level benchmarks based on scientific computing workloads and is unique in its support for GPU clusters.

**Rodinia**: provides representative real-world applications targeting GPU systems from multiple domains (e.g., data-mining and medical imaging) that can be run “out of the box” as target workloads for architecture research. While Rodinia does provide complete applications, the selected applications are taken from ports of existing CUDA workloads.

**Parboil**: The Parboil [105] benchmarks also provide workloads from domains such as data-mining and medical imaging. However, Parboil also provides workloads with serial
C++ versions and different source-level optimizations for CUDA, OpenMP and OpenCL workloads. This feature of Parboil allows compiler writers to compare hand-coded source code optimizations against compiler optimizations on different architectures.

**AMD APP SDK**: The AMD APP SDK provides simple applications that are commonly used as templates for OpenCL application development.

The benchmark suites discussed above have been designed to benchmark the “compute device”. These benchmark suites are useful to the computer architect seeking to evaluate architectural enhancements on a single compute device such as a GPU. Valar is focused on assessing both a host and a device, and includes more heterogeneous applications. The differences between these publicly available benchmarks and our goals for Valar have been summarized in Table 4.1.

<table>
<thead>
<tr>
<th>Parboil</th>
<th>Rodinia</th>
<th>SHOC</th>
<th>Valar</th>
</tr>
</thead>
<tbody>
<tr>
<td>• CUDA, OpenCL &amp; OpenMP</td>
<td>• CUDA, OpenCL &amp; OpenMP</td>
<td>• CUDA, OpenCL &amp; OpenMP</td>
<td>• OpenCL for heterogeneous systems</td>
</tr>
<tr>
<td>• Data driven examples</td>
<td>• Data driven examples</td>
<td>• Data driven examples</td>
<td>• Data driven examples</td>
</tr>
<tr>
<td>• Image processing, molecular simulation, fluid dynamics, and astronomy</td>
<td>• Based on scientific workloads, data mining, medical imaging and graphs</td>
<td>• Based on scientific computing and linear algebra workloads</td>
<td>• Varying interaction scenarios between hosts and devices for each application</td>
</tr>
<tr>
<td>• Multiple source optimization levels to evaluate optimizations, compilers &amp; architectures</td>
<td>• Complex application kernels useful for evaluating architectural enhancements of compute devices</td>
<td>• Supports GPU Clusters</td>
<td>• Applications for heterogeneous devices with shared memory</td>
</tr>
<tr>
<td>• Serial versions for parallelizing compilers</td>
<td>• Benchmark divergence study compares applications</td>
<td>• Provides workload partitioning for GPUs in clusters</td>
<td>• Applications execute different kernels between command queues on compute devices</td>
</tr>
<tr>
<td>• Consistent data size definitions across benchmarks</td>
<td></td>
<td>• Microbenchmarks for stress testing and comparing OpenCL and CUDA implementations</td>
<td>• Integrated with multiple profiling systems to study applications at different layers of abstraction</td>
</tr>
</tbody>
</table>

Table 4.1: Existing open source benchmarks commonly used in heterogeneous computing research and the goals of Valar. The table lists the common characteristics across multiple suites, followed by their differences.

### 4.2 Valar Benchmark Characteristics and Behavior Patterns

An application’s execution is a function of the algorithm and it’s mapping to a heterogeneous device. The importance of architecture-independent metrics to model applications has been discussed in prior work for single-threaded programs and multi-core devices [63]. For heterogeneous systems, where work is offloaded to a secondary device, we use application-level features such as AL1 features. These features serve the purpose of a simple architecture-neutral model to characterize host-device interaction that can occur on a heterogeneous system. The Valar benchmarks have been categorized based on:

- **Implementation**: the mapping of computation onto the OpenCL compute device or devices, and
- **Behavior**: the interaction of the compute devices with the host.

The following characterizes the implementation characteristics of OpenCL kernels.
CHAPTER 4. VALAR AND OTHER HETEROGENEOUS APPLICATIONS

Figure 4.1: The dynamic execution patterns of applications considered for Valar. (a) shows mapping of a heterogeneous application onto compute devices. (b) shows the possible host-device behavior with regards to data sharing.

Computation Pipeline Implementations: A computational pipeline denotes applications where the majority of the algorithm’s execution is carried out on a particular OpenCL device. Applications implemented for discrete GPU devices [12, 58, 90] follow such an implementation pattern to minimize the cost of data movement [45] over PCIe and efficiently utilize all the available compute units.

Multi-Device Decoupled Implementations: This category refers to applications which utilize multiple compute devices in the execution of their algorithm. In this context, decoupled refers to infrequent communication between the devices. Applications that can benefit from a multi-device, decoupled, implementation have independent units of work that can be dispatched to different devices and do not need to communicate with each other [103].

Multi-Device Coupled Implementations: This category also refers to benchmarks which utilize multiple compute devices in the execution of their algorithm. However, coupled implementations have a higher degree of communication between the compute devices and the host. Coupled implementations have recently become feasible with the introduction of APU devices (where the CPU and the GPU share a common system memory), architectural support for data movement (e.g., memory mapping over PCIe) and lower kernel launch overhead. Some recent coupled implementations possessing irregular parallelism include hashing [51], databases [13] and image feature extraction [73].

The three implementation patterns are illustrated in Figure 4.1. In addition to the implementation, the behavior of a heterogeneous application describes how the host and the OpenCL devices interact with respect to shared data.
Latency Sensitive Behavior: Latency sensitive applications commonly have real time constraints and tolerate only limited coalescing of work. Such applications may not fully utilize a compute device’s memory bandwidth or the PCIe bandwidth.

Streaming Behavior: Streaming behavior denotes applications which have a continuous input stream and/or output stream of data. Such applications commonly have large amounts of data and are bandwidth intensive.

Quality of Service or Anytime Behavior: Quality of Service (QOS) denotes an application where the output must match certain quality requirements (e.g., residue error in an algebra solver). Anytime Behavior refers to algorithms where there is flexibility in the quality of the resultant output. Web search is an example of an Anytime Behavior application \[38\] \[50\]. In Anytime applications, execution time is determined by external factors such as system load \[50\]. Implementing applications based on QOS is non-trivial for heterogeneous platforms due to the “offload to compute device” programming model, weak exception support and a non-negligible cost of checking data.

It should be noted that the Implementation characteristics discussed are not mutually exclusive within an application. For example, an application could be a compute pipeline with multiple kernels executing on a device and also be a decoupled implementation when it shares the pipeline’s results with other compute devices. Similarly the behavioral characteristics are not always the same for an application. For example, a benchmark could be latency sensitive in scenarios with low system load, while QOS sensitive during times of high system load. In this work, the behavior classification is restricted to interaction between a host and device. We leave the interaction within a compute unit on a OpenCL device such as load imbalance between compute units and communication using atomics for future work. Section 4.3 discusses our sample applications and the characteristics they exhibit.

4.3 The Valar Benchmark Suite

The Valar benchmarks generate the execution patterns shown in Figure 4.1. We have selected applications from a number of domains such as scientific computing, computer vision and data mining. The benchmarks selected exhibit variations in their OpenCL kernel execution and host device interaction, based on their real world input parameters.

The implementation and behavioral properties for Valar are presented in Table 4.2. Designing inputs for these benchmarks is a challenging task. As shown in Table 4.3, the input parameters stress different components of a heterogeneous system. The notion of an “input size”, as commonly discussed in SPEC benchmarks \[50\], cannot be easily applied here. Workload sizes and input arguments to the Valar benchmarks need be chosen based on behavioral requirements. The benchmarks are described in more detail in the following sections.
Table 4.2: The application domains covered by each benchmark. Implementation and behavior characteristics seen in each benchmark are also shown.

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Benchmark Domain</th>
<th>Implementation Characteristics</th>
<th>Behavior Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>SURF</td>
<td>Computer Vision</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Adaptive FIR</td>
<td>DSP</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Search</td>
<td>Data mining</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Traffic</td>
<td>Analytics</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Physics</td>
<td>Physics model</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Total # of Occurrences</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4.3: Input arguments for benchmarks and their influence on benchmark execution.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>IP Parameters</th>
<th>Effect on application behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>SURF</td>
<td>Image size and colors</td>
<td>Kernel execution and host device I/O. Data driven since color gradients affect processing</td>
</tr>
<tr>
<td></td>
<td>Threshold</td>
<td>No. of iterations in kernel and output data size</td>
</tr>
<tr>
<td>Adaptive FIR</td>
<td>Hessian stack Depth</td>
<td>No. of kernel calls / frame</td>
</tr>
<tr>
<td>filter</td>
<td>Total No. of Blocks</td>
<td>Upper bound on total amount of processing</td>
</tr>
<tr>
<td></td>
<td>Block Size</td>
<td>Minimum amount of work in each kernel call, minimum granularity of data movement</td>
</tr>
<tr>
<td></td>
<td>Tap value modulation frequency</td>
<td>Frequency of calling secondary OpenCL device (CPU device) to change filter weights</td>
</tr>
<tr>
<td></td>
<td>Dispatch Size</td>
<td>No. of blocks processed at a time, controls I/O and computation per kernel</td>
</tr>
<tr>
<td>Search</td>
<td>Input Size</td>
<td>Changes number of data chunks to be searched by the GPU</td>
</tr>
<tr>
<td></td>
<td>Update Frequency</td>
<td>Frequency of calling secondary OpenCL device (CPU) for tuning the search parameters</td>
</tr>
<tr>
<td>Traffic Simulation</td>
<td>Maximum Velocity</td>
<td>No. of kernel calls</td>
</tr>
<tr>
<td></td>
<td>No of vehicles</td>
<td>Amount of I/O and kernel execution performance</td>
</tr>
<tr>
<td></td>
<td>Distance simulated</td>
<td>Amount of I/O and kernel execution performance</td>
</tr>
<tr>
<td></td>
<td>Update frequency</td>
<td>Application I/O frequency</td>
</tr>
<tr>
<td>Physics</td>
<td>Distribution of particles</td>
<td>Workload balance between CPU and GPU execution</td>
</tr>
<tr>
<td></td>
<td>No of particles</td>
<td>The amount of work done by the CPU and GPU</td>
</tr>
</tbody>
</table>

4.3.1 Speeded Up Robust Features (SURF)

The SURF (Figure 4.2) algorithm generates features from images that are strongly invariant to changes in scale, orientation and intensity in the image. SURF represents a large class of algorithms where profiling and profile-guided optimization can play a role in tuning performance. We modified a publicly available, optimized OpenCL implementation of SURF [73].

SURF is commonly used as a component in computer vision applications, such as image search and video stabilization [14]. The SURF implementation serves as a target computational pipeline in future chapters of this thesis. SURF has been extended and specialized for different applications as shown in Figure 4.2.
CHAPTER 4. VALAR AND OTHER HETEROGENEOUS APPLICATIONS

4.3.2 FIR Filter Application

FIR filters are widely used in digital signal processing. The input signal to an FIR filter is split into chunks, called blocks, which are processed by a kernel. Each element in the block is multiplied by a number of coefficients, called taps, producing an output block (i.e., an output signal). The number of taps for the filter decides the sharpness and stop-band attenuation characteristics. The number of taps affects the memory usage and amount of computation per OpenCL workgroup. Adaptive filtering [110] extends the FIR filter by changing the weight of the taps for the filter on a separate command queue based on signal characteristics. Adaptive filters are used in audio filtering, speech recognition, and pulse detection applications.

Figure 4.3: Adaptive FIR Filter application with Tap weight Change.
4.3.3 Mixed Particle Simulation

A mixed particle simulation refers to a physics simulation where particles are of different sizes \cite{11}. While particle simulations are easily parallelized, mixed particle simulations are irregular in nature and inefficient if implemented naively. The inefficiency results from the non-uniform granularity of the computation, especially for the collision detection, which is the most expensive part in a particle simulation. If there is a single large particle and many small particles, the number of collisions detected on the large particle can be significantly more than the number detected on the small particles \cite{41}. The particle simulation has been implemented as a pipeline shown in Figure 4.4. Figure 4.4 shows that large-small particle collisions and the large-large particle collisions are calculated on the CPU. The small-small collisions which are regular in nature are calculated on the GPU.

![Figure 4.4: Collision Detection Application.](image)

4.3.4 Traffic Simulation

Traffic simulation is an example of an agent-based modeling \cite{106} application. The traffic is modeled as a simple cellular automaton model for flow. The model can reproduce traffic jams, (i.e., the simulator can capture the deceleration of a car when the road is crowded with a high density of cars). The model is based on randomization since one car braking due to a random cause can slow down the cars behind. We base our implementation on previous published work on traffic simulators \cite{78,106,116}.

![Figure 4.5: Cellular automaton of simulated traffic for a two lane road. A car’s (highlighted cells) speed and position is expressed in terms of cells.](image)
Search algorithms have multiple strategies for refining their results. In search algorithms, system load and cutoff latency determine room for refinement. Our search application is an online system capable of dynamic updates, where the CPU and GPU cooperate to build the final result. The GPU executes the search for elements in different ranges of values for the input data. The CPU reduces the data and generates a statistical mean of the output ranges.

The search application adapts according to the intermediate result. The adaptation involves changes in search size, which change the load on the GPU dynamically. Dynamic search applications are used in industrial engineering for process capability tuning.

Figure 4.6: Search benchmark. The GPU executes the search on the input data for elements in different target ranges. The CPU reduces the data set.

Concurrent Command Queue Applications

The Valar benchmarks provide examples of applications with host-device interaction. Additionally one of our goals while designing benchmarks was to evaluate novel applications leveraging the OCU. The Storm processing framework with its abstractions of Spouts, Tuples and streams would be ideal to evaluate scheduling and performance monitoring using the OCU. However, Storm has been built using multiple open source components implemented in different programming languages. For example, Storm is built using Nimbus and ZooKeeper and other Java Virtual Machine (JVM) based frameworks.

To avoid the complications associated with integrating JVM frameworks into our simulation infrastructure, we have built another set of OpenCL-based compute benchmarks which build computational topologies, within OpenCL. The role of our concurrent command queue benchmarks is to provide test applications to allow us to study mapping of
CHAPTER 4. VALAR AND OTHER HETEROGENEOUS APPLICATIONS

computation onto heterogeneous devices.

We use two sets of benchmarks for evaluating the architectural enhancements proposed in Chapter 7. We use Set 1 for evaluating multiple command queue mapping and Set 2 for evaluating pipe-based communication. Overall, six applications from different computing domains are used for evaluation. Existing suites such as Rodinia and Parboil are not appropriate for our evaluation as they only utilize single command queue mapping. Those would require substantial modifications to their implementation to support multiple command queue mapping.

Set 1: Multiple command queue mapping

1. Matrix Equation Solver: Application is a Linear Matrix based solver implemented on GPUs [101]. The benchmark evaluates the equation \( C = (\alpha(AB^{-1}) + \beta((A+B)*B) \), where A, B and C are square matrices. The computation is done in three parts:
   (a) \( C_0 = \alpha(AB^{-1}) \),
   (b) \( C_1 = \beta((A+B) \times B) \) and
   (c) \( C = C_0 \times C_1 \).

Parts (a) and (b) execute in parallel using different subdevices. Part (c) is computed using the entire GPU device.

2. Communication Channel Analyzer: 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: 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.

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

3. Clustering Algorithm: An EDA application that forms clusters of different cells from a netlist based on a clustering score for efficient placement design [86]. The benchmark uses two kernels to perform clustering on more than 100,000 cells. Kernels are mapped to different compute units on separate command queues.

4. Search Application: Search application benchmark from the Valar benchmark suite [75]. Application performs a distributed search using two kernels. The kernels are mapped to different command queues for parallel execution.
Set 2: Pipe-based Communication

1. **Audio Signal Processing:** A two-channel audio signal processing application with three stages of compute is used as a benchmark. The three stages are 1) FFT (Fast Fourier Transform) 2) FIR (Finite Impulse Response) filter and 3) Gaussian Noise Elimination [76]. Each stage provides data to next stage for processing. Each stage of compute is a separate command queue mapped to a different compute unit. The data communication between stages is achieved using pipe objects.

2. **Search-Bin Application:** The search benchmark, described in Set 1, is enhanced to perform bin allocation with the search [75]. The application uses two kernels to search through an input data set. The third kernel assigns the searched data to different bins. The search kernels provide data to the bin kernel using a pipe object. All of the kernels are mapped to different command queues for simultaneous execution.

### 4.5 Concurrent Command Queue Microbenchmarks

Microbenchmarks were also developed to observe the effects of non-homogeneous distribution of work to different compute unit of an accelerator device. Our initial concurrent command queue exploration was carried out with these microbenchmarks.

**Unbalanced Vector Operation:** Unbalanced Vector Operation shown in Figure 4.7 is a simple microbenchmark that can generate kernels where different workgroups conduct a varying amount of computation and memory accesses. The work assigned to each workgroup is defined in a global memory buffer accessed by its workgroup id. The number of memory accesses and arithmetic operations carried out by each workgroup can be defined by changing a size array. By controlling the distribution of work and the scheduling policy of workgroups to compute units, we can generate a variety of distributions of work assigned to each compute unit.

**Concurrent Command Queue Vector Operations:** The Unbalanced Vector Operation benchmark was extended to support multiple command queues mapping to a single OpenCL device (Figure 4.7). Multiple unbalanced kernels are enqueued on each command queue. These command queues can map to all compute units in the device or a subset of the device.

### 4.6 Future Targets: Storm and Computational Topologies

Storm is an open source framework introduced by Twitter to implement a processing model similar to MapReduce [68, 69]. Storm is a distributed realtime computation system and exposes a set of primitives to simplify implementing realtime applications. The primitives simplify development of parallel real-time computation similar to how MapReduce simplifies
CHAPTER 4. VALAR AND OTHER HETEROGENEOUS APPLICATIONS

a) Imbalanced Vector addition benchmark where different workgroups execute an unequal amount of work.

b) Test benchmark for concurrent command queue experiments.

Figure 4.7: Imbalanced vector addition microbenchmark. The workgroup’s unique group id identifier used by each OpenCL workgroup to find its work.

development of parallel batch processing. Storm has been applied to processing messages, updating databases (stream processing), continuous queries on data streams and streaming results to clients (continuous computation).

Storm’s computation is expressed using a set of primitives combined into a topology. Topologies consist of tuples, streams and bolts. A popular implementation of the Storm primitives leverage open source technologies such as Apache ZooKeeper, Hadoop and other libraries. This publicly available implementation of Storm has been widely used in big data processing for social networking and analytics. Storm topologies are executed on clusters built using commodity multicore systems.

The relevant terminology for a Storm’s topology is described in more detail below.

• **Tuple:** A data point consisting of multiple fields.

• **Stream:** An unbounded set of tuples.

• **Spout:** A source of streams in a topology. Spouts read tuples from an external source and insert them into the topology (e.g., user data from the Twitter API).

• **Bolt:** Unit of computation that accepts input streams and produces output streams. Bolts can do filtering, functions, aggregations, joins, talking to databases. Complex
stream transformations often require multiple steps across multiple bolts.

A sample Storm topology is shown in Figure 4.8. We see in Figure 4.8 that Storm provides a similar model of computation to the persistent kernel programs discussed in Chapter 1. Storm topologies are similar to persistent kernels, since in either case, data is continuously operated on by a set of functions.

**Implementing Storm Topologies:** A Storm cluster is superficially similar to a Hadoop cluster. If you run on Hadoop you run “MapReduce jobs”, on Storm you run “topologies”. “Jobs” and “topologies” themselves are very different – one key difference is that a MapReduce job eventually finishes, whereas a topology processes messages forever (or until the user kills it).

Implementation of a Storm topology using presently available open source technologies is shown in Figure 4.8. There are two kinds of nodes on a Storm cluster: the master node and the worker nodes. The master node runs a daemon called ”Nimbus” similar to Hadoop’s ”JobTracker”. Nimbus is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures.

**Supervisor:** A ”Supervisor” is a daemon run on each node. The supervisor listens for work assigned to its machine and starts and stops worker processes as necessary based on what Nimbus has assigned to it. Each worker process executes a subset of a topology; a running topology consists of many worker processes spread across many machines.

**Zookeeper:** Communication between the Supervisor and the master is implemented using a ZooKeeper cluster. Apache ZooKeeper is open-source server software which enables highly reliable distributed coordination. ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. These services are used in some form or another by distributed applications.

### 4.7 Future Targets: Applications

For future work, additional benchmarks can be evaluated and possible candidates have been summarized below. The lack of publicly available OpenCL applications is one of the main constraints preventing a larger range of benchmarks from being included in Valar. The interaction between the different components within these workloads make them prime candidates for inclusion in Valar.

1. **Multiresolution Adaptive Numerical Environment for Scientific Simulation (MADNESS):** MADNESS provides a high-level environment for the solution of integral and differential equations in many dimensions using adaptive, fast methods with guaranteed precision. MADNESS enqueues a number of small matrix multiplication kernels with varying dimensions. These kernels cannot individually fully
utilize a heterogeneous device. Concurrent execution of kernels on a discrete GPU has demonstrated performance improvements.

2. **Multiphysics Examples**: More complex physics applications can be developed to study the interaction between two linear algebra modules and their mapping to heterogeneous devices.

3. **Genetic Algorithms**: Genetic algorithms can utilize profiling capabilities to report the number of individuals or the fitness of these individuals.

4. **Mummer GPU**: Mummer is a commonly-used string matching engine used in DNA research. Tracking and profiling the number of matches found in a dataset could be implemented using the *Haptic* framework discussed in Chapter 5.

5. **Compressed Sensing**: Compressed sensing is an upcoming field within image processing. In compressed sensing, the final image is reconstructed using a variety of algorithms in a data-driven fashion.

6. **Barnes Hutt Algorithm**: Barnes Hutt is commonly used to study the scalability of data parallel architectures when tasked with irregular algorithms. The Barnes Hutt algorithm is based on partitioning resources and communication between cells. The algorithm carries out spatial partitioning of the volume occupied by the bodies. The bodies are placed in bins (cubes, for instance). In the bin, precise brute-force collision detection is carried out for all bodies. The Barnes Hutt algorithm can also have feedback-based usage, where the size of the precise all-pairs kernel can be changed based on the problem size.
7. **Track Learn and Detect (TLD) Algorithm:** The TLD algorithm \[60\] is a modern object tracking algorithm that relies on the interaction of a feature detection and a machine learning system for tracking moving objects.

### 4.8 Summary of Heterogeneous Benchmarks

The evaluation of heterogeneous applications and the introduction of the Valar benchmarks are important contributions of this thesis. We have provided a detailed discussion of these applications in this chapter. We have contributed a taxonomy to properly characterize heterogeneous applications, both from an *implementation* standpoint and a *behavioral* perspective. The applications discussed here have served as the computational pipelines used to evaluate our proposed *Haptic* profiling framework.

In addition to contributing to workload characterization of heterogeneous systems, additional architectural enhancements are proposed in further chapters of this thesis. To evaluate the applications enabled by our architectural enhancements, modern processing frameworks such as Storm were evaluated. This evaluation of modern processing frameworks has motivated the development of the additional concurrent command queue benchmarks.
Chapter 5

**Haptic Profiling Architecture and Analysis Devices**

5.1 Introduction

To simplify programming heterogeneous devices, the OpenCL kernel and the compute device are being abstracted away using domain specific languages (DSL) [23] and libraries [15]. This abstracts away the complications of parallel programming such as synchronization and resource management. Abstraction using domain specific libraries simplifies application development for developers who are not parallel programming experts. Developers do not have to implement data parallel code and can simply invoke the underlying parallelism using familiar interfaces. However, modularization makes tuning applications harder [62], reducing performance. This issue is especially relevant for cross-platform specifications such as OpenCL, which target different classes of devices using the same program. Also, due to device specific optimizations, specialization usually requires deep understanding of hardware and source code access [15].

The importance of profiling architectures and performance monitoring frameworks has been discussed in Chapter 1. In this chapter we present one of the key contributions of this thesis towards profiling heterogeneous applications. We present the architecture of the Haptic framework (Heterogeneous Application Profiling Tools in OpenCL). The Haptic framework is based on OpenCL which allows programmers to apply it on all OpenCL compliant devices. The Haptic framework can provide performance monitoring capabilities and manage the interaction between multiple OpenCL devices. However, since Haptic is based on OpenCL, its performance awareness is restricted to OpenCL events, which are only available at a command granularity. Our contributions in later chapters address these limitations using additional hardware and software support.

We present the implementation of Haptic in Section 5.4. In Section 5.5 we discuss how Haptic has been applied for profiling and specialization of heterogeneous applications. The
performance results relevant to *Haptic* have been covered in Chapter 8.

### 5.2 OpenCL Events

The OpenCL standard includes a profiling function `clGetEventProfilingInfo`, which returns timing statistics regarding OpenCL commands such as kernel launch and device IO. Common usage scenarios of OpenCL events, include managing asynchronous IO and generating timing information. Providing a profiling framework as part of an open standard can greatly facilitate performance analysis, since any compliant performance analysis tool can be leveraged. A profiling framework can also enable more complicated and dynamic scheduling policies across devices for data-driven programs. We aim to be able to extract profiling data from an application in a cross-platform manner similar to the Tau [66] toolchain.

<table>
<thead>
<tr>
<th>Stamp Name</th>
<th>Command Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_QUEUED</td>
<td>Enqueued, waiting to submitted</td>
</tr>
<tr>
<td>CL_SUBMITTED</td>
<td>Submitted to device</td>
</tr>
<tr>
<td>CL_START</td>
<td>Started executing on device</td>
</tr>
<tr>
<td>CL_END</td>
<td>Finished execution</td>
</tr>
</tbody>
</table>

Table 5.1: OpenCL event states provided by compliant implementations. These states denote the status of the corresponding OpenCL command.

### 5.3 OpenCL Event Profiling

To provide for coarse-grained synchronization, OpenCL provides barrier calls that include `clFlush()` and `clFinish()`. To enforce finer-grained synchronization, the OpenCL events (`cl_event`) can be used to determine the status of a command. Event objects identify unique commands in a queue and thus provide command-level control.

Wait lists are arrays of `cl_event` objects passed to OpenCL commands. Wait lists can be used to block on more than one command. Events and wait lists can be used to enforce synchronization (by checking command status) in a multi-device usage scenarios or with out-of-order command queues. Once all the commands tied to the event objects in the wait list complete, the waiting command queue can be started. The `clGetEventProfilingInfo()` function is utilized heavily in *Haptic*. 
5.4 OpenCL Analysis Devices

5.4.1 Architecture and Implementation

A library writer implementing a computational pipeline targeting heterogeneous devices has to write generalized code, leading to non-optimal performance. Application developers using such libraries could improve performance using application-specific knowledge (e.g., preprocessing, data or program transformations). Using this class of specialization can improve a computational pipeline’s performance. But specialization is challenging to implement since it may require additional platform-specific data management and synchronization (e.g., optimal data placement may change in an APU versus a discrete GPU) [9, 41].

To address this problem, Haptic’s software defines an architectural resource known as an analysis device which interacts with a computation pipeline. For example, an analysis device could be a device that preprocesses data before a stage in the computational pipeline. The potential benefit of such specializations cannot be easily predicted during library development and thus such specializations cannot be hardcoded into a pipeline. However, application developer can specialize pipelines by leveraging Haptic’s performance monitoring and choose an analysis devices to launch preprocessing functions, change data layout

Figure 5.1: Architecture of Haptic’s software stack based on OpenCL. The compute pipeline is called by an High Level Application. The application can provide domain specific information to the pipeline which will be used by the analysis device interface. OpenCL kernel execution on analysis devices is managed by Topology and Profiler modules. Table 5.2 discusses the role of the modules.
and track values.

*Haptic* implements an analysis device framework using OpenCL, as shown in Figure 5.1. Two modules, the profiler module and the topology module, maintain the state necessary to introduce specialization into a pipeline. The profiler module provides performance monitoring and maintains a platform-independent performance picture of the OpenCL computation pipeline. Within the profiler module, the event profiler keeps track of executed OpenCL commands and the value tracker is used to monitor data that may provide optimization information. The topology module manages the OpenCL devices available on the platform. The topology module also enqueues analysis kernels for execution on the analysis device and ensures proper coordination with compute devices. Because the topology module exposes the underlying OpenCL command queue (`cl_command_queue`) [41] as part of its interface, existing OpenCL applications can leverage *Haptic* with minimal modification. *Haptic*’s modules are described in more detail in Table 5.2.

The functionality of the modules in Table 5.2 is exposed to library developers via the `analysis_device` interface. The API (discussed in Section 5.4.2) allows developers to specify: value checking and performance monitoring frequency, analysis kernels to launch, and topology of compute and analysis devices which are mapped to the available resources (e.g., CPU and GPU).

<table>
<thead>
<tr>
<th>Profiler Module</th>
<th>Role in HAPTIC</th>
<th>Topology Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Profiler</td>
<td>Maintain a vector of cl_event objects. Records</td>
<td>Topology</td>
</tr>
<tr>
<td></td>
<td>command execution timestamps and status</td>
<td>Manage &quot;cl_command_queue&quot; and context</td>
</tr>
<tr>
<td></td>
<td>provided by OpenCL runtime. Minimize overhead</td>
<td>interfaces to OpenCL applications. Enables</td>
</tr>
<tr>
<td></td>
<td>by query events off the critical path</td>
<td>existing OpenCL compute pipelines to</td>
</tr>
<tr>
<td>Value Profiler</td>
<td>A value profiler instance records data in OpenCL</td>
<td>Analysis - Topology</td>
</tr>
<tr>
<td></td>
<td>buffers specified by a user. Implemented using calls</td>
<td>manage command queue mapping to analysis</td>
</tr>
<tr>
<td></td>
<td>to OpenCL API functions e.g. clEnqueueMapBuffer</td>
<td>device. Launches analysis kernels based on</td>
</tr>
<tr>
<td>Sampling Information</td>
<td>Sampling granularity information is required when</td>
<td>Sampling and synchronization with Topology</td>
</tr>
<tr>
<td></td>
<td>the analysis device and the value profiler is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>initialized. Tradeoff between sampling interval</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(fidelity) and performance overhead</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Table 5.2: The main components of Haptic and their role in heterogeneous applications.](image)

a) Main components of profile module  

<table>
<thead>
<tr>
<th>Component</th>
<th>Role in the HAPTIC system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Kernel</td>
<td>Configure kernel executed on analysis device</td>
</tr>
<tr>
<td>Inject-Analyis</td>
<td>Function that launches analysis kernels based on &quot;Analysis-Toplogy&quot; defined and the profiling granularity chosen</td>
</tr>
<tr>
<td>Create Buffers</td>
<td>OpenCL cl_mem buffers which serve as input to the analysis kernels</td>
</tr>
</tbody>
</table>

c) Interface exposed to library developers to build analysis devices that can be integrated into their pipelines and leverage the profiling and topology capabilities provided by HAPTIC.
The low-level nature of OpenCL does not provide performance portability, due to differences across device architectures. OpenCL runtime calls have varying performance on different devices. We leverage the capabilities of architectures such as AMD Fusion and a discrete GPU by studying vendor SDK microbenchmarks for different data locations and using the right runtime calls for data management (e.g., clEnqueueMapBuffer versus clEnqueueReadBuffer). Interaction between analysis devices and computation pipelines could be improved through driver support [26], but our focus is on presenting an architecture-neutral interface using OpenCL.

5.4.2 Use of Analysis Devices

Library developers can use Haptic’s API within a compute pipeline to enable possible specializations at runtime. A usage example of Haptic’s API is shown in Figure 5.2. The example shows the source modifications required to add a data-preprocessing step before a stage in a compute pipeline.

The analysis_device base class in Figure 5.2 provides an interface to a user to build analysis devices. The base class interface maintains the necessary OpenCL state within the respective modules (Figure 5.2). To define a new device as shown in Figure 5.2 (p_device), the library developer creates a new class by inheriting from the analysis_device base class. The declaration of p_device contains the relevant OpenCL buffers and invocations to Haptic’s APIs. The Haptic-provided function inject_analysis() (Figure 5.2) synchronizes the compute and analysis command queues and launches the analysis kernel. The function init_device() is necessary to configure the topology used by p_device and the profiler’s sampling intervals. Simpler APIs would result in fixed sets of buffers or fixed OpenCL topologies. This would reduce the applicability of Haptic. The API we have developed in this thesis does not require detailed OpenCL knowledge or the implementation of threads or multi-device synchronization.

The programming paradigm followed by Haptic has gained acceptance in well-known open-source projects such as Ceres (a non-linear optimization toolchain) [5], where users implement their non-linear optimization problems by reusing and extending base-class interfaces. This methodology allows usage of architecture-agnostic performance information such as required Quality Of Service (QOS) and acceptable error rate to optimize computation.

To summarize, the effort of specialization is divided between the library developer and the application developer. The library developer can build and expose multiple analysis devices (using Haptic’s interfaces), while application developers can choose to use these devices based on domain experience. Due to the object-oriented (C++) nature of Haptic, library developers can also implement custom synchronization schemes (e.g., waiting on non-OpenCL applications or subsets of command queues) by over-riding the base class. Usage scenarios of analysis devices in Sections 5.5.
Figure 5.2: Preprocessing device implemented by inheriting from the `analysis_device` class. The “Preprocessing” kernel and topology details are provided to Haptic by the library developer. HAPTIC shows API utilized by developer. The application developer can choose preprocessing.

### 5.5 Haptic evaluation using Valar Benchmarks

The Valar benchmarks SURF and FIR are common OpenCL pipelines extended by application developers. We have evaluated Haptic by modifying these applications to utilize analysis devices to perform additional functionality not present in the original applications.
5.5.1 FIR Applications

The FIR application discussed in Section 4.3 has been extended by adding *Adaptive filtering*. In this context, adaptive filtering refers to a filtering technique implemented by changing the weight of the taps for the FIR filter based on signal characteristics. Adaptive filters are used in audio filtering, speech recognition, and pulse detection applications.

We define an analysis device using *Haptic* that interacts with the FIR filter and updates the tap weights to perform adaptive filtering. The *Haptic* implementation of the FIR benchmark is shown in Figure 4.3. The performance improvements obtained when using analysis devices for adaptive FIR filtering is discussed in Section 8.2.

![Figure 5.3: The FIR kernel is executed for each block of the input. FIR pipeline is enhanced with an analysis device that enables adaptive filtering by modifying filter tap weights.](image)

5.5.2 SURF Applications

We modified the Valar implementation of SURF discussed in Section 4.3 to serve as the computational pipeline used by *Haptic* [73]. SURF is commonly used as a component in computer vision applications, such as image search and video stabilization [14, 73, 75].

Applications that use SURF have specific needs, and the algorithm may perform sub-optimally if the requirements of the application are not taken into account. For example, using application-specific knowledge, data transformations can be used to improve spatial locality. Also, in some cases, portions of the computation pipeline can be disabled entirely. To demonstrate *Haptic* and improve the performance of SURF, we have implemented analysis devices that take advantage of application-specific information to perform useful optimizations. We describe below, the implementation of analysis devices. The performance impact on applications is discussed in Section 8.2.

**Disabling Unnecessary Kernels:** Image processing pipelines commonly contain preprocessing stages to eliminate redundant data. Similarly, for a video consisting of static image scenes across frames, the features produced by SURF will not change. To take advantage of this fact, we implement an analysis device that performs a preprocessing step of comparison between frames, and disables the entire feature generation pipeline if frames...
Chapter 5. Haptic Architecture

Figure 5.4: SURF Algorithm and its usage as a computation pipeline in three applications.

that do not change significantly (Figure 5.4a). The benefits of throttling execution of the SURF pipeline include power savings and increased frame-rate.

Disabling Orientation Kernels: For a video feed where the camera’s focus and view does not change, the orientation of features does not change. This allows us to implement an analysis device that monitors the change in orientation across features (Figure 5.4b). If the analysis device does not see a variation in orientation, corresponding compute stages can be disabled.

Data Monitoring and Transformations: By counting the number of features generated by SURF across a window of frames, we can detect scene changes. We evaluate the performance of an analysis device that uses the value tracker module for OpenCL buffers. Using the value profiler module, we monitor the buffer that stores the feature count (Figure 5.4b). This device has also been used to modify the threshold input parameter if a consistent number of features are required across frames. The number of features detected affects execution time, this technique can be used to provide a consistent frame-rate regardless of data.

5.6 Summary

This chapter discusses the benefits of analysis devices provided by Haptic and how they have been applied to the Valar benchmarks. The Haptic OpenCL extensions allows computational pipelines to be much more extensible when incorporated into heterogeneous applications. Chapter 6 discusses architectural enhancements for other closely-coupled workloads. Chapter 8 discusses the performance results of Haptic discussed in this Chapter.
Chapter 6

Architectural Support for Profiling

6.1 Introduction

Previous chapters of this thesis have described heterogeneous applications and methods to classify their behavior (Chapters 3–4). To study the heterogeneous applications discussed in Chapter 4, we have developed the Haptic framework (Chapter 5). Haptic has been used to better understand the interaction between compute devices while running applications developed in OpenCL. The high-level application optimizations enabled using Haptic can be implemented using performance information obtained using OpenCL events. Due to the low-level nature of OpenCL, performance optimization of the underlying compute kernels is still key to achieving good performance. Performance optimization requires representative information from the underlying hardware. However, additional high quality performance information can also enable new uses of performance monitoring hardware for non-traditional uses. For example, recent research has demonstrated unique applications of online performance information towards anomaly detection and debugging support [85, 98].

Hardware support for performance analysis in the form of performance counters has a rich history [34, 91, 117]. Performance profiling of heterogeneous applications requires low overhead sampling and minimal perturbation to system state, just like any other application family. An alternative to hardware support for profiling is software instrumentation [64]. Accurately profiling multi-threaded architectures (such as a GPU) using instrumentation is a challenge since adding code can easily affect the execution of the program in unexpected ways. For example, inserting instrumentation into a kernel that reads a time-stamp for each work item can change register usage [11, 31]. A change in register usage could subsequently affect the number of threads scheduled on a compute unit (occupancy) and affect performance [11, 31]. To minimize perturbation due to profiling, vendor provided profilers usually gather statistics on a subset of compute units [2, 54] or over multiple program runs. To instrument kernel code with profiling routines, insertion of instrumentation code at carefully selected sampling points can build a set of performance snapshot using sparse
This chapter describes a new class of architectural support for performance monitoring. The architectural support for profiling in this thesis consists of a definition of performance counters at different stages in the compute unit that can be used to measure hardware utilization levels. The utilization levels of compute units on a heterogeneous platform can be subsequently used by the OCU workgroup scheduler. The architectural enhancements discussed below have also been used to build performance models for heterogeneous applications. The models have been used to study the time varying behavior of an application and also to compare different compute kernels executing on an heterogeneous system.

This chapter is organized as follows. Section 6.2 discusses how to select an appropriate profiling granularity. Section 6.3 covers the performance counters that have been defined for the AMD Evergreen and Southern Island architectures. We have explained our evaluation methodology in Section 6.5. We have show how throughput counter information can be used to study time varying behavior of a compute kernel in Section 6.6.

### 6.2 Choosing a Profiling Granularity

*Profiling granularity* denotes the interval at which performance measurements are recorded. The profiling granularity on a pipelined processor can be measured in terms of executed instructions or cycles. Choosing an interval measurement unit and the interval sampling rate are the key challenges in building any profiling architecture. A coarse-grained profile (long sampling intervals) can obscure behavior within the interval, while a finer granularity (shorter intervals) may overwhelm our capture capabilities, and may limit the number of samples we can capture. We identify the appropriate unit for defining profiling granularities below.

#### 6.2.1 Basic-Block Granularity

Profiling of single threaded and out-of-order platforms was commonly done on a basic block granularity. In a basic block (a contiguous set of instruction without control breaks), all of the instructions between the first and last instruction in the block are executed, in order. This unique property of basic block boundaries\(^1\) allows insertion of instrumentation and statistics recording at only the boundaries, reducing overhead and interference.\(^64\) Instrumentation code for multicore platforms can also be inserted at the basic block boundary\(^113\) since each core would have a single thread of execution, and thus clear boundaries.

Basic block based profiling methods do not logically map to throughput oriented devices such as GPUs, where the cumulative behavior of multiple threads dictates performance and hardware supports fast context switching with negligible cost.

---

\(^1\)One entrance and one exit of code
Additionally, basic block profiles are highly application specific (e.g., they will vary based on the basic block length). This does not allow us to use basic block characteristics across applications in order to predict performance of different applications \[\text{[10, 11, 72]}\]. To enable utilization of profiling information across applications, profiling information needs to be gathered at fixed intervals. These intervals can be cycle-based or instruction-based fixed intervals.

### 6.2.2 Instruction Intervals: Shards and Idempotent Regions

It is known that profiling monolithic application behavior, such as average instruction mix, obscures inter-application diversity \[\text{[72]}\]. Inter-application behavior can be obscured, since measurement units such as basic blocks are highly application specific.\[3\] For example, the basic block size can vary substantially between two applications. One of the goals of the application modeling discussed in Chapter 3 is to compare behavior of different applications.

Menon et al. have proposed a novel profiling granularity based on a dynamic instruction count \[\text{[72]}\]. A dynamic instruction stream is referred to as a shard. Shard-level profiles eliminate application boundaries and enable sharing across applications. The granularity of recording statistics in this work has been examined by changing the recording length \[\text{[72]}\]. An application can be divided into shards, each with the same number of dynamic instructions. Shards exploit variability of intra-application behavior and similarity of inter-application behavior. We observe links between shard characteristics and performance.

The amount of active state present in a program at any point in its execution can also be used to define a profiling interval. The active state of a thread refers to the live register state of the execution thread. A program execution point with no live register state can be used as a recovery point for speculation, checkpointing or exception handling. A program can be split into such regions where there is no active state present in the register file at the region boundary. Such regions are commonly known as Idempotent regions \[\text{[72]}\]. Idempotent regions are regions of source code with a small amount of register state present at the boundary. GPU programs commonly exhibit this pattern and profiling algorithms have been developed that track the amount of valid register state in a program \[\text{[72]}\]. The resultant idempotent regions can be marked to provide recovery points to support speculative execution.

### 6.2.3 Cycle Intervals

Instruction interval measurements such as shards provide a useful methodology to compare intra-application properties. While each instruction interval contains the same number of dynamic instructions, the time duration for each instruction interval can vary substantially.

An alternative to the instruction interval profiling granularity is cycle interval profiling. Cycle intervals accumulate performance information over a fixed number of cycles.
Cycle intervals can be used to compare the time varying behavior of a resource on heterogeneous architectures. While cycle intervals allow the study of inter-application diversity, the primary challenge of cycle intervals is selecting the interval size.

6.3 Throughput Counters for AMD Evergreen and SI GPUs

Throughput counters introduced in this thesis are *Cycle Interval* performance metrics. The goal of throughput counters is to build a utilization picture of the compute unit of an accelerator device over a cycle interval. Throughput counters can be defined as *AL4* level metrics, since they provide performance information within a compute unit.

Capturing utilization information for superscalar architectures is a nontrivial challenge due to the large amount of active state live at any time and the varying nature of this state, even in a single thread [36, 104, 114]. Generating utilization optimization for out-of-order and superscalar processors has concentrated on quantifying the loss in performance due to the finite capacity of architectural resources [36, 114]. The utilization information is then generated by complicated post-processing algorithms and can only be applied offline to guide performance optimizations [36].

Gathering utilization information on GPU compute units is an easier challenge due to the in-order nature of GPU compute units. We present throughput counters as a means to gather utilization information on GPU compute units. We have introduced throughput counters in this thesis due to the absence of an interface or metrics that can be used to study the time varying behavior of a GPU workload.

This thesis contributes throughput counters targeting AMD Evergreen and Southern Island GPUs. The compute units of the Evergreen GPU consists of decoupled engines as shown in Figure 6.1. ISA instructions are scheduled to run on the respective engines. Each engine has a different role, such as handling global memory accesses, performing floating point computation etc.. The AMD Southern Island compute unit is a more generic architecture consisting of multiple SIMD units, where each SIMD can execute any of the instructions. Throughput counters have been defined for both types of compute units by modifying our simulation framework [112]. The decoupled nature of these engines allow us to measure the utilization of each engine of a compute unit independently. The counters defined in this thesis measure the utilization of these engines and are listed below.

**No. of ALU Instructions Issued:** This metric is indicative of resource utilization of the ALU in the GPU. The metric reports the number of instructions fetched by ALU in the compute unit in the cycle interval.

**No. of Local Memory Instructions:** This metric is indicative of resource utilization of the LDS in the GPU. The metric reports the number of instructions that access the LDS in the compute unit within each interval.

**Memory Accesses in Flight:** Multicore platforms commonly implement a “witness“
counter to keep track of the number of memory accesses in flight. We track the witness counter within the compute unit. The number of accesses in flight denote the utilization of the memory subsystem as seen from the compute unit. The counter would keep track of the number of memory accesses in flight by increasing its value when accesses are dispatched from the compute unit and decrementing its value when accesses complete. The counter value is copied after every cycle interval.

**No. of Mapped Workgroups:** This metric counts the number of mapped workgroups in the interval. The metric can be used to evaluate how many workgroups are dispatched to a compute unit in a cycle interval. Based on the number of workgroups mapped on an empty compute unit, a scheduler can develop an upper limit on the resource requirements of a workgroup.

**No. of Unmapped Workgroups:** Counts the number of workgroups unmapped from a compute unit in each interval. The number of workgroups retired by a compute unit in each interval can be used to judge the dynamic compute intensity of the executing workgroups. The rate at which workgroups complete provides information on the load imbalance between different compute units.

A simple example is provided to show the benefits of having throughput counters, as presented in Figure 6.2. The benchmark shown in Figure 6.2 is an unbalanced vector addition microbenchmark (Section 4.5), where the work done by each workgroup is unbalanced. We see that the average number of memory inflight accesses in each compute unit is a
Figure 6.2: Benchmark of imbalanced vector addition on an Evergreen GPU. The number of memory accesses in flight in a compute unit are related to the workload imbalance.

function of the number of workgroups issued to that compute unit.

6.4 Implementing Throughput Counters on Compute Units

The goal of using throughput counters is not to guide offline application optimization, but is instead to provide online information to a workgroup scheduler or develop application signatures. Throughput counters can be implemented as a distributed profiling system, as shown in Figure 6.3. A detailed exploration into the implementation of performance counters for GPU compute units is a topic for future work beyond this thesis.

Our implementation methodology is similar performance monitoring designs found in current multicore processors. In multicore processors, uncore performance metrics refer to quantities measured in the vicinity of the processor core. In Intel platforms, the uncore performance monitoring facilities are organized into per-component performance monitoring (or PMON) units [30, 54]. A PMON unit within an uncore component may contain one of more sets of counter registers. With the exception of the system config controller, each
Figure 6.3: High level model of distributed profiling modules for compute devices. Utilization information is gathered per compute unit and aggregated to examine utilization of the complete device.

PMON unit provides a unit-level control register to synchronize actions across the counters within the unit (e.g., to start/stop counting).

One of the primary future challenges of this work is an examination of the latency that can be tolerated within the system. For a discrete GPU which typically contains at least 20 compute units, the distance between the throughput counters and workgroup scheduler could be substantial. Hardware profiling modules have been proposed for multithreaded CPU platforms [91, 96, 117]. Performance monitoring architectures have not been studied for the large core counts present in today heterogeneous systems and so have not been available to guide scheduling decisions and online feedback-directed optimizations.

6.5 Evaluation of Throughput Counters

In the heterogeneous computing ecosystem, the goal of vendor-provided profilers and hardware performance counters is to guide application developers when considering various performance optimizations to apply to their applications. Vendor-provided profilers, such as Nvidia Insight or AMD Code XL, usually gather statistics on a subset of compute units [2, 84]. Other methods of gathering performance information include sparse instrumentation, which can gather snapshots of profiling data [11] in a sparse manner. The sparsely sampled information is used to reconstruct a performance model of an application.

Throughput counters on current heterogeneous systems are not especially useful since performance information of individual compute units is not provided on modern GPUs. To study throughput counters, we present a simulation driven study using Multi2Sim [112]. The simulation methodology is shown in Figure 6.4. Multi2Sim has been enhanced with a
spatial profiling report to record statistics which can then be visualized.

Figure 6.4: Methodology to study throughput counters. The counters are updated as the simulation progresses. The output trace can be visualized or further analyzed.

Figure 6.5: Throughput counter data gathered for two workloads. Periodic behavior can be seen for each workload.

We show examples of performance information provided by throughput counters in Figure 6.5. The throughput counters capture the periodic behavior of a heterogeneous application where workgroups are mapped and unmapped. We see that the Histogram
CHAPTER 6. ARCHITECTURAL SUPPORT FOR PROFILING

65

trace (Figure 6.5a) shows the application behavior, where a large number of ALU operations (used to calculate a histogram) have been carried out, but only after global memory accesses complete. We also see how ALU throughput is impacted in Matrix Multiplication (see Figure 6.5), just before workgroups are unmapped.

6.6 Phases within Heterogeneous Computing Kernels

Characterizing the time varying behavior of an application into discrete phases has a number of known uses. By understanding the temporal phase-based patterns present in an application, we can potentially reduce simulation time, improve code generation, build workload models and generate representative input data sets for architectural research [79, 100]. The benefits of workload models which can be built using application phase information have been summarized in Section 3.1.1.

These benefits of phase awareness also apply to kernels executing on heterogeneous devices. However, phase awareness has not yet been evaluated for heterogeneous devices and OpenCL applications. Gaining insight into the time varying behavior of an application executing on heterogeneous devices is challenging due to the following reasons:

- Performance information is presently only available once a compute kernel finishes execution. This is due to the underlying driver model adopted by GPU manufacturers and the standard use of the performance data to assist developers.

- Kernels usually have a short execution duration. Due to this fact, their time varying behavior has not been well studied. However, this trend has been changing with persistent kernel style implementations gaining popularity [108, 80].

The throughput counters (discussed in Section 6.3) measure absolute quantities such as memory accesses and ALU instructions. These quantities are measured using different units, and thus need to be normalized. To quantify these diverse measurements using the same scale, our measurements are normalized to the maximum possible values of each metric. Thus, ALU Utilization for a compute unit at any sample ($t$) can be defined by:

$$\text{ALU Utilization}(t) = \frac{\text{Measured ALU Instruction Issued}}{\text{Maximum ALU Instructions}}$$ (6.1)

In our evaluation, the maximum value used for normalization is dependent on the sampling interval $\delta$. The maximum values can be calculated based on our knowledge of the simulated model and results obtained from running all our sample benchmarks. Similar normalization steps have been carried out for memory inflight and LDS instructions issued.

$\delta$The number of instructions issued would simply scale with time, assuming no other bottlenecks
6.6.1 Phase Generation Methodology

The utilization information shown in Figure 6.5 illustrates time varying behavior of the application as it executes on the device. The plots show how ALU and memory utilization vary over time and the periodic mapping and unmapping of workgroups.

To study the similarities and differences between the behavior of different workloads, the statistics shown in Figure 6.5 need to be reduced to a fixed set of values. This set of values can be treated as a vector that can be used to compare different workloads or decompose a workload into a set of behaviors. We rely on clustering the profiling data in order to get a fixed number of centroids.

Our phase generation is based on K-means clustering [47]. K-means is a common clustering algorithm that has been applied to characterizing execution of HPC applications [81]. Our use of K-means to characterize the time varying behavior of a heterogeneous applications is a contribution of this thesis. K-means has a large configuration space that can drastically affect the quality of clustering. The key parameters of the K-means algorithm are been listed in Table 6.1. The configuration parameters have been applied to implementations in popular open source machine learning packages [59]. Our evaluation consists of the following steps:

1. Obtain throughput counter results from simulation. The metrics captured have been discussed in Section 6.3
2. The normalize the throughput counter values using Equation 6.1. This allows us to capture utilization% values for ALU, memory bandwidth and Local Data Store (LDS) usage.
3. Apply K-means clustering (Equation 6.2) to obtain a set of centroids. The three dimensions for clustering are i) Memory Utilization, ii) ALU Utilization and iii) LDS Utilization.
4. Assign each sample to the cluster whose centroid is nearest to the sample.

\[ \text{Cluster Centroids} = \text{Kmeans(Normalized Spatial Report, No. Clusters)} \] (6.2)

Our results capture the application phase information derived from the throughput counter data, as seen in Figure 6.6. Figure 6.6 show the results when five clusters are used (i.e., when k is set to 5) for the data gathered from each benchmark. The clusters in Figure 6.6 have been sorted by their memory utilization and their cluster ids are assigned, based on the sorting. We see that the clusters are of different sizes for different heterogeneous applications. Based on the size of each cluster, we can quantify the frequency of each behavior within an application’s runtime.
### Table 6.1: Configuration of K-means clustering algorithm.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Chosen Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampling interval</td>
<td>Defined based on simulation parameters. The interval parameter only changes the value of the normalization parameter</td>
</tr>
<tr>
<td># of clusters</td>
<td>This is based on the resolution or # of phases desired</td>
</tr>
<tr>
<td>Initial centroid locations</td>
<td>Initial centroids chosen by randomly selecting observations from the observation matrix</td>
</tr>
<tr>
<td>Distance Metric</td>
<td>Euclidean distance used since all the dimensions utilize the same scale</td>
</tr>
<tr>
<td># of iterations</td>
<td>Maximum number of iterations limited to 100</td>
</tr>
</tbody>
</table>

![MatrixMultiplication Clustered Distribution](image1)

![SobelFilter Clustered Distribution](image2)

Figure 6.6: Location of centroids in MatrixMul(1024) and Sobel Filter(Image10). The size of each cluster denotes the number of samples within the respective cluster.

#### 6.6.2 Comparing Workload Behavior

We have applied our phase generation methodology to study the behavior present within an offloaded kernel, and use it to compare different OpenCL kernels. To understand the time varying behavior of a workload, we would have to compare it against the behavior of other workloads.

In order to cluster behavior data across workloads, we run the different workloads independently and obtain the throughput counter results for each workload. The resultant data is normalized and clustered using Equations [6.1] and [6.2] respectively. We have chosen a set of 4 diverse workloads from the AMD APP SDK and applied our clustering methodology. The workloads have been described in Figure [6.7]. The results of our clustering analysis are shown in Figure [6.7]. To study the correlation between clusters and the workloads, we tag each sample with a workload id. It should be noted that the workload id has not been used as a dimension of our clustering method. We see the distinct clusters formed in Figure [6.7] and show how each workload is distributed within different clusters. Most of the samples of each workload fall within a single cluster.
6.7 Summary

This chapter presented how to utilize hardware support to support profiling on heterogeneous devices. Throughput counters have been introduced as a performance metric that can be used to track utilization of a compute device over time. We considered unique applications of throughput counters in Section 8.3.1, wherein we used throughput counters to detect load imbalance between workloads on different compute units. We have also applied throughput counters to quantify the similarities and differences between different compute kernels.
Chapter 7

Offload Control Unit

7.1 Introduction

Previous chapters of this thesis have discussed throughput counters and how they can be leveraged to describe the utilization of resources within a compute units on a heterogeneous system. However, throughput counters provide limited value to application developers. The performance awareness enabled by throughput counters (described in Chapter 6) are not particularly helpful to guide application developers since they cannot be easily tied to actual source code.

In this chapter, we leverage throughput counters, along with other architectural modifications, to described a new hardware device called the Offload Control Unit (OCU). We describe the development of the Offload Control Unit abstraction. The OCU can be described as a scheduler that can control the scheduling of workgroups onto compute units and controls the mapping of command queues to compute units. The OCU architectural enhancements, along with the accompanying software support, enables tuning of heterogeneous applications, simplifies application development and improves fine-grained application performance.

In this chapter we introduce the OCU and show how the OCU within a heterogeneous system (Section 7.2) can be used to guide scheduling decisions. The architectural enhancements made within AMD Southern Island GPU architecture have already been presented (Section 7.3). We also describe the software API necessary to leverage our proposed architectural enhancements. We have designed a software API based on OpenCL 2.0 (Section 7.4) pipes. We also provide application scenarios that can benefit from our proposed architectural modifications (Section 7.5).
7.2 OCU Interaction Model

Before diving into a discussion on the architectural modifications that make up the OCU, we want to clearly define the host-device interaction presently available on heterogeneous systems. We then present a model of the host-device interaction enabled by the OCU to clarify its role in a heterogeneous architecture. The interaction of the OCU with different compute devices in a heterogeneous system is critical, since it defines the possible applications of the architectural modifications.

The interaction between a host and device in OpenCL applications on available heterogeneous systems is summarized in the finite state machine (FSM) shown in Figure 7.1. The OpenCL API and the graphics driver limits host-device interaction to the FSM, as shown in Figure 7.1. This offload mechanism for heterogeneous systems is exposed to the user using the OpenCL programming API, where compute kernels and host-device data movement commands are asynchronously enqueued. This interaction model simplifies driver-software development for heterogeneous architectures, since low-level details such as interaction with kernel modules and device subsystem initialization is not exposed to the application developer. In this FSM, invoking a kernel for execution on a heterogeneous device becomes equivalent to a function call to a parallel library or spawning pthreads.

The FSM in Figure 7.1 shows that the host CPU is the only resource that can trigger data movement to the accelerator and trigger kernel launches.

![Figure 7.1: A FSM describing the interaction between the host and the accelerator device of heterogeneous systems. The FSM describes interaction as exposed by the OpenCL API.](image)

Figure 7.2 shows how the OCU extends host-device interaction. The OCU serves as an interface between the host and device, as shown in Figure 7.2. The OCU decouples the launch of kernels (from the host) from the execution of kernels on the accelerator.
device. Modern device drivers commonly implement this decoupling to group commands for a device. Grouping commands reduces driver and interconnection network overhead. However, control over this optimization space is not exposed to application, library or runtime developers. The OCU decouples the host and the device with a clean interface. The software support for this interface allows application runtimes to enable host-device interaction optimizations, such as write combining to improve I/O efficiency. The OCU provides architectural support for QoS, dynamic resource partitioning and priority-driven kernel execution.

Nvidia GPUs (e.g., Kepler) have extended host-device interaction by providing for on-device kernel launch. The on-device kernel launch model refers to the capability of a GPU workitem to launch a new kernel to the same GPU (nested parallelism). Support for nested parallelism (on device kernel launch) is exposed within the CUDA-C API. However the CUDA-C API does not allow control over the compute devices used by nested kernels. Due to this lack of low-level control, and the fact that this capability is presently available only on Nvidia GPUs, we leave device launch analysis as future work.

Figure 7.2: FSM describing interaction in an OCU-based heterogeneous system. The OCU decouples the offload of kernels from the host and the launch of kernels on the accelerator.

### 7.3 Architectural Enhancements Proposed for OCU

The interaction model shown in the FSM in Figure 7.2 defines the possible roles of the OCU. In this section, we summarize the architectural modifications proposed to build an OCU for heterogeneous systems. The OCU leverages the counters discussed in Section 6.3 and the scheduling support discussed in Section 7.3.1.
The OCU architectural model is shown in Figure 7.3. The throughput counters denote time-varying behavior of workgroups executing on a compute device. This form of utilization information will be applied to scheduling multiple command queues. In our present implementation we have used it for performance monitoring and workload characterization. The OCU could be implemented using a purely hardware or hybrid hardware-software implementation. The hardware implementation is based simple state machines and contains storage for performance statistics gathered from different compute devices. The main components of the OCU are:

- **Scheduling Unit**: The scheduling unit includes the command queue scheduling capabilities.

- **Command Buffers**: Command buffers on the OCU store workgroup priority and NDRange information. The command buffers would also store compute unit utilization information.

- **Signaling state machine**: A control unit responsible for providing compute units with workgroup id information and performing the workgroup scheduling.

Future directions exploring the implementation possibilities based on hardware or software methods have been discussed in Section 9.2.2. In this thesis we have implemented the OCU support within our simulation framework, but has not carried out a thorough design space exploration that considers power and latency. We leave this exploration as future work.

![Figure 7.3](image.png)

Figure 7.3: A functional model of the main components of an OCU system. The scheduling support is based on the concurrent scheduling mechanisms discussed. The throughput counters are used to provide compute unit utilization.
7.3.1 Architectural Support for Concurrent Command Queues

The OCU interaction FSM (Figure 7.2) and the architectural diagram (Figure 7.3) show the role of the OCU. The OCU provides for flexible compute unit to command queue mapping. The goal is to allow application developers to take advantage of this flexibility and develop new applications to target accelerators such as GPUs. This has motivated improvements to workgroup schedulers for the compute units. To enable scheduling of workloads with uneven workgroups (e.g., possessing multiple NDRanges) to be properly enqueued on multiple command queues, enhancements were made to the base workgroup scheduling subsystem.

Figure 7.4: Motivating Workgroup scheduling for unbalanced workloads. A simple round robin scheduler would lead to delays in command queues.

Heterogeneous applications targeting discrete GPUs have commonly utilized multiple command queues to overlap data movement over the PCIe bus with kernel computation. The granularity of data movement has to be managed by the application developer. Although such optimizations complicate development, they have been used widely since they can yield substantial performance gains [45, 75, 2]. The performance gains are due to the fact that the accelerator is not waiting for data movement. However, multiple command queue support added to modern GPUs does not allow for fair and efficient concurrent kernel execution, since most concurrent scheduling policies targeting the accelerator implement a left-over policy [80]. This policy, introduced for Nvidia’s Fermi GPUs, handles scheduling a complete kernel at one time. The drawback of a left-over policy is shown in Figure 7.4, where a simple round robin schedule would lead to the scheduling of smaller kernels being delayed. The left-over queue scheduling mechanism is inadequate for workloads that can benefit from using prioritizing between commands and when applications possess uneven workgroups across NDRanges.

1Workloads discussed in (Section 4.4)
To enable scheduling in heterogeneous devices of workloads with uneven workgroup distributions, architectural enhancements were made to the workgroup scheduler. An architectural model for the concurrent command queue scheduling system that extends the left-over policy is shown in Figure 7.5. Multiple command queues are used to enqueue kernels to the device. The work-group scheduler chooses a compute unit and a ready workgroup. The lists shown in Figure 7.5 allow for the implementation of a wide range of scheduling algorithms. The architectural modifications leverage concurrent command queue mechanisms and the throughput counters (described in Section 6.3).

The architectural model of the GPU was extended by tracking the source NDrange of each workgroup. Tracking source NDranges is not significant, since workgroup id information needs to be tracked for synchronization purposes already. A subdevice-id for each compute unit records the command queue that it belongs to. The scheduler carries out a resource check before a workgroup can be mapped to a compute unit. Before scheduling a workgroup on a compute unit, the scheduler checks the local memory, wavefront pool entries, vector registers and scalar registers using the expression shown in Figure 7.5. This scheduling step differs from legacy GPU architectures, where the resource requirements of all the workgroups that could be scheduled were homogeneous (for the respective kernel) and could be calculated at the start of the kernel (occupancy). The scheduler implementation uses lists of compute units to keep track of compute unit utilization and their availability. These lists are described below.

- **Available CU List**: Compute units where workgroups of active NDranges can be dispatched.
- **Busy CU List**: Compute units where none of the workgroups on active NDranges can be dispatched.
- **Usable CU List**: A subset of the available list which cannot accept workgroups, since none of the active NDrange subdevice ids match the compute unit subdevice id.

The role of each list using the scheduling algorithm is shown in Algorithm 1. The workgroup scheduler chooses a compute unit and a ready workgroup. Our implementation can be described as a parameterized round robin scheduling scheme. A round robin algorithm can be used since in programming models such as OpenCL, once a workgroup is scheduled to a compute unit, its local memory requirements and wavefront pool entries do not change. Algorithm 1 is parameterized to assume a fixed granularity for populating the idle pool and ready pools. The scheduling code within our simulator has been instrumented with a cost metric that can be used to limit the number of scheduling iterations in each cycle and allows us to study the subsequent performance results. This methodology is similar to the techniques applied to maintain power budgets in power-aware multicore processor [29].
Figure 7.5: The concurrent command queue model for heterogeneous devices. When a workgroup is finished, the compute unit is inserted in the workgroup scheduler’s ready list.

Algorithm 1: Scheduling Algorithm Used for Concurrent Command Queues. The ready pool is filtered for each NDrange when the NDrange is scheduled.
CHAPTER 7. OFFLOAD CONTROL UNIT

7.3.2 Inter-NDrange Atomic Support and OpenCL Pipes

Communication between multiple OpenCL NDranges is challenging to implement since the communication primitives used in OpenCL kernels are built using atomic operations within workgroups of a NDrange. While atomic operations have traditionally been treated as expensive (in terms of performance) operations, recent architectural support has been proposed to improve the performance of atomic operations [80, 122]. Atomic operations have also already been used to support inter-workgroup communication in heterogeneous systems [122].

We leveraged the cache coherency protocol within our simulation framework (NMOESI) to implement support for atomic operations on global memory buffers. Atomic operations across NDRanges is a trivial extension of atomic accesses between workgroups of the same NDrange. We extend current implementations of atomic operations and add a pipe communication primitive. The pipe is used for communication between different subdevices. OpenCL pipes are an integral part of the OpenCL 2.0 standard and provide a convenient software abstraction for leveraging the architectural enhancements making up the OCU.

7.4 OCU Software Support - Subdevices and Pipes

To allow modern frameworks such as Storm and the concurrent command queue applications (Section 4.4) to better utilize heterogeneous systems, architectural enhancements have been proposed. However, architectural enhancements such a new command queue scheduler requires additional software-level support, if we want to reap the full benefits. To leverage this new architectural support in applications, additional software APIs are proposed. We develop a new software mechanism that exposes host-device interaction, partitioning of compute resources and performance awareness can enable a wider range of applications to target heterogeneous devices.

Software support for the OCU was added to our target OpenCL runtime and the device driver for our simulated Southern Islands platform[^2]. The Southern Islands driver within our simulation infrastructure controls the architectural timing simulator. Preliminary software support was added for the following purposes.

1. **Subdevice API:** We provide support for spatial partitioning a compute device between multiple command queues. This support is similar to the Sub-Device API introduced in OpenCL 1.2. Support was also added to describe priorities between command queues. However, neither preemption nor compute unit migration support has been fully implemented in our simulation platform.

2. **OpenCL Pipe API:** We leveraged the atomic support within our simulated platform to develop a pipe communication primitive. The pipe allows communication in

[^2]: Our simulation methodology is discussed further in Section 7.6.
the form of ordered data transfer between kernels on different command queues. Pipe communication is established between different subdevices. Pipeline-based communication is common in multicore systems. OpenCL pipes have been proposed in the OpenCL 2.0 specification.

7.5 OCU Application Scenarios

Equipped with fine-grained control to offload computation provided by the OCU, we can optimize execution for the following application scenarios:

- **Complex Event Processing:** Support for persistent kernels and complex event processing using scheduling for concurrent command queues and pipe based communication.

- **Performance Aware Middleware:** Architecture-level support for frameworks such as Haptic.

7.5.1 Complex Event Processing and Persistent Kernels

The command queue benchmarks discussed in Section 4.3 have served as our benchmarks for evaluating the architectural and runtime support of the OCU. Our target benchmarks have both batch and latency-sensitive computation. In such benchmarks, resource management is challenging. Even though latency sensitive tasks have a higher priority, the batch computations should not starve for execution resources. This lack of resource control is one of the main barriers preventing the utilization of heterogeneous systems in such frameworks.
CHAPTER 7. OFFLOAD CONTROL UNIT

in an online setting. In existing implementations of event processing frameworks targeted for multicore clusters, a number of dedicated subsystems handle scheduling and resource allocation for executing Bolts. For example, the open source tool Zookeeper is used by Storm for synchronization of Bolts.

Complex event frameworks require communication structures such as streams to simplify data movement between execution units. Our OCU implementation also provides communication abstraction of pipes. We have leveraged our implementation of pipes (Section 7.4) to extend the OpenCL kernel offload programming model. OpenCL pipes provide communication to support persistent kernel style of programming. The value of the pipe communication mechanism is evident, especially given the fact that Khronos been included it in the OpenCL 2.0 specification.

7.5.2 Performance Aware Middleware

The throughput counters provide utilization status of the compute units. We propose to apply this information to generate a compute unit’s heartbeat [52]. Processor heartbeats have a number of real world applications, such as tracking application level performance, reliability and throughput. In multicore CPUs, heartbeats are restricted to timestamps, since they are implemented using a rather coarse granularity at the operating system level [52]. The heartbeat information provided by the compute units consist of rich performance information, and includes the individual subsystems discussed in Section 6.3.

A preliminary scheduling algorithm based using heartbeats as input is described in Algorithm 2. For the update interval, the workgroup scheduling subsystem of the OCU queries for the heartbeats of each queue. If the cumulative performance of the compute units in the past interval does not meet the required QOS, an idle compute unit is appended to the resources of the queue. Conversely, if the QOS is being surpassed for a set number of intervals, a compute unit can be removed from the queue.

One of the challenges of this approach is how to express high-level QOS (usually expressed as application-level throughput) in terms of low-level performance metrics. One possible methodology would be to calibrate the system by measuring compute unit utilization when QOS is not exceeded and then proceed to use the measured utilization as a threshold. The answer to this question we leave for future work.

7.6 Evaluation of OCU Feasibility

To evaluate the possible benefits of an architecture-level resource such as an OCU, we evaluate the cost of the architectural modifications required to implement the OCU. We

---

3Streaming frameworks such as storm are also called complex event processing systems.
CHAPTER 7. OFFLOAD CONTROL UNIT

Algorithm 2: Algorithm denotes gathering heartbeats from compute units and selection of an idle compute unit.

also consider the scalability of our OCU-based framework by evaluating performance for a compute-intensive application with a large number of queues.

7.6.1 OCU Evaluation Methodology

Our evaluation requires fine-grained control over workgroup scheduling, control over NDRange creation and robust support for OpenCL subdevices from our target platform. In order to support the OpenCL pipe mechanism, we have implemented these mechanisms in a cycle-based simulator for heterogeneous platforms.

We have implemented both the scheduler and the pipe object model in the Multi2Sim simulator [112]. Multi2Sim is a publicly available architectural simulation framework targeting x86, AMD Evergreen GPUs and AMD Southern Islands (SI) GPUs [112]. Multi2Sim provides software layers defined as “runtime” and “driver”, to execute OpenCL applications on different simulated GPUs. The runtime layer provided by Multi2Sim is a library that exposes the OpenCL API. To execute applications on Multi2Sim, we link our program with the Multi2Sim OpenCL runtime library.

The OpenCL runtime library is simulated on the x86 CPU and invokes the Multi2Sim driver layer for the requested GPU device. A benchmark executing on the simulator invokes
the Multi2Sim OpenCL runtime. This OpenCL runtime invokes the Multi2Sim driver using unused system calls. The architecture-specific OpenCL driver is responsible for communication and initialization of the requested GPU architecture. The driver invokes the functional and timing simulator \[112\] for the requested GPU architecture. For our evaluation, we configure the simulator to use detailed timing-based simulation for a dual-core x86 CPU and an AMD Radeon 7970 Southern Island GPU.

### Table 7.1: Device configuration of the Southern Islands GPU modeled in this thesis.

<table>
<thead>
<tr>
<th>Device Configuration</th>
<th></th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compute Unit Config</th>
<th></th>
</tr>
</thead>
<tbody>
<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>

### Figure 7.7: Memory hierarchy of modeled Southern Islands GPU device.

The modifications discussed in Section 7.3 have been implemented in Multi2Sim. Support for multiple command queues and subdevices is added within the OpenCL runtime layer. Support for tracking resource usage of the compute units and requirements of the workgroups, is added to the driver layer. The architectural simulator is updated to support atomic operations and execution of workgroups belonging to different NDrange kernels.
It should be noted that our methodology does not take into account some of the following factors:

- the time to move data and commands to the GPUs,
- the cost to launch kernels on the GPUs, and
- the cost of system call simulation.

These details are not considered in most architectural simulation frameworks since their overhead is usually insignificant as compared to the time spent executing compute kernels [12, 112]. We ignore data transfer over the PCIe since our interest is studying GPU performance and our benchmark applications' compute time is many times greater than the data transfer.

### 7.6.2 Scalability Study

To evaluate the feasibility of our simulated design, we report on application-level speedup. We apply the concurrent scheduling subsystem to an application scenario with a large number of command queues.

![Multi-Receiver Communication Analysis](image)

**Figure 7.8:** (a) Execution Performance and (b) L2 cache performance of an application with a large number of command queues mapped to different compute devices.

The communication channel application (Section 4.4) can leverage more than one receiver using multiple command queues. The execution performance and L2 cache performance is shown in Figure 7.8a and Figure 7.8b, respectively. We observe that the execution time (in cycles) increases and the number of emulated receivers increases. Each channel
is a separate command queue, each with an equal number of compute units. We also observe that the execution performance of the two emulated receivers is better than the single command queue implementation.

Figure 7.8b shows the L2 cache performance of the applications when we scale the amount of work in the command queues. Our L2 investigation is motivated by the fact that the L2 is the closest shared memory resource to the compute units (see Figure 7.7). The increase in the number of emulated receivers increases access frequency to the input data across all kernels. The L2 cache hit rate is 83% on average and exceeds 90% for 4 emulated receivers, as seen in Figure 7.8b. If we try to use a large number of command queues that are mapped to separate compute units, we will degrade execution performance due to a lack of compute resources.

7.6.3 Scheduling Cost Experiments

Figure 7.9: The effect of restrictive workgroup scheduling on execution performance of GPUs. The number of scheduling iterations at each scheduler invocation is limited.

The OpenCL runtime and driver for our target GPU can allocate a different number of compute units to each command queue. The workgroup scheduler allocates workgroups to available compute units on the device. When invoked, our round-robin scheduler continues until all the active workgroups have been assigned to available compute units.

One of our future goals is to build a more accurate model of the OCU on a full heterogeneous systems. As an initial step, Algorithm 1 is parameterized to enable a fixed granularity for populating the idle pool and ready pools. The scheduling code within Multi2Sim has been instrumented with a threshold value that can limit the number of scheduling iterations in each cycle and allows us to study performance. The threshold restricts the number of workgroups that can be assigned to compute units in one scheduling invocation. When the threshold is reached, the scheduler exits until the workgroup completes execution on the device, thus freeing up resources.
CHAPTER 7. OFFLOAD CONTROL UNIT

Figure 7.9 shows that by restricting the number of workgroups scheduled in a single call to the scheduler, execution performance suffers. We found that allocating 30 workgroups in one scheduling invocation yields stable execution performance for the applications. The study enables us to understand the appropriate number of workgroups which can assigned in one invocation of the scheduler. We see that the scheduling iteration count (which is similar to the number of compute units) provides a good tradeoff between performance and scheduling overhead. Lowering the number of scheduling iterations impacts the utilization of compute units on the device and the resulting performance. Restrictive scheduling can be adopted to save resources on the compute device. This methodology is similar to power budgets maintained in power-aware multicore processors [29].

7.6.4 Overhead of Atomic Operations

Figure 7.10: Execution overhead due to atomic operations in OpenCL pipe.

The pipe object implementation uses atomic accesses to update head and tail pointers of the pipe. We study the overhead imposed by the use of atomic updates for our implementation of the pipe object.

We develop microbenchmarks for evaluating the performance overheads due to atomics. The first microbenchmark kernel updates a location in a memory buffer without using atomics. The global-id of the workitem serves as the offset into the buffer. The second microbenchmark kernel increments a global memory address atomically. The performance overhead due to the atomic access is shown in Figure 7.10. The overhead in Figure 7.10 increases as we increase the number of global workitems. We observe an average increase of 2.3x in execution time (in cycles) for updates using atomics when compared to using non-atomic update. We observe a execution overhead of 16 cycles/workitem from Figure 7.10.

Atomic accesses impose an overhead on the execution time of the microbenchmark. However, due to the significant amount of compute in our real world applications, the overhead observed for atomics should not adversely affect performance scaling and we observe performance improvements for applications using pipes (see Figure 8.13).
7.7 Summary

In this chapter we have discussed the implementation and further details of the Offload Control Unit. We showed how to leverage the counters and software frameworks proposed in Chapters 5, 6. We have evaluated the OCU framework using the benchmarks discussed in Chapter 4. In further chapters we carry out more detailed performance evaluation.
Chapter 8

Performance Results

In this chapter present simulation results for the different workloads and mechanisms de-
veloped in this thesis that have been described in Chapters 4, 5, 6 and 7. This chapter
begins with a description of our target platforms. Subsequent sections have been organized
as follows:

1. We evaluate the Haptic framework and study how analysis devices can be used to
   extend a heterogeneous compute pipeline.

2. The Haptic profiling framework was used to evaluate the Valar benchmarks. As part
   of the evaluation, optimization case studies have been shown that evaluate host-device
   behavior.

3. Preliminary evaluation of throughput counters and how they provide a useful metric
   to identify load imbalance across compute units.

4. Evaluation of queue utilization of a compute device targeted using multiple command
   queues.

5. Performance results evaluating the concurrent command queue subsystem of the OCU
   in an application setting.

8.1 Tested Heterogeneous Platforms

The range of OpenCL-compliant heterogeneous architectures include the following device
categories: 1) discrete GPUs connected to a multi-core CPU device using a PCI bus, and
2) APU devices, such as the AMD Fusion platforms, where the CPU and GPU device share
a single system memory [88]. The primary platforms considered in this thesis are described
in Table 8.1.
CHAPTER 8. PERFORMANCE RESULTS

Our evaluation of the OCU and the throughput counters has been carried out using simulation-driven studies. Our simulation configuration is based on the AMD 7970 GPU and is described in more detail in Section 7.6.

<table>
<thead>
<tr>
<th>Discrete Platforms</th>
<th>APU Platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Radeon 7850</td>
<td>AMD Radeon 7970</td>
</tr>
</tbody>
</table>

| GPU Architecture | Southern Islands | Southern Islands | Northern Islands | Evergreen |
| Device Name | Pitcairn | Tahiti | Trinity | Llano |
| CPU | Number of cores | 4 threads each | 4 threads each | 4 | 4 |
| GPU | Compute units (CUs) | 16 | 32 | 6 | 5 |
| Processing Elements per CU | 64 | 64 | 16 | 16 |
| Register file size / per CU (KB) | 256 | 256 | 64 | 64 |
| Local memory size per CU (KB) | 64 | 64 | 32 | 32 |

Table 8.1: Platforms utilized for Haptic and Valar evaluation.

8.2 Profiling Experiments Using Haptic

We evaluate the performance benefits of Haptic using the applications based on FIR filter and SURF.

8.2.1 Results - Haptic with Adaptive FIR Filter

In this section, we discuss how the adaptive FIR filter workload benefits from an analysis device implemented using Haptic. Three cases of the tap weight modification kernel (100, 200 and 300 iterations) have been considered in order to see how the execution time spent on the tap-weight-modify analysis kernel affects the overall time of the FIR kernel. The baseline, two-command-queue, implementation uses one command queue for the FIR filter and one to provide the tap changes. A single command queue for both the FIR and the tap change would not be possible, as tap-weight modifications are commonly applied asynchronously and having FIR kernels already enqueued when a tap change is requested.
CHAPTER 8. PERFORMANCE RESULTS

a) FIR execution on an APU, Tap weight modification on CPU. (without Haptic)

b) FIR execution on an APU, Tap weight modification on CPU. (with Haptic)

Figure 8.1: Execution time of the FIR compute and tap weight modification application (Figure 4.3) on an APU with and without Haptic. FIR kernel executes 1000 filter blocks.

would lead to incorrect results. This necessitates the separation of FIR and tap change systems.

The cumulative execution time for the two command queues for the FIR application is shown in Figure 8.1a. The charts show three different tap iteration values across four sets of filter block sizes. The hatched portion shows the time spent by the tap weight modify command queue. The FIR kernel’s execution time is highly correlated with filter block size, but is independent of the tap modification. However, there is an increase in FIR time across tap iterations for a fixed block size. This is due to the time spent by the FIR command queue while waiting for the tap-weight-change kernel to complete in the baseline implementation of the application.

When the FIR pipeline has been integrated into Haptic, performance is improved by allowing asynchronous tap configurations and IO, while avoiding multi-command-queue synchronization delays. Using overlapped IO and compute, Haptic is able to improve the overall execution time by an average of 18% (Figure 8.1b) with little user intervention or synchronization tuning.

8.2.2 Results - Haptic in SURF Applications

Frame Preprocessing Analysis Device: The preprocessing analysis device (Figure 4.2c) executes a simple comparison kernel on incoming video frames to disable the feature generation pipeline for similar images to improve the frame rate. Figure 8.2a shows improvements in execution time.
CHAPTER 8. PERFORMANCE RESULTS

a) Effect of preprocessing analysis device (Figure 4.2a) on SURF’s execution time on an AMD APU.

b) Effect of varying the orientation comparison analysis device’s sampling interval on SURF’s performance.

Figure 8.2: Effect of Haptic’s Analysis Devices on SURF’s performance.

As expected, the end-to-end performance of SURF with preprocessing improves with larger thresholds. For a larger threshold, the difference between two consecutive frames must be larger to launch feature generation, thus reducing calls to feature generation. To maintain usability we use realistic thresholds, ensuring that no object visible in the source video is omitted when frames are dropped. The improvement shown includes the time spent in the comparison of video frames. The improvement is proportional to the size of each video frame and the number of frames executed after threshold preprocessing. Similar performance improvements are found for a discrete GPU system as well.

Comparing Orientation: The orientation device (Figure 4.2b) allows us to study the impact of the sampling frequency of the value profiler. The orientation analysis device uses the orientation data produced by the SURF pipeline and disables the orientation stage for a fixed number of incoming frames if the orientation between features did not change. Figure 8.2b shows the results as the sampling interval is reduced.

Intuitively, disabling the orientation stage should improve performance. However, the impact on performance is small in this case since the kernels that would have been skipped in the orientation stage only make up 15% of the execution time. In this particularly scenario, where profiling is active but the resulting optimization applied does not result in substantial benefits, allows us to measure overhead. Using a high threshold disables the orientation stage for only a few frames in each video. From Figure 8.2b we observe that the worst-case overhead (smallest video, highest sampling frequency) is a 17.5% slowdown.

Feature Count Tracking Application: The SURF algorithm’s “threshold” input parameter affects the number of features generated, and thus overall performance. We use the analysis device implemented in Figure 4.2b to track feature counts and the threshold
CHAPTER 8. PERFORMANCE RESULTS

parameter. This device has been applied in two disjoint scenarios. The first use case is to track variations in the number of features generated to automatically detect scene changes. The results for a single video sequence are shown in Figure 8.3a. By maintaining a window of 10 frames (where the average feature count is tracked by the value tracker), changes in the scene can be detected by comparing the window’s average to the feature count of the processed frame. The second use case of feature tracking is a form of feedback parallelism to maintain a constant number of features per frame. Based on the number of features being generated, the threshold can be changed (Fig 8.3b) to reduce variation in the number of detected features.

Performance, report in frames per second (FPS), is shown in Figure 8.3c. We can see that a substantial variation in FPS occurs natively when not using Haptic and while detecting scene changes (due to varying feature counts). Varying the feature threshold parameter stabilizes performance. We found that the feature count tracking for scene changes introduces no significant overhead. The online tracking involves computing a running sum and performing a comparison, and requires very little data movement.

Figure 8.3: Feature count tracking for (a) detect scene change (b) regulate feature count.

8.3 Valar Characterization

The Haptic framework provides a mechanism to profile the behavior of a heterogeneous application. Haptic allows an application developer to evaluate the performance of a heterogeneous computation pipeline when it is embedded in a real world application. However, the evaluation possible using Haptic is restricted to the granularity of an OpenCL kernel.
This restriction is due to the nature of OpenCL events, where only timestamps are available per kernel.

Our study of Valar extends the evaluation done in Haptic by examining resource usage within an OpenCL kernel. In our evaluation of the Valar benchmarks we study the coverage of OpenCL kernels and the variations in resource utilization of OpenCL kernels. The Valar benchmark consists of a set of applications with a range of possible host-device interaction behaviors. We leverage the Valar applications to study host-device interaction and evaluate the optimizations possible.

### 8.3.1 Characterization of Kernel Execution using Valar

Evaluation of the OpenCL kernels within Valar is necessary to check the coverage of Valar’s kernels over a range of the execution metrics. The coverage of Valar benchmarks has been previously discussed in detail [75].

### 8.3.2 Characterization of Host-Device Interaction using Valar

In these experiments, we study the interaction of the host and the device. The interaction between the host and the device can be considered as a AL1 optimization (see Section 3.4), since it affects how the host and the device share data for the same application. In the following sections we have leveraged Valar to study the benefits of streaming data movement optimizations and studied the effect of host-device interaction on application performance.

#### Investigating Streaming Optimizations - FIR

In this section, we demonstrate Valar’s ability to study the heterogeneous system optimization space with respect to data movement.

The Adaptive FIR application is controlled by an input argument Dispatch (Table 4.3) that can be used to vary the granularity of work offloaded to the compute device. Dispatch denotes the number of filter blocks batched into an OpenCL buffer before initializing execution on the GPU. Batching data transfers is a commonly implemented optimization. Batching data transfers on GPU architectures typically increases throughput due to more efficient utilization of the PCIe bus and amortization of kernel launch overheads.

To study the true benefits of batching using the adaptive FIR application, we varied the dispatch size of the filter. The results are shown for the discrete GPU and the APU in Figure 8.3a and 8.4b, respectively. The filter block size shown in Figure 8.4a denotes the size of each block and is the minimum unit of work done by the GPU device. We see that the execution time reduces to a set level, which is dependent on the filter block size. For larger dispatch sizes, shown in Figure 8.4a multiple filter blocks are being offloaded to the GPU at a time.

---

1Coverage refers to the range of performance statistics in the kernels measured via profiling.
The results in Figure 8.4 show why the performance of APU devices is more dependent on the Filter Block Size versus when running on discrete devices. Figure 8.4 shows the time spent in data movement and kernel computation. On the discrete GPU, the time spent moving data substantially reduces by up to 30% as dispatch size increases. This is due to a reduction in the number of transactions and a better utilization of the PCIe bus bandwidth. In the APU case, due to the smaller number of compute units, the computation time is much larger than the data management time. These results lead us to conclude that streaming IO optimizations such as batching are not as beneficial for APU devices due to the low count of compute units (Table 8.1) and low cost for data movement.

Adaptive FIR has a tap weight modulation component (evaluated in Section 8.2) concurrently executing on the CPU. However, it is a lightweight CPU-based OpenCL kernel that only changes the value of the filter taps. Generally, fewer than 1024 filter taps are used in real-world applications. The tap weight modulation does not get affected by the batching since it only operates on the taps of the filter.

**Multi-Device Coupling Effects - Search**

In this section, we study the effect of the interaction between IO and compute on compute devices in different heterogeneous systems.

The *Search* benchmark (Section 4.3) searches for a set of target data values in blocks of data using a GPU-based OpenCL kernel. The application hands off the data that was found back to the CPU for a final reduction step. *Search* has a parameter to vary the frequency

Figure 8.4: Variation in execution time for the Adaptive FIR application.
CHAPTER 8. PERFORMANCE RESULTS

sending results from the GPU device to the CPU. To study the effects of communication between the CPU and GPU on discrete and APU platforms, we vary the frequency at which the CPU device reads in data from the GPU’s buffers. The results are shown in Figure 8.5.

Figure 8.5: Search execution characteristics for a range of communication intervals between the CPU and the GPU compute devices.

Figure 8.5 shows the variation in throughput of search when the frequency of communication between the CPU and GPU device is varied. From Figure 8.5a, we see that the performance of the GPU is more stable than the GPU across communication intervals. This occurs in search, since for less frequent communication, the GPU allocates more data to search through for the target values. However, the time to carry out the data reduction on the CPU is independent of the communication interval since the CPU’s work depends on the number of target elements being searched for by the GPU. In Figure 8.5b, the GPU execution time per kernel increases as the communication frequency is reduced. The discrete GPU performs better for the search kernel due to higher memory bandwidth and a larger number of compute units. We also see that for a shorter communication interval...
(i.e., frequent communications), the CPU kernel on the APU behaves poorly. This can be attributed to the shared memory of an APU, where the memory bandwidth is shared by the GPU and CPU on the APU. We also observe that our CPU kernel may need further tuning. The kernel does not efficiently utilize the multithreaded CPU cores in the Pitcairn system, since CPU kernel’s performance is comparable to the quadcore system.

We see that the effective throughput is comparable on both platforms for shorter communication intervals. However, for less frequent communications, the high throughput of the discrete GPU and the lower usage of the PCIe bus increases performance more than for the APU.

Multi-Device Coupling Effects - Physics

The Physics application (Section 4.3) partitions the collision detection pipeline between the CPU and the GPU. We observe the effects of partitioning a compute pipeline on two heterogeneous platforms.

The Physics application has been studied by observing the resultant throughput of the application for different small::large particle ratios. Figure 8.6 shows the results on the discrete AMD Southern Islands GPU (Tahiti) system and the Trinity APU. We have chosen three values for the number of small particles and vary the number of large particles for each testcase. As expected, we observe that the time taken by the simulation increases.

![Figure 8.6: Throughput for Physics for SI discrete GPU (Tahiti) system and Trinity (APU). As we reduce the number of large particles, performance difference between the platforms is grows.](image)

To understand the difference in throughput between a discrete GPU and a APU system, we examine the time taken by each to perform the large particle collisions. The large particle collisions are a CPU-intensive task. Figure 8.7 shows that the throughput of the large-large particle interaction for both systems, while fixing the number of small particles. We see that the systems have comparable throughput. The Tahiti system has higher throughput on the large-large calculations due to the larger number of CPU cores in the system.
Thus we see that when we run the application with the maximum number of small particles (57,344), the discrete GPU Tahiti system has better performance by up to 2X when the number of large particles is small, and the speedup reduces to 1.3X as we increase the number of large particles.

![Figure 8.7: Large particle throughput for increasing large particle counts for the Tahiti and Trinity systems.](image)

To summarize, in Section 8.3.2 we discussed how optimization of interaction between the host and the OpenCL devices enables up to 30% improvement and 15% improvement in throughput on discrete and APU systems, respectively. We observed that for Search, effective throughput of the discrete and APU systems was comparable when the devices were closely coupled. For Physics, we see that the lower throughput of the large particle collision computation reduces the performance advantage provided by a discrete GPU.

### 8.4 Additional Throughput Counter Performance Results

The performance counters discussed in Section 6.3 can be applied to track utilization of compute units over time. We have used throughput counter data to also evaluate the time varying behavior of a workload (see Section 6.6). To examine the utility of our throughput counters and evaluate their ability to measure compute unit utilization, we examine the performance of NDranges with unbalanced work across workgroups.

#### 8.4.1 Generating Workgroup Imbalance to Study Throughput Counters

To observe compute unit utilization over time, we generated kernels where the individual workgroups carry out a varying amount of work. These workgroups, when scheduled onto compute units using a round robin scheduler, would result in an uneven amount of computation dispatched on each compute unit. Such workloads can be contrasted with balanced workloads, where all of the workgroups carry out a similar amount of computation on their respective data points. Matrix Multiplication is an example of a regular workload, where
each workgroup, processes its own block of data, but all the workgroups execute the same number of arithmetic and memory operations.

Figure 8.8 shows two mappings of computation across workgroups in a single NDRange. The possible mappings are shown as Mapping 0 and Mapping 1. The target architecture discussed is the AMD 7970 discussed in Table 8.1. Both distributions show the same amount of work assigned to the 128 workgroups. Mapping 0 in Figure 8.8 shows a periodically increasing work-distribution. Mapping 1 shows a distribution where the amount of work is same for 4 consecutive workgroups, before it increases.

These distributions allow us to implicitly control the assignment of compute to the workgroups since we use the simulator’s original workgroup round-robin scheduling policy. The mapping of work to compute units for Mapping 0 and Mapping 1 is shown in Figure 8.8b. Mapping 0 causes substantial load-imbalance since workgroups with less computation get
scheduled on different wavefront pools of the compute unit. In contrast, Mapping 1 shows a similar work-distribution among all the compute units since the workgroups follow a smooth distribution of work. In Section 8.4.2, we examine how throughput counters capture the compute unit utilization due to workgroup imbalance.

8.4.2 Variation in Utilization

The variation in compute unit utilization for the two workgroup distributions shown in Figure 8.8 is discussed below. The goal of studying the variation in utilization across compute units, for known workgroup distributions, is to verify that throughput counters provide an accurate metric to guide the work to be scheduled on each compute unit.

Utilization variation for three compute units is shown in Figure 8.9 and Figure 8.10 for Mapping Distribution 0 and Mapping Distribution 1, respectively. Figure 8.9 shows the utilization corresponding to Mapping Distribution 0. We see that the variation in work mapped to each compute unit unit is reflected in the utilization statistics. Compute unit 2 finishes computation much before Compute Units 2 and 10, as shown by the interval when its input workgroups are unmapped. We also see similar information from the memory

![Figure 8.9: Variation in compute unit utilization over time for Vector Add Imbalance Benchmark with Mapping Distribution 0.](image-url)
inflight statistics.

In contrast, mapping distribution 1 shows a balanced distribution of work between the different compute units. The utilization information for Mapping Distribution 1 is shown in Figure 8.10. We see that there is a smaller variation in the utilization statistics. Compute units 2, 10, and 31 process workgroups and have memory accesses in flight through the kernel execution duration. The workgroups are unmapped by the compute units at distributed intervals.

Figure 8.10: Variation in compute unit utilization over time for Vector Add Imbalance Benchmark with Mapping Distribution 1.

The throughput counter results show how the measured performance statistics can be utilized to judge the utilization of a compute unit on a heterogeneous system.

8.5 OCU Subsystem Evaluations

We have further evaluated the concurrent command queue subsystem of the Offload Control Unit using microbenchmarks and real world applications...
8.5.1 Concurrent Command Queue Execution - Microbenchmarks

The results in Section 8.3 show how throughput counters can be used to detect workload imbalance across compute units on a heterogeneous system. We present results showing command queue utilization for applications with multiple OpenCL command queues. Our concurrent command queue execution results are also based on modifications to Multi2Sim [112]. We do not include results on native platforms since the AMD Southern Island GPUs do not expose OpenCL device partitioning.

In compute intensive HPC applications, each compute device is usually targeted with a single command queue, since the kernel consumes all the resources within the GPU. However, the efficiency of devices with multiple command queues mapped to a set of compute units has not been well studied. The utilization of an OpenCL command queue within an interval \( (T_{start} - T_{end}) \) where \( k \) NDRange commands have been completed on the queues is defined in equation (8.1).

\[
Utilization\% = 100 \times \frac{\sum_{i=0}^{k} KernelDuration[i]}{(T_{end} - T_{start})} \quad (8.1)
\]

Utilization of a queue is dependent on the scheduler that dispatches work to the command queue. As per equation (8.1), an ideal scheduler would result in 100% utilization. The utilization of an OpenCL command queue shows the importance of scheduling kernels in applications with multiple queues and timing constraints. In presently available heterogeneous systems, the OpenCL command queue is simply a pipe which abstracts the underlying software stack from the application developer. To study queue utilization, we define the following two experiments:

- **Experiment 1**: Enqueue a similar amount of work per kernel on each command queue.

- **Experiment 2**: Enqueue a different amount of work on each command queue.

**Experiment 1**: The size of the vector operated on by each enqueued kernel is varied and the results are shown in Figure 8.11. The results in Figure 8.11 show utilization of the command queues. In each case, an equal number of kernels is offloaded to all the command queues. As seen in Figure 8.11, the utilization of the command queue is reduced since the time spent context switching between multiple contexts becomes a significant portion of the overall execution time.

**Experiment 2**: The results in Figure 8.12 show the performance effects of varying the amount of work enqueued on each command queue. The number of inner-loop iterations for each enqueued kernel is passed as an argument that is read from global memory. This allows us to vary the number of ALU operations in each of the kernels that is enqueued. In Figure 8.12, \( Q0 \) denotes the command queue with the largest number of ALU iterations.
Figure 8.11: Expt 1: Command queue utilization for three kernel input sets. The target platform has 6 queues ($Q_0 \ldots Q_5$), with 5 compute units per queue.

Figure 8.12: Expt 2: Command queue utilization with 6 CUs mapped to each command queue. The variation in ALU operations is larger in Case 2 vs Case 1.

(assuming longer kernel execution), while $Q_3$ indicates the smallest number of ALU iterations. We observe reduced utilization for $Q_3$, as the imbalance between workgroups is increased. It should be noted that this reduction in utilization is not due to resource contention in the memory hierarchy, since the kernels enqueued are highly compute bound. The reduction in utilization is also not due to contention for compute units, since each command queue is assigned a separate set of compute units.
These performance results in scheduling multiple command queues helps motivate future research on architectural support for workgroup scheduling. Our results show that the existing implementation of software host threads handling OpenCL command queues and round-robin scheduling are not scalable for future workloads. We will need better scheduling schemes to deal with emerging workloads containing a time-varying number of workgroups that are enqueued on a heterogeneous device.

### 8.5.2 Concurrent Command Queue Execution - Applications

We evaluate the efficiency of multiple command queue mapping by comparing the execution performance to the performance obtained by using a single command queue mapping. Applications from the Set 1 benchmarks (Section 4.4) are evaluated for three input sets.

The execution performance is shown in Figure 8.13. The single queue baseline has the entire GPU available for each kernel. Applications using multiple command queues exhibit a performance speedup of 3.6x on average, as observed in Figure 8.13. Simultaneous execution of kernels overlaps the computation of each kernel, and reduces execution time of the application.

Applications such as Communication Channel Analyzer enqueue 4 kernels on the same device for simultaneous execution. Each kernel utilizes 8 unique compute units available on the device. A speedup of 4.3x over a single queue is observed for the largest input size. Matrix Equation Solver enqueues 2 kernels operating on same input data for two unique computations. Results of both of the kernels are used for next stage of compute. The Clustering application also uses a similar design of enqueuing 2 kernels for simultaneous execution. Both applications show a performance gain of 3.2x over the single command queue implementation. The improvement in performance can be attributed to full utilization of resources available on the device since the single queue does not utilize the device.

![Figure 8.13: Performance improvement when using multiple command queues mapped to different compute unit sets on the same compute device for Set 1 benchmarks.](image)

**Cache Utilization:** We explore performance of the L2 cache on the GPU with the use of multiple command queues. Our L2 investigation is motivated by the fact that the L2 is the closest shared memory resource to the compute unit (Figure 7.7).

As seen from Figure 7.7, all compute units have one L1 cache unit each and share six L2 cache units. Applications using multiple command queues execute on different sets of
CHAPTER 8. PERFORMANCE RESULTS

Figure 8.14: L2 cache efficiency for applications implemented using multiple command queues for Set 1 benchmarks.

calculate units. Kernels which utilize the same input data benefit from the shared L2 caches. The L2 cache efficiency is evaluated in Figure 8.14. Applications such as Matrix Equation Solver, Communication Analyzer and Clustering show a 22% improvement in terms of the L2 cache hit rate when using multiple command queues. The single command queue implementation schedules a cache flush after completing the execution of each kernel on the compute device. This reduces the cache performance of subsequent kernels operating on the same input data. The Search application enqueues two kernels, which operate on two separate halves of the input data. Hence, both the kernels do not access the same data. This affects the L2 cache hit-rate for the Search application over its single command queue mapping version. The higher hit rate for our multiple command queue implementation lowers the number of memory accesses to global memory, thus improving execution performance.

8.5.3 Performance Evaluation of Pipe Memory Object

Next, we evaluate the performance of the pipe object (Section 7.4). Applications with multi-stage computations can benefit greatly from the use of a pipe object and multiple command queues. Kernels belonging to different stages of compute can be spatially partitioned across the compute device. The communication among these simultaneously executing kernels is achieved using pipes. Figure 8.15 shows the design of an application utilizing pipes for communication. The producer kernel processes the input data and stores the output in a temporary buffer. The data/tile ID of the processed data is written to the pipe by the producer kernel. The consumer extracts the data/tile ID from the pipe and accesses data from the temporary buffer using the extracted ID as an offset into the buffer. Both the kernels execute on separate command queues on the same device, and thus, maintain a communication channel.

We evaluate the execution performance and L2 cache performance of the Set 2 benchmarks as described in Section 4.4. The serial baseline in this experiment is running the OpenCL kernels for the application within a single command queue. The data sharing is done using a global memory buffer, indexed by a workitem’s global id. The Figure 8.16
CHAPTER 8. PERFORMANCE RESULTS

Figure 8.15: Application design utilizing a pipe object to communicate with the next stage of computation. Tid is the tile-id of the data.

Figure 8.16: Performance of multiple command queues using a pipe object for communication between mapped kernels for Set 2 benchmarks. Shows the execution performance of the Set 2 applications. Both applications communicate shared data using the pipe object between stages of computation. A 3.7x speedup is observed over the single command queue implementation. Due to data dependencies

Figure 8.17: L2 performance for Set 2 benchmarks using a pipe object. Kernels using pipes are mapped to multiple command queues.
present between stages, the single command queue implementation completes all updates to the dependent data before enqueuing the next kernel. The pipe object allows for such kernels to reduce the number of kernel calls and to overlap computation and communication effectively.

Figure 8.17 shows the L2 cache performance for the applications using a pipe object. We observe a 35% increase in L2 hit rate for applications using pipe objects. Both applications achieve a L2 cache hit rate of more than 80% for all input data sizes. The improved hit rate is attributed to the number of accesses to the data buffers shared across simultaneously executing kernels.

8.6 Summary of Performance Results

This chapter has covered a range of performance results based on both the benchmarking contributions and the architectural contributions of this thesis. The performance results presented in this thesis demonstrate the benefits of Haptic. The results also demonstrate the usefulness of the Valar benchmarks in evaluating the behavioral space of heterogeneous applications. The results show that the throughput counters capture the time-varying behavior of a heterogeneous workload over an application’s execution. Finally, our results also motivate future work towards developing additional architectural support for concurrent command queue execution.
Chapter 9

Summary and Conclusions

The results discussed in Chapter 8 demonstrate the importance of research into heterogeneous applications, profiling frameworks and the possible roles of the OCU. The benchmarks discussed in Chapter 4 and the design space exploration in Chapter 8 also demonstrate the importance of a flexible benchmark suite to study the range of behaviors possible within heterogeneous applications.

In this chapter, we summarize the work completed for this thesis and provide an overview of the directions of future research enabled. This thesis can drive further research in performance counters, OCU design and benchmark design:

9.1 Completed Work

The completed work of this thesis is summarized below.

1. **Benchmarks**: The novel contributions of Valar include an evaluation of the optimization space of host-device interaction (Section 8.3). The implementation of the Valar benchmarks is covered (Chapter 4). Contributions have also been made in the form of additional benchmarks to evaluate concurrent command queues (Section 4.4).

2. **Application Models** Multi-layered application models to study the behavior of heterogeneous applications (Chapter 3).

3. **Performance Aware Middleware**: Demonstration of the profiling capabilities of OpenCL events for applications built using heterogeneous computation pipelines (Chapter 5). A novel OpenCL-based profiling framework (*Haptic*) was developed. *Haptic* was evaluated utilizing the Valar benchmarks as computational pipelines and extending them into new applications (Section 8.2.1, 8.2.2).

4. **Utilization Statistics**: Definition of profiling granularity and metrics that serve as throughput counters (Sections 6.2, 6.3) and measured compute unit utilization. These
counters have been evaluated by modifying the Evergreen and Southern Island GPU models in Multi2Sim.

5. **Throughput counters**: Counters have been validated using the AMD APP SDK benchmarks. The goal of our experiments with throughput counters was to study the time-varying behavior of OpenCL applications. Throughput counters were also evaluated to assess their capability to detect load imbalance between compute units executing different workgroups (Section 8.4.2).

6. **OCU Scheduling Subsystems**: Design, implementation and evaluation of a novel concurrent command queue scheduling model on AMD SI GPUs (Section 7.3.1). The command queue model is highly parameterized to manage the size of the scheduling pool. The scheduling model also includes a simple scheduling cost metric that can be utilized to minimize the scheduling loop iterations.

7. **OCU Software Support**: Software support in Multi2Sim for exposing the concurrent command queue implementation to OpenCL programs (Section 7.5). This implementation is novel since no GPU OpenCL platform exposes concurrent execution of compute kernels. Our evaluation included the required software support for mapping upcoming frameworks in social networking and data mining to heterogeneous platforms. The goal of studying such frameworks is to leverage the OCU. The motivating framework’s (Storm in Section 4.6) applications have been studied.

8. **OpenCL Pipe Support**: As part of the OCU’s software support OpenCL pipes were added to Multi2Sim. Pipes provide communication support for applications leveraging concurrent command queues. Pipes simplify application development by reducing synchronization and also improve performance by improving cache utilization.

### 9.2 Future Research Directions

The following sections explain directions for future work.

1. Developing more accurate models for the architectural modifications that generate the utilization information using throughput counters.

2. Evaluating OCU configurations and exploring the possible implementation methodologies (hardware or software).

3. Extending the Valar benchmark suite and building a test case generator
9.2.1 Throughput Counters and Profiling Architectures

The throughput counters discussed in Section 6.3 can be implemented as a distributed profiling system that can be added on a variety of OpenCL capable devices. Our profiling system has been implemented as a simplistic model within Multi2Sim. The recording of performance information adds no cycle overhead within the simulator’s results and thus does not reflect true application performance. The overhead of resetting the counters is not modeled. Improving the accuracy of the recording model for heterogeneous architectures would also allow us to understand the overhead of gathering counter data. An accurate architectural model for performance counters would also allow for the evaluation of the power overheads of profiling.

9.2.2 The Offload Control Unit

By allowing for a variable CPU:GPU resource mapping using multiple command queues, the OCU enables new heterogeneous applications. The OCU can be assumed to be a domain specific processor whose main role is handle host-device communication. OCU-aware runtimes targeting heterogeneous systems configure the OCU to carry out runtime optimizations such as reconfiguring command queues, using feedback of the capacity information or changing dispatch frequency.

The OCU Applied towards Batch and Latency Topologies

Workloads with both batch and latency performance requirements are common in data mining workloads and social networking systems [21, 68]. For example, in social networking systems, trending topics have to be propagated with minimal latency, while committing updates to persistent records such as personal information can be grouped into batches. By enabling concurrent execution on a shared pool of data, we attempt to develop architectural support for such workloads to utilize heterogeneous systems. By adapting the command queue to compute unit mapping for long running kernels, the OCU fulfills a role similar to the Zookeeper library discussed in Section 4.6. Additional software support for OpenCL topologies is required, as was shown in Figure 9.1.

The benefits of the feedback-directed performance information using only software have been described for Haptic in Section 8.2. The OCU can also be used to optimize more of the Valar Benchmarks discussed. For example, dispatch granularity optimizations have been discussed in Section 8.3 for streaming benchmarks, such as the adaptive FIR. Given the adaptive behavior of the search benchmark, it could easily benefit from profiling.
CHAPTER 9. SUMMARY AND CONCLUSIONS

Figure 9.1: A Storm topology with Bolts implemented with OpenCL computation. The topology would be implemented using the proposed software support for the OCU.

**OCU Design and Implementation methodology**

The OCU has been explored using simulated models and evaluation of concurrent command queues on a simulated GPU. Our future work includes developing detailed architectural models. A purely software implementation of OCU functionality and the OpenCL-based topology API shown in Figure 9.1 can be easily built by extending Haptic.

The implementation of the OCU and the host-device interface can be part of a multiple-ISA heterogeneous system. In such a system, x86 cores are not distracted by the management of a heterogeneous device, but instead can focus on managing external sources of data such as databases and sensors. The OCU, due to it’s flexible role, can be implemented as a separate hardware entity or a lightweight processor. A sample OCU implementation could contain:

- X86 CPU: An external load store engine to handle databases
- ARM CPU: The OCU to handle device management and data movement
- AMD SI GPU: Data parallel computation horsepower

Server platforms have been recently announced where a number of APUs share an interconnect. The goal of server-style APU platforms is to enable the GPU’s subsystem of the APU to accelerate web-scale workloads such as image-search, analytics, etc. However, production server software is commonly implemented using a combination of different software frameworks [68, 85] which cooperate to achieve a goal. Research has not addressed management of heterogeneous devices on server platforms, where a number of low-power
heterogeneous systems communicate as part of a computation. A hardware or software-based OCU implementation could play an important role in such a system as part of a control network between heterogeneous devices.

9.2.3 Extending and Generalizing the Valar Benchmark Suite

The Valar benchmarks have been derived from different research applications [73, 74, 75]. The main barrier to the more general usability of the Valar benchmarks among architects is the difficulty in generating benchmark configurations.

The application models discussed in Chapter 3 can be utilized to design representative benchmarks and behavior scenarios across applications for heterogeneous architecture research. Future work includes the design of a test generator that, given host-device interaction requirements (AL1 model), allows the generation of benchmark configurations. This would require building additional input data sets for all benchmarks, and would include the infrastructure to quantify their host-device interaction.

Deriving a benchmark from a real world application is a challenging task and has been examined in the context of CPU benchmarks such as PARSEC [19] and SPEC [33]. Multi-core benchmark suites examine the coverage of reduced input set sizes with respect to native sizes. Due to the range of native behavior in heterogeneous applications, the classification discussed in Chapter 3 is necessary. The classification can be applied towards additional case studies, which could consider how a computer architect and a domain specialist together can convert a heterogeneous applications into benchmarks.

9.3 Publication List


2. Pact paper submitted


1Native behavior varies by platform, input parameters and algorithm usage scenarios.


Bibliography


