HIERARCHICAL SCHEDULING AND UNIFORM ACCESS PROGRAMMING FRAMEWORKS FOR HETEROGENEOUS CPU-GPU COMPUTING CLUSTERS

A Dissertation

presented to

the Faculty of the Graduate School

at the University of Missouri-Columbia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

---

by

KITTISAK SAJJAPONGSE

Dr. Michela Becchi, Dissertation Supervisor

JULY 2015

The undersigned, appointed by the dean of the Graduate School, have examined the dissertation entitled

HIERARCHICAL SCHEDULING AND UNIFORM ACCESS PROGRAMMING FRAMEWORKS FOR HETEROGENEOUS CPU-GPU COMPUTING CLUSTERS

presented by Kittisak Sajjapongse,

a candidate for the degree of doctor of philosophy,

and hereby certify that, in their opinion, it is worthy of acceptance.

Dr. Michela Becchi, Assistant Professor, Department of Electrical and Computer Engineering

---

Dr. Guilherme DeSouza, Associate Professor, Department of Electrical and Computer Engineering

---

Dr. Tony Han, Associate Professor, Department of Electrical and Computer Engineering

---

Dr. Prasad Calyam, Assistant Professor, Department of Computer Science

ACKNOWLEDGEMENTS

This work is completed with the help and support from those whom I sincerely respect. Without their support, my efforts could have been far from complete. I am fortunate to have them along the course of my studies and my future career.

My mentor: I would like to deeply thank my advisor, Michela Becchi, for her support and mentorship. I am fortunate and grateful to have worked with her during my studies. The association with Dr. Becchi trained me to think systematically in both professional and personal aspects. I also appreciate her generosity and flexibility in offering me opportunities to interact with other scientists and to explore the research community.

My Colleagues & Friends: Thank you to all the current and previous members of the Networking and Parallel System laboratory, especially Da Li, Ruidong Gu, Huan Troung, Tejaswi Agarwal, and Xiang Wang for working together through difficult times and for their friendship. Thank you to my dear friend and collaborator, Vignesh Ravi, for his valuable discussions and great opportunities. I would also thank to Fadi Muheidat who is very generous and could always keep me positive.

My family: I would like to express my deepest appreciation and thanks to my parents and sisters: Adisak, Tommarat, Jurairat, and Naovarat Sajjapongse for their unconditional love and encouragement. I am very grateful to have a supportive family who puts my education as the first priority.

My wife: My wife, Sasiwimon Yoo-eam, is a great researcher with whom I can always share ideas even though we focus on different disciplines. Thank you for your love, support, staying beside each other, and sharing our lives together.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ... ii

LIST OF ILLUSTRATIONS ... vii

LIST OF TABLES ... x

ABSTRACT ... xi

Chapter

CHAPTER 1 INTRODUCTION ... 1

1.1 Introduction ... 1

1.2 Motivations ... 2

1.2.1 Software support for GPUs ... 3

1.2.2 Resource Managers ... 5

1.2.3 Programmability ... 7

1.3 Contributions ... 9

1.3.1 Runtime Supports for Distributed GPU applications ... 9

1.3.2 Hierarchical Resource Management ... 9

1.3.3 Programming Framework for Uniform Resource Access ... 10

1.4 Organization ... 10

CHAPTER 2 BACKGROUND AND RELATED WORK ... 12

2.1 Background ... 12

2.1.1 Graphic Processing Units (GPUs) ... 12

2.1.2 Programming Models for Heterogeneous Clusters ... 14

2.1.3 Message-Passing Interface (MPI) ... 17

2.1.4 Partitioned Global Address Space (PGAS) ... 18

2.2 Related Work ... 21

2.2.1 Node-level Schedulers and Runtime Support 21
2.2.2 Cluster-level Schedulers and Resource Sharing................................23
2.2.3 Programming Models for Heterogeneous Clusters 26
CHAPTER 3 NODE-LEVEL RUNTIME...28
3.1 Objectives 28
3.2 Reference Architecture 33
3.3 Scheduling Policies 35
3.3.1 Batch Scheduling...35
3.3.2 Controlled n-way GPU Sharing 36
3.3.3 Preemptive Sharing 40
3.4 Use Cases for Preemption 41
3.5 System Design 44
3.5.1 Overall Design...45
3.5.2 Context Queues 46
3.5.3 Connection Manager 46
3.5.4 Dispatcher..47
3.5.5 Virtual GPUs 49
3.5.6 Memory Manager 50
3.5.7 Fault Tolerance & Checkpoint-Restart...57
3.5.8 Inter-node offloading..58
3.6 Supporting Preemptive GPU Sharing 59
3.6.1 Defining the Preemption Policy 59
3.6.2 Implementation..60
3.7 Experimental Results 63

3.7.1 Single-process Application 63
3.7.2 Multi-process Application 74
CHAPTER 4 CLUSTER-LEVEL SCHEDULER 90
4.1 Objectives 90
4.2 Cluster-level Scheduler 92
4.2.1 Scheduler Architecture 95
4.2.2 Scheduling API 97
4.3 Scheduling Policies 101
4.3.1 Co-locating Scheduler 102
4.3.2 Latency-reducing Scheduler 106
4.4 Experimental Results 108
4.4.1 Benchmark Applications 108
4.4.2 Experimental Setup 112
4.4.3 Analysis of Co-location 112
4.4.4 Determining the Weights for the Nodes 113
4.4.5 Experiments on Heterogeneous Workloads 115
CHAPTER 5 INTER-NODE VIRTUAL MEMORY PROGRAMMING MODEL 118
5.1 Objectives 118
5.2 Background and Motivations 120
5.2.1 Traditional Programming Models 120
5.2.2 Load Balancing 121
5.2.3 The IVM Programming Framework 123
5.3 IVM Framework Design 124
5.3.1 Execution Model 124

5.3.2 Memory Model 125

5.3.3 System Design 127

5.3.4 GPU Support 130

5.3.5 IVM Programming Interface 133

5.3.6 Integration to Higher-level Scheduler 136

5.4 Benchmark Applications and Load Balancing 137

5.4.1 Load Balancing Schemes 137

5.4.2 Benchmark Applications 140

5.5 Experimental Evaluation 142

5.5.1 Experimental Setup 143

5.5.2 Dynamic Spawning Load Balancing (DS-LB) 145

5.5.3 Online Monitoring Load Balancing (OM-LB) 147

5.5.4 Discussion 149

CHAPTER 6 CONCLUSION & FUTURE WORK 150

BIBLIOGRAPHY 157

VITA 164

LIST OF ILLUSTRATIONS

Figure Page

Figure 2-1 – Memory model between CPUs and GPU 12

Figure 2-2 – GPU architecture under CUDA framework 13

Figure 3-1 – Two deployment scenarios for our runtime: (a) VM-based cloud computing service and (b) HPC cluster resource manager. 29

Figure 3-2 – Operation of different scheduling mechanisms in the presence of multi-tasks distributed applications with synchronizations and *intra-application* imbalance. Application A consists of 4 tasks and includes 2 GPU execution phases, with a global synchronization at the end of each. Application B and C consist of two tasks and a single GPU execution phase, also ending with a global synchronization. Synchronizations are represented through (red) dashed vertical lines. A_{jk} represents k^{th} GPU phase of task j belonging to application A. Idle GPU times are represented in black. 38

Figure 3-3 – Operation of different scheduling mechanisms in the presence of multi-tasks applications with synchronization and *inter-application imbalance*. Application *A*, *B*, and *C* consist of 3 tasks and include 3 GPU execution phases, with a global synchronization at the end of each. A_{jk} represents the kth GPU execution phase of task *j* belonging to application *A*. Idle GPU times are represented in black. 42

Figure 3-4 – Overall design of the runtime 46

Figure 3-5 – State diagram showing the transition of **`isAllocated/toCopy2Dev/toCopy2Swap`** flags 52

Figure 3-6 – Preemption cycle 62

Figure 3-7 – Execution time reported with a variable number of short-running jobs on a node with one GPU. The bare CUDA runtime is compared with our runtime 66

Figure 3-8 – Execution time reported with a variable number of short-running jobs on a node with three GPUs. The bare CUDA runtime cannot handle more than eight concurrent jobs. 67

Figure 3-9 – 36 *MM-L* jobs (with conflicting memory requirements) are run on a node with three GPUs. The fraction of CPU code in the workload is varied. We indicate the number of *swap* operations occurred on top of each bar 68

Figure 3-10 – 36 jobs (BS-L and MM-L) are run on a node with three GPUs. The workload composition is varied. We indicate the number of *swap* operations that occurred on top of each bar. 69

Figure 3-11 – Unbalanced node with two Tesla C2050s and one Quadro 2000: effect of load balancing through dynamic binding. The number of *MM-S* jobs migrated to *fast* GPUs is reported on top of each bar. 70

Figure 3-12 – Two-node cluster using TORQUE: effect of GPU sharing and load balancing via inter-node offloading in the presence of short-running jobs and in the absence of conflicting memory requirements. 72

Figure 3-13 – Two-node cluster using TORQUE: effect of GPU sharing and load balancing via inter-node offloading in the presence of long-running jobs and conflicting memory requirements. 73

Figure 3-14 – Preliminary experiments: comparison among five scheduling and sharing schemes for a 4-job workload including *all-to-all* communication primitives. In (5-a), each job consists of four processes, and the *GPU phase duration* parameter of one of these processes is higher than that of the other three by a factor *percentage imbalance*. In (5-b), workload composition $j_1\times[p_1]+j_2\times[p_2]$ indicates that j_1 jobs consist of p_1 processes and j_2 jobs consists of p_2 processes. 79

Figure 3-15 Intra-application imbalance--speedup for broadcast communication pattern 82

Figure 3-16 Intra-application imbalance--speedup for scatter-gather communication pattern 82

Figure 3-17 Intra-application imbalance--speedup for barrier synchronization pattern 82

Figure 3-18 Inter-application imbalance-- speedup for broadcast pattern 83

Figure 3-19 Inter-application imbalance-- speedup for scatter-gather pattern 83

Figure 3-20 - Inter-application imbalance-- speedup for barrier pattern 84

Figure 3-21 Overall execution time in cluster settings 88

Figure 4-1 – Architecture of the cluster-level scheduler 96

Figure 4-2 – Performance of three benchmark applications running with 8 processes on different node configurations (from 1 to 3 nodes). $Node_i[j]$ indicates that j processes are run on $Node_i$. 103

Figure 4-3 – Co-location-based scheduler with relaxed constraints (1, 2 and 3 nodes/job allowed) 113

Figure 4-4 – Throughput for different sets of weight assignments 115

Figure 4-5 – Overall throughput and QoS for the heterogeneous workload summarized in Table 4-5. 117

Figure 5-1 – Overall execution time of MPI-CUDA implementations of the Himeno and Needleman-Wunsch benchmarks under different process-to-node assignments. In all cases, eight processes are run on two heterogeneous nodes (Host-I and Host-II) 121

Figure 5-2 – IVM’s Execution and Memory Models 125

Figure 5-3 – System design and GPU support 128

Figure 5-4 – Graphical representation of load balancing schemes 138

Figure 5-5 – Speedup and load distribution in case of DS-LB 144

Figure 5-6 – Execution time of HIMENO with OM-LB 148

LIST OF TABLES

Table Page

Table 2-1 – Examples of MPI communication routines 17

Table 2-2 – Examples of SHMEM communication routines 20

Table 3-1 – For each routine issued, actions performed by the node-level scheduler and possible errors returned. A black in the third column indicates any error generated by the CUDA runtime (i.e. result ≠ cudaSuccess). VM_entry = Virtual Memory Entry. 54

Table 3-2 – Benchmark programs 64

Table 3-3 - Description and setting of the parameters of our benchmark generator 75

Table 3-4 – Characteristics of the nodes 78

Table 4-1 – Scheduling API 98

Table 4-2 – Summary of benchmark characteristics 108

Table 4-3 – High-end cluster setup 111

Table 4-4 – Commodity cluster setup 111

Table 4-5 – Sequence of jobs submitted (the number of processes spawned is indicated in square brackets) 116

Table 5-1 – IVM Programming API 132

Table 5-2 –Hardware setup 143

HIERARCHICAL SCHEDULING AND UNIFORM ACCESS PROGRAMMING FRAMEWORKS FOR HETEROGENEOUS CPU-GPU COMPUTING CLUSTERS

Kittisak Sajjapongse

Dr. Michela Becchi, Dissertation Supervisor

ABSTRACT

The advance of the GPU hardware architecture has made GPUs attractive devices for general-purpose computing. Modern GPUs are equipped with an increasing number of cores, a flexible memory hierarchy, and a large memory capacity. While the computational power of modern GPU devices has allowed their introduction in high-performance computing (HPC) clusters and the efficient processing of ever larger workloads, existing software components for HPC clusters still offer basic support for hardware heterogeneity and often cause performance limitations in the presence of GPU devices. In particular, two kinds of limitations are associated with these software components: *runtime support* and *programmability*. We found that these limitations are due to the fact that existing software frameworks for heterogeneous clusters treat GPUs as dedicated coprocessor devices.

In this dissertation, we propose two software frameworks for addressing the performance and hardware underutilization issues found in heterogeneous CPU-GPU clusters as well as increasing their programmability. Our frameworks provide a uniform view of compute resources and treat CPUs and GPUs equally as first-class resources, allowing efficient management of heterogeneous compute resources. First, we propose a hierarchical scheduling framework consisting of a node-level runtime and a cluster-level scheduler that provides abstraction of heterogeneous compute resources at different granularities. This hierarchical framework targets existing applications and does not require their modification. In the node-level runtime, we identify and design mechanisms, such as *virtual GPUs*, *GPU virtual memory*, *dynamic load*

balancing and *pre-emption,* which are necessary to support efficient sharing and load balancing schemes for GPUs within a compute node. In the cluster-level scheduler, we introduce mechanisms to abstract compute nodes and perform load balancing in concert with the node-level runtime. Our hierarchical scheduling framework allows supporting different load balancing policies and does not require additional inputs (such as profiling information) from users. Second, we propose a programming framework based on a novel memory and execution model. Our memory model hides disjoint addressing spaces (corresponding to different CPUs, GPUs and compute nodes) and provides a view of a single virtual memory space that can be accessed by all compute resources in a heterogeneous cluster. Our execution model provides uniform access to compute resources and allows our framework to treat all CPUs and GPUs equally and to access data in the virtual memory space.

CHAPTER 1

INTRODUCTION

1.1 Introduction

For decades, high-performance computing (HPC) has played a major role in pushing research toward answers and new discoveries. Challenging problems in science and engineering have traditionally been solved by simulation and analysis on high-performance computing platforms. Nowadays, the scale of the data used to solve many scientific problems has become significantly larger. As a consequence, the solution of these problems in a reasonable amount of time necessitates even more the aid of high-performance computing clusters. In the last decade, Graphic Processing Units (GPUs) have also been widely used to accelerate scientific and engineering computations. Thanks to their high computational power and low price-per-performance, these devices have gained popularity in the research community. The importance of GPUs is also emphasized by their adoption in the world's largest supercomputers [3]. However, GPUs have been originally intended as dedicated accelerators for simple applications and, therefore, their software stacks was not designed for use in shared computing environments, such as HPC clusters. The use of these software stacks in shared cluster environment causes resource underutilization and performance penalty/degradation.

In this dissertation, we analyze the inefficiencies due to the limitations of existing GPU software stacks and propose hierarchical scheduling frameworks to maximize both overall throughput of HPC clusters and performance of applications. Our frameworks enable dynamic load balancing by performing coarse- and fine-grained scheduling of loads to resources. In the first part of our work, we study the limitations of current software support for GPUs and propose a hierarchical scheduling framework to dynamically schedule applications to compute resources without requiring modifications to the applications. In the second part of our work, we address the inefficiencies of existing programming models for CPU-GPU clusters and propose a programming model that can be supported by a runtime framework, including the resource sharing and scheduling mechanisms that we have identified. Our ultimate goal is to enable applications to exploit fully the hardware capabilities as well as increase programmability of heterogeneous clusters consisting of CPUs and GPUs.

1.2 Motivations

Over the last decade, many-core devices (such as GPUs) have been widely used to accelerate a variety of applications [4-6]. In its online catalog [7], Nvidia lists about 200 GPU-accelerated applications from different domains, including computational chemistry, biology, physics, numerical analytics, weather prediction, computational finance and data mining. The popularity of GPUs is emphasized by their increased use in HPC clusters: as of November 2013, four of the top 10 fastest supercomputers in the world [3] (Tianhe-2, Titan, Piz-Daint and Stampede) are equipped with Nvidia GPUs.

Existing resource management framework for CPU-GPU clusters, such as TORQUE and SLURM, rely on traditional software stacks like CUDA and OpenCL.

These software stacks, however, were designed viewing GPUs as dedicated accelerators, and lack good support for their use in multitenant environments. The use of existing frameworks in heterogeneous clusters can lead to performance and hardware underutilization issues. In the following sections, we discuss these issues in more detail.

1.2.1 Software support for GPUs

Current software stacks for GPU applications, such as CUDA, view GPUs as dedicated accelerators. Applications relying on these software stacks explicitly acquire GPUs, manage them and perform computations on them. The accelerator model views CPUs as *hosts* and GPUs as *devices* where the host handles general tasks including managing GPUs and offloads only major computations to GPU devices. Most GPU applications, therefore, alternate between CPU and GPU phases. The common structure of GPU applications is the following. First, data is initialized and pre-processed on CPU. The CPU then offloads the computation to GPU after the initialization is complete. After the GPU completes the computation, the CPU gathers the results from the GPU and possibly performs post-processing on the results. Since the CUDA runtime also requires applications to explicitly select the GPUs for their computation, the CUDA runtime binds applications to the selected GPUs. The binding between an application and a GPU is static that implies that the application data will only reside on the selected GPU and the computation will be offloaded only to that GPU for the entire lifetime of the application.

This computational model has several issues. First, it is obvious that the alternation between CPU and GPU phases leads to hardware underutilization because either CPU or GPU will be utilized in each phase causing the other type of resource to be idle. A remedy to this issue is to utilize both CPU and GPU for the computation.

However, this utilization comes at the cost of a complex implementation since it requires judicious load balancing between CPUs and GPUs. Second, the explicit GPU procurement can be problematic in the presence of concurrent applications. In a shared environment, each application is not aware of the existence of other applications, possibly leading to resource contention. For example, it is possible that applications may collectively select the same GPU and thus overload it while leaving other GPUs idle. In a shared environment, a resource manager is required to distribute applications to GPUs in a harmonious manner in order to avoid interference between applications. Explicit procurement of GPUs may not be suitable to shared environments since it limits the ability to dynamically distribute the application load to the available GPU devices. Finally, static binding of applications to GPUs may limit the ability to manage GPUs and to perform load balancing at run-time [1]. In one scenario, as an example, a GPU application may be mapped to a GPU with low performance capability due to the temporary unavailability of GPUs with high performance capability. To maximize performance, this application should be moved to a GPU with higher capability as soon as it becomes available. However, static binding prevents dynamic migration of applications among the available GPUs. In another scenario, an application may bind to a specific GPU and store its data on that GPU. However, if the CPU phase of the application is significantly long, the application is binding to the GPU without utilizing the GPU well and also limits the possibility for other applications to use the GPU. This kind of GPU underutilization is particularly problematic for applications with long CPU phases or distributed applications that perform communications frequently. This problem may be avoided by preempting applications [2].

1.2.2 Resource Managers

Traditional frameworks were designed to manage homogeneous clusters and can be inefficient for heterogeneous clusters in term of utilization and performance. Most widely adopted resource managers and cluster schedulers, such as TORQUE [8] and SLURM [9], were designed to target homogeneous clusters. With the increasing popularity of GPUs, they have recently been extended to handle accelerators such as GPUs. Since they mostly rely on software stacks intended for dedicated setting to provide GPU supports, these systems lead to suboptimal throughputs and underutilization.

State-of-the-art resource managers accept resource requests (i.e. CPUs/GPUs, RAM, disk space, network traffics etc.) from users and distribute the load at a coarse-grain by mapping applications to nodes according to the requests and current system load. Once an application is assigned to a node, the assignment is static; and the application will execute on the same set of nodes for its entire lifetime. This design leads to two issues that can cause inefficiencies in heterogeneous clusters. First, the user will need to have some knowledge on the configuration of the cluster to make resource requests to the resource managers [10]. For example, users will need to identify the nodes with GPUs and determine the number of GPUs available on each of the nodes in order to make requests. Second, the coarse-grained mapping of applications to nodes and the fine-grained mapping of applications processes to GPUs are static. Load balancing between heterogeneous nodes can be difficult with current resource managers since they do not account for heterogeneity. Under heterogeneous clusters, the amount of load each node will handle is determined by the performance capability of the node prior to application-to-node mapping/scheduling [10]. This maneuver can be cumbersome to users since they

have to estimate good hardware configurations for their applications in order to make requests to the resource managers. In addition, load balancing is not possible since these resource managers do not directly manage GPUs inside nodes.

In addition, state-of-the-art resource managers offer very limited support for resource sharing, and this support is generally limited to CPU resources. Coarse-grained space sharing is extensively studied with this type of resource managers [11-13]. Backfilling is one of the mechanisms proposed to reduce application latency and improve quality-of-service (QOS) for users. Although some coarse-grained space-sharing policies can be defined within the resource managers, hardware utilization of both CPUs and GPUs can be poor. To understand how space-sharing can be inefficient in current systems, consider an application requesting resources leaving free resources in the cluster that cannot be used to accommodate subsequent requests. For example, suppose we have a cluster consisting of seven nodes with three GPUs on each of the nodes. Three applications request three nodes with eight GPUs each. In this case, the resource managers will allow only two applications to proceed and leave five GPUs and one node idle. This situation can be inefficient since the third application submitted needs to wait for the first two applications to complete before being able to proceed. This causes the latency of the application to increase and the cluster to be underutilized. It is worth noting that existing resource managers do not allow fine-grained space-sharing or time-sharing of a single GPU across applications. Currently, only little study has been conducted on time-sharing resources in clusters. Time-sharing can be a mechanism to improve hardware utilization and, as a result, the quality of service of applications. In the previous example, a resource manager with time-sharing enabled will allow the third application to

execute concurrently with the other two. For example, the third application can be assigned to nodes such that six of its tasks time-share a set of GPUs on these nodes. The application can even be assigned entirely to a single node (the one remaining after mapping the first two applications) to limit application interference. Time-sharing can, however, produce negative effects, such as interference between applications or unexpected performance degradation. A careful study of time-sharing resources in heterogeneous clusters must be conducted to maximize performance, utilization, and quality-of-service.

1.2.3 Programmability

Heterogeneous clusters can include CPUs and GPUs with different compute capabilities. Not only does this complicate resource scheduling, but it also requires significant development effort since complex architectural details such as disjoint memory spaces are exposed to programmers. For example, it is necessary for programmers to explicitly manage copies of data on separate nodes and GPUs. In addition, GPUs are viewed as discrete accelerators that are explicitly managed by CPUs. As a consequence, CPUs and GPUs are not accessed in a uniform way. These factors complicate programming of heterogeneous clusters with GPUs.

The development of distributed GPU applications requires a good knowledge of parallel programming at different levels, i.e. inter-node, intra-node, and intra-GPU levels. At the inter-node level, programming models such as MPI and SHMEM are used to support communications between nodes. Programmers need to determine the best communication pattern to achieve the lowest communication latency given a particular system configuration. However, these programming models do not handle alone GPUs.

At the intra-node level, programmers need to mange consistency between copies of data on CPUs and GPUs and judiciously orchestrate the computation. Since each CPU and GPU device has its own physical memory, the clusters are also characterized by a distributed memory system. Lastly, the differences in architectural details among different types of CPUs and GPUs must be carefully taken into account to achieve peak performance. While programming techniques and optimizations for compiler and runtime support for homogeneous CPU systems/nodes are mature, application development on GPUs does still require significant programmer effort. Not only do programmers need to write massively parallel codes, but they need also to deal with complex memory system [14].

Hybrid programming models can be used for developing distributed GPU applications. For example, MPI-CUDA or SHMEM-CUDA can be employed to distribute load between nodes and allow applications to use GPUs on each node. In the presence of heterogeneity, judicious load placement is necessary to avoid performance and underutilization issues. Load balancing can be complex using these hybrid programming models; programmers must implement load balancing schemes at different granularities, i.e. inter-node and intra-node. In our previous work, we have developed a hierarchical load balancing for massive sequence alignments on heterogeneous clusters with GPUs [15]. At the inter-node level, we have implemented a load balancing scheme based on the producer-consumer model and used MPI for inter-node communication. At the intra-node level, we have used multithreading and CUDA streams [14] to manage computations on multiple CPUs and GPUs with different performance capabilities. However, the resulting application complexity is not trivial.

In summary, existing CPU-GPU clusters lead to issues in terms of programmability and runtime support. This dissertation addresses both problems.

1.3 Contributions

In this dissertation, we make the following contributions:

1.3.1 Runtime Supports for Distributed GPU applications

We propose a node-level runtime component that provides runtime support for GPU applications. We identify virtual memory for GPUs and GPU virtualization as essential mechanisms to abstract physical GPUs from users, provide dynamic load balancing, and improve application performance and system utilization. Our node-level runtime provides mechanisms such as *dynamic binding* of applications and *GPU preemption* to enable dynamic load balancing across concurrent applications. These mechanisms efficiently manage applications even in the presence of CPUs and GPUs with different compute capabilities. In addition, our node-level runtime supports dynamic load balancing in case of GPU addition and removal, resilience to hardware failure, and checkpoint-restart capabilities. Our node-level runtime supports different scheduling policies.

1.3.2 Hierarchical Resource Management

We propose a cluster-level scheduler for heterogeneous systems and integrate it with our proposed node-level runtime to form a hierarchical scheduling framework. This hierarchical framework allows better utilization of heterogeneous CPU-GPU clusters by judiciously distributing the load at different levels. At the coarse-grain, applications are distributed to different nodes. At the fine-grained, the node-level runtime dynamically distributes tasks to GPUs. Our cluster-level scheduler is configurable and allows administrators to easily define custom scheduling policies to meet specific requirements

related to cluster setup, applications characteristics, and the users' needs. This hierarchical scheduling framework operates on unmodified CPU-GPU binaries and transparently schedules application and dynamically performs load balancing without the user intervention. We show the effectiveness of the proposed scheduling framework on a variety of workloads.

1.3.3 Programming Framework for Uniform Resource Access

We propose a novel programming framework based on the shared memory model to improve programmability and enable efficient scheduling of application in heterogeneous clusters with GPUs. Our programming framework reduces complexity in development of distributed applications by providing uniform view of the memory space and resources with a heterogeneous cluster. We propose a programming framework called Inter-node Virtual Memory (IVM) that provides the programmer with a uniform view of compute resources and memory spaces within a CPU-GPU cluster, and a mechanism to easily incorporate load balancing within the application. We compare the use of MPI and IVM on four distributed CUDA applications. While the main goal of IVM is programmer productivity, the use of the load balancing mechanisms offered by IVM can also lead to performance gains.

1.4 Organization

The rest of this dissertation is organized as follows. Chapter 2 provides a background on programming models and runtime systems for heterogeneous clusters. Chapter 3 describes our node-level runtime for GPU applications. Chapter 4 describes the cluster-level scheduler that we have designed and its integration with our runtime component to form a hierarchical scheduling framework. In Chapter 5, we then describe a novel

programming framework to enable unified programming and facilitate the coding of applications embedding dynamic load balancing in heterogeneous clusters with GPUs. Finally, we provide future directions for the current research in chapter 6.

CHAPTER 2

BACKGROUND & RELATED WORK

2.1 Background

2.1.1 Graphic Processing Units (GPUs)

Graphics Processing Units (GPUs) have been traditionally used to accelerate image/video processing applications and 3D-games. These platforms have a highly parallel architecture consisting of a large number of processing cores since they were originally designed to handle a large number of pixels concurrently. Current software stacks for GPUs also enable more general purpose computations such as scientific and engineering applications to utilize these devices.

GPUs are generally used as accelerator devices (or coprocessors). The host CPU handles general parts of the application, such as book-keeping, pre-/post-processing, and I/O interface, and offloads compute intensive tasks to the GPUs. Because GPUs have their own memory space, memory transfers between the host CPU and the GPUs are

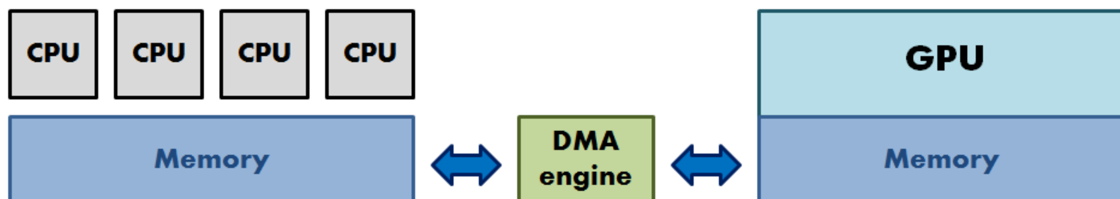


Figure 2-1 – Memory model between CPUs and GPU

necessary to enable communication between them. Figure 2-1 shows the memory configuration of a hybrid CPU-GPU node. Both CPUs and GPUs have their own disjoint physical memory spaces and they can communicate and exchange data by copying memory content back and forth. Memory transfers are usually performed by a Direct Memory Access (DMA) engine to allow fast transfer and low CPU overhead. The most common scenario in a CPU-GPU interaction is the following. First, the host CPU initializes the datasets and data structures its own memory space (CPU memory). Once this initialization is complete, these data are transferred to the memory space belonging to the GPUs (GPU memory). The CPU can then initiate the computation on the GPUs. The results may finally be transferred from the GPU to the CPU memory for further processing or report generation.

The Compute Unified Device Architecture (CUDA) [16] and Open Computing Language (OpenCL) [17] are well known frameworks for creating GPU applications.

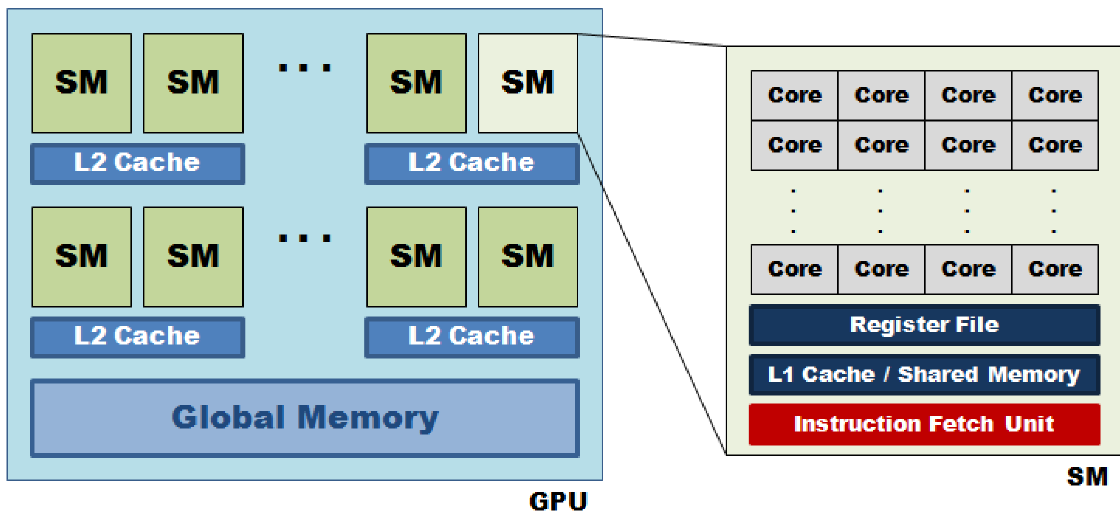


Figure 2-2 – GPU architecture under CUDA framework

These frameworks define the computation and memory models for GPUs. They consist of a runtime library and a compiler. It is worth noting that, while CUDA targets only GPUs, OpenCL also targets other types of accelerators such as FPGA. Since, in this dissertation, we will consider only GPUs, we will mostly refer to CUDA and use terminology defined by this framework. Figure 2-2 shows the high-level architectural details of GPUs under CUDA. A GPU consists of a collection of Streaming Multi-processors (SMs), which share the same global memory space. The global memory space was earlier referred to GPU memory wherein data from the CPU memory can be stored. A group of SMs share an L2-cache which is fully controlled by the hardware. A SM consists of a collection of scalar cores or SIMD-lanes which share a register file, an L1-cache, and one or more instruction fetch units. Since the cores share the same instruction fetch units, they process instruction in lock-step. However, while all cores execute the same instruction, operands for the instruction at a time step can be different which each of them operates on different data. The register file and L1-cache are the storage areas with the lowest access latency. The L1-cache can partially be programmatically configured as *shared memory*. Shared memory allows software to manage the content of the memory whereas L1-cache is managed by the hardware. Programmers can, therefore, decide what kind of data to be put in the shared memory.

2.1.2 Programming Models for Heterogeneous Clusters

In a heterogeneous cluster, multiple compute nodes cooperate to process distributed applications submitted by users. A compute cluster usually consists of a *head node*, which acts as a frontend interface for submitting applications. A resource manager or scheduler usually run on the head node and distributes tasks to the other compute nodes.

Tasks within a distributed application usually communicate with each other. Therefore, tasks on different nodes will cooperate and exchange data. Task execution can be broken into two phases: *computation phase* and *communication phase*. Tasks perform computation independently on CPU or GPU in the computation phase; they communicate and exchange data in the communication phase. Tasks will alternate between the two phases for the entire lifetime of the application. Compute nodes in clusters are commonly connected through either Ethernet networks [18] or Infiniband fabrics [19]. The networks enable compute nodes to exchange data based on relationships and dependencies between tasks.

Ethernet is the most common network technology used in both commodity and high-end clusters. Ethernet offers a reliable stream-based connection for a pair of nodes. All modern operating systems include *socket API* as a programming interface to implement point-to-point communication between nodes through send (*send*) and receive (*recv*) methods. The communication method based on send and recv is called *two-sided communication* since it requires the participation of the sending and receiving side. The correct operation of two-sided communication is guaranteed when information sent from the sending side is acknowledged by the receiving side. Complex communication routines that involve more than two nodes can also be implemented using the socket API.

Infiniband fabric is mostly deployed in high-performance computing clusters. The goal of this network technology is to provide messaging service between nodes without the intervention of the operating system in order to achieve low-latency communication and reduce CPU overhead. Under Unix-based operating systems such as Linux, this allows applications to interface with the network adaptor directly from user-space and

bypass the kernel-space. In addition to the two-sided communication, Infiniband also offers a communication method that allows nodes to directly read from and write to memory locations of remote nodes. This method is technically known as Remote Memory Direct Access (RDMA), and it is used to implement remote read/write (*rem-rd*, *rem-wr*) operations. This communication method is also called *one-sided communication* because only one party will need to complete the information exchange while the other end is free to perform other tasks. This communication method allows communication and computation phases of tasks to execute concurrently in a more efficient manner. Infiniband fabrics also allow nodes to perform atomic operations (e.g., atomic increments and compare-and-swap) on a memory location of a node. Programmers can implement their distributed applications on Infiniband fabric through the Verbs API.

Implementing distributed applications directly on Socket or Verbs is an extremely difficult and challenging task requiring a large development and debugging effort. Most distributed applications in science and engineering are written using programming frameworks based on the Socket or the Verbs API. These frameworks facilitate communication between tasks and allow programmers to easily implement work-distribution methods. The most widely used programming models for distributed applications include the Message-Passing-Interface (MPI) and Partitioned Global Address Space (PGAS). These programming models employ the two communication methods mentioned above. In particular, MPI employs two-sided communication while PGAS mainly employs one-sided communication. These programming models are described in more detail in the next sections.

2.1.3 Message-Passing Interface (MPI)

Message-Passing-Interface (MPI) defines a standard for creating distributed applications based on the message-passing paradigm. Tasks in a distributed application communicate by exchanging messages. In many MPI implementations, such as OpenMPI [20], MPICH2 [21], and MVAPICH2 [22], tasks are distributed to nodes in the form of processes managed by the local operating system. These processes can run on the same or different nodes, and each has an identification number called *rank*. These processes do

MPI routines	Descriptions	Type
Point-to-point communication		
MPI_Send()	Sends a message to a specific rank.	Blocking
MPI_Recv()	Receives a message from a specific rank.	Blocking
MPI_Isend()	Sends a message to a specific rank in an unblocking manner.	Non-blocking
MPI_Irecv()	Receives a message from a specific rank in an unblocking manner	Non-blocking
MPI_Testall()	Tests for the completion of all previously initiated communications. This is useful especially for unblocking routines.	Non-blocking
MPI_Waitall()	Waits for all given communication to complete. This is useful especially for unblocking routines.	Blocking
Collective communication		
MPI_Gather()	Gathers together values from a group of processes.	Blocking
MPI_Allgather()	Gathers data from all tasks and distributes it to all.	Blocking
MPI_Scatter()	Sends data from one task to all other tasks in a group.	Blocking
MPI_Reduce()	Reduces values on all processes to a single value.	Blocking
MPI_Allreduce()	Combines values from all processes and distributes the result back to all processes.	Blocking
MPI_Alltoall()	Sends data from all to all processes.	Blocking
MPI_Barrier()	Blocks until all processes have reached this routine.	Blocking
MPI_Bcast()	Broadcasts a message from the process with rank “root” to all other processes of the group.	Blocking

Table 2-1 – Examples of MPI communication routines

not share memory address spaces or data structures. Each process has its own memory space. Therefore, information exchange only occurs by transferring memory content from one process to another. MPI provides various communication routines implementing point-to-point, one-to-many, many-to-one, or many-to-many communication. The routines involving more than two parties are known as *collective communication* primitives. MPI collective communication primitives implement two-sided communication.

Table 2-1 summarizes the MPI calls for exchanging information among processes. These communication primitives are divided into point-to-point and collective communication routines. These primitives can either be blocking or non-blocking. A blocking routine waits until the communication is complete; a non-blocking routine immediately returns even if the communication is not complete. The non-blocking routines allow a process to post communication requests to the MPI framework while performing other computation or communication tasks in parallel. Programmers can test the completion of non-blocking communication routines by issuing *MPI_Testall()* or *MPI_Waitall()* calls. The blocking nature of some collective communication routines can be performance bottleneck and cause other performance issues, which will be described in more detail throughout this dissertation.

2.1.4 Partitioned Global Address Space (PGAS)

Partitioned Global Address Space (PGAS) is a programming model for distributed applications that provides a shared-memory abstraction to some degree. In PGAS, tasks are distributed as units called *Processing Elements (PEs)*, which execute in parallel either on the same or on different nodes. PGAS provides a memory model similar to the Non-

Uniform Memory Access architecture (NUMA) wherein the entire memory address space consists of multiple memory regions that are local to different PEs. Each PE has a local region that is divided into two sections. The first section of the region is called private section and can only be accessed by that PE. Examples of variables residing in the private section include local variables that appear in functions or loops, non-static variables and variables or memory regions that are dynamically allocated. The other section is called *symmetric region* and accesses to it from other PEs are allowed. During execution, PEs can exchange information using the one-sided communication by writing to and reading from memory locations in symmetric sections. The symmetric section includes static variables, global variables, and symmetric heap. The symmetric heap is the central heap that a PGAS programming framework can use to dynamically allocate variables from a PE and allows other PEs to access the variables. Variables residing in the symmetric heap of each PE are called *symmetric objects* and can be remotely written or read by the other PEs. All PEs have different copies of a symmetric object; these copies will have the same data type and size but may contain different contents. There are several well-known implementations of the PGAS model: Unified-Parallel-C, Co-Array Fortran, and SHMEM. Unified-Parallel-C and Co-Array Fortran use compiler-techniques and provide extensions to standard programming languages (C/Fortran). Such frameworks provide a set of PGAS-enabled compilers, linkers and runtime support for distributed applications. SHMEM is library-based and allows programmers to create distributed applications using generic compilers. SHMEM provides more flexibility and portability by allowing applications to be written in different programming languages. Recently, SHMEM has gained popularity in the high-performance computing community. This framework was

SHMEM Routines	Descriptions
Point-to-point communication	
shmem_putmem()	Copies data from a contiguous local data object to a data object on a specified PE.
shmem_getmem()	Copies data from a data object on a specified PE to a contiguous local data object.
Atomic operations	
shmem_int_add()	Performs an atomic addition operation on a remote data object of a specified PE.
shmem_int_cswap()	Performs an atomic conditional swap on a remote data object of a specified PE.
shmem_int_inc()	Performs an atomic increment operation on a remote data object of a specified PE.
Collective communications	
shmem_barrier()	Registers the arrival of a PE at a barrier and suspends PE execution until other PEs arrive at the barrier and all local and remote memory updates are completed.
shmem_broadcast32/64()	Broadcasts a block of data from one PE to one or more destination PEs.
Shmem_collect32/64()	Concatenates blocks of data from multiple PEs to an array in every PE.

Table 2-2 – Examples of SHMEM communication routines

originally designed to take advantage of the RDMA mechanism offered by Infiniband interconnects. SHMEM implementations are available from various manufacturers such as SGI, Mellanox and OpenSHMEM. OpenSHMEM is an open source specification of SHMEM created in a standardization effort. The OpenSHMEM library implements communication routines as a set of function calls included in the Application-Programming-Interfaces (APIs). These communication routines are implemented in the form of remote writes and reads called *Put* and *Get* methods respectively.

Many frameworks, such as Unified Parallel C (UPC) [23], Co-Array Fortran [24] and SHMEM [25], are based on the PGAS programming model. In this dissertation, we will mainly refer to the SHMEM programming model and OpenSHMEM [26] as a

specific implementation of it. Table 2-2 shows a set of OpenSHMEM communication routines. Similarly to MPI, communication routines are also divided into point-to-point and collective routines; and they are based on the one-sided communication paradigm. In addition, the nature of one-sided communication allows atomic routines on variables residing in the symmetric address space.

2.2 Related Work

2.2.1 Node-level Schedulers and Runtime Support

A number of previous studies have proposed node-level techniques and mechanisms for heterogeneous nodes to improve individual performance of applications or aggregated performance and utilization of CPUs and GPUs in clusters. These proposals can be divided into two categories according to the types of applications considered. The first category includes scheduling and virtualization frameworks for single-process GPU applications, where the second covers more complex, distributed applications.

GViM [27], vCUDA [28], rCUDA [29] and gVirtus [30] are runtime systems to enable GPU applications in virtualized environments. These frameworks provide GPU visibility from within virtual machines by using API remoting to submit computation requests for GPUs from guest OS to host OS. GViM and vCUDA leverage the multiplexing mechanism provided by the CUDA runtime in order to allow GPU sharing among different applications. GViM uses a Working Queue per GPU to evenly distribute the concurrent applications to GPUs. As additional feature, GViM provides a mechanism to minimize the overhead of memory transfers when GPUs are used within virtualized environments. In particular, its authors propose using the *mmap* Unix system call to avoid data copies between the guest OS and the host OS. Whenever possible, they also propose

using page locked memory (along with the cudaMallocHost primitive) in order to avoid the additional data copy between host OS and GPU memory. Gelado et al., [31] and Becchi et al., [32] explored GPU abstraction by providing memory-management frameworks for nodes including CPUs and GPUs. These frameworks abstract the underlying distributed memory spaces of CPUs and GPUs and provide a uniform view of the memory space. This eliminates the need for programmers explicitly to transfer memory content between CPUs and GPUs. As optimization, their frameworks also avoid some unnecessary memory transfers.

GPU sharing can be used to improve hardware utilization. Guevara et al., [33] proposed kernel consolidation as a way to share GPUs. They showed that this mechanism is particularly effective in the presence of kernels with complementary resource requirements (e.g.: compute intensive and memory intensive kernels). The concept of kernel consolidation has been reconsidered and explored in the context of GPU virtualization by Ravi et al., [34].

In term of balancing loads across GPUs, our work differs from GViM [27] and vCUDA [28]. These node-level schedulers statically assign applications to GPUs, which can lead to suboptimal performance--especially in the presence of heterogeneity. Our node-level scheduler provides GPU abstraction and enables dynamic scheduling and effective GPU sharing mechanisms. In addition, we designed our framework to allow administrators to implement different load balancing policies in the scheduler. Our GPU sharing mechanism differs from GViM [27], Guevera et al., [33], and Ravi et al., [34]. These proposals allow applications to share GPU based on the assumption that the aggregated memory requirements of applications sharing the same GPU does not exceed

the memory capacity of the GPU. Our scheduler targets multi-tenant environments even in the presence of conflicting memory requirements among applications. On one hand, the memory module we designed has some similarities with the frameworks described by Gelado et al., [31] and Becchi et al., [32]. On the other hand, it extends these frameworks and focuses on multi-tenant scenarios. Some optimizations such as zero-copy memory transfer proposed by GViM [27] are orthogonal to our work and can also be incorporated in our scheduler.

2.2.2 Cluster-level Schedulers and Resource Sharing

Scheduling and resource sharing for distributed applications have been extensively studied. A number of previous studies proposed co-location strategies, such as Gang-Scheduling [35, 36], and Co-Scheduling [37-45], for distributed applications to share resources under homogeneous environments. Some of these frameworks allow applications to share resources so as to avoid interference [46, 47]. The primary goal of these frameworks is to both maximize CPU utilization and performance of individual applications.

Gang-scheduling is a scheduling scheme that aims to minimize the cost of communication among tasks belonging to the same application. This scheduling scheme reduces communication latency by scheduling all tasks to compute resources at the same time, thus reducing the latency incurred by context-switching. Gang-scheduling allows time-sharing to concurrent applications by preempting all the tasks of the same application and yielding compute resources to another application. Feitelson et al., [35] presented a performance evaluation of gang-scheduling in shared-memory systems. Hori et al., [36] proposed an implementation of gang-scheduling across nodes in a cluster.

Since gang-scheduling makes scheduling decisions across all participating tasks, it assumes that all tasks communicate at the same time slot. This situation does not hold for point-to-point communication. In fact, it may incur unnecessary synchronization overhead in the presence of large number of tasks and has limited scalability.

Several proposals [37, 39, 41, 43, 44] have proposed more scalable and efficient co-scheduling schemes aimed at reducing inter-process communication. In particular, Arpaci-Dusseau et al., [39] proposed communication-based co-scheduling. In their scheme, information such as the response time and the type of communication among the processes in a parallel application is used to guide scheduling and sharing decisions. Choi et al., [43] proposed a co-scheduling scheme that blocks processes involved in point-to-point communication and tries to co-schedule processes that perform collective communication. Sobalvarro et al., [44] describe spin-block communication in which communicating tasks spin for a predefined period and then block if the communication is not completed within such period. However, all the aforementioned studies target only homogeneous environments including only multi-core CPUs, and do not address performance and utilization issues faced in heterogeneous environments. Further, no effort has been made to extend these works to enable scheduling of distributed applications on clusters with GPUs.

Existing scheduling frameworks such as TORQUE [8] and SLURM [9] are the most common cluster resource managers used in supercomputing centers, such as Texas Advanced Computing Center (TACC) [48]. These tools have been recently extended with the GPU support capabilities. Although these tools do not allow time-sharing of resources, they use back-filling to allow space-sharing of resources including GPUs [11-

13] to improve both the quality-of-service of applications and the cluster utilization. Recently, a number of studies have proposed scheduling and load balancing schemes for clusters with GPUs [49-51]. The main goal of these frameworks is to dispatch applications to nodes and GPUs so as to maximize the aggregated throughput of clusters and allow their better utilization. More recently, Ravi et al., [51] have considered time-sharing on clusters with GPUs. Specifically, the proposed scheduling schemes map applications onto different resources (i.e. CPUs/GPUs) based on user-input and profiling information of applications. Irwin et al., [52] and Ravi et al., [50] proposed scheduling schemes based on user's satisfaction and provider's profit for CPU-only clusters and GPU clusters respectively. Li et al., [15] have presented a hierarchical load balancing scheme for a distributed Needleman-Wunsch algorithm under the presence of heterogeneity. Although these proposals have presented scheduling schemes at the cluster level, their ability to schedule concurrent distributed applications in heterogeneous environments is limited for two reasons. First, these scheduling frameworks assume only a single independent task per application. Second, the scheduling schemes cannot be applied across applications. Dependencies and communication among tasks belonging to concurrent applications complicate the scheduling. The effects of communication are not investigated in these studies. In addition, these scheduling schemes are based on the availability of profiling information and input from users. In our work, we consider also multi-process application and study the effect of inter-process synchronization on scheduling. Our work does not assume the availability of user inputs.

2.2.3 Programming Models for Heterogeneous Clusters

Previous work has proposed extensions to existing frameworks to provide programmers with a more intuitive view of the memory space. CUDA Unified Virtual Addressing (UVA) [53] simplifies the view of memory spaces belonging to different GPUs inside a node. However, different GPU memory spaces are still exposed to programmers. CUDA-Aware MPI [20, 22] is an extension to the MPI frameworks that reconciles memory spaces of CPUs and GPUs. Potluri et al., [54] have proposed mechanisms and optimizations to enable efficient GPU-to-GPU memory transfers for the OpenSHMEM framework. Although these proposals have presented memory abstractions to hide the complexity of disjoint memory spaces between CPUs and GPUs, these memory models cannot be easily integrated with efficient load balancing mechanisms and require programmers to be aware of separated memory spaces between nodes.

Programming models such as MPI and PGAS mainly target homogeneous clusters. Although some extensions to support GPUs have been proposed, these programming models only offer basic support for load balancing between CPUs and GPUs. Load balancing can be a non-trivial task as these programming models require programmers explicitly to code and embed load balancing schemes into their applications. Previous work has proposed load balancing approaches based on work-stealing and the producer-consumer model [15, 55, 56]. Recently, the MPI-2 standard [57] has defined Dynamic Process Creation (DPC) as a mechanism to perform dynamic load balancing. However, the usage of DPC in a MPI framework is not straightforward and may require significant programming skills. *Charm++* [58] provides a more natural way to perform load balancing. The common load balancing strategy within Charm++ is

to over-decompose the workload of applications into *Chare* objects [59], which can then be dynamically scheduled onto resources. Because of this over-decomposition, Charm++ may suffer from excessive overhead of scheduling Chare objects. Kurt et al., [60] have proposed a domain-based programming model which dynamically sizes loads that bind to resources. The proposed programming model is derived from the analysis of the recurring patterns of computations found in a set of scientific applications and aims to reduce the scheduling overhead. Since the proposed programming model is based on the observed computation patterns of a set of applications, it covers only a limited range of applications and may not be applicable to other applications, such as dynamic graph construction. It is worth noting that none of these works attempts to address the issues of scheduling and load balancing for concurrent applications in shared environments.

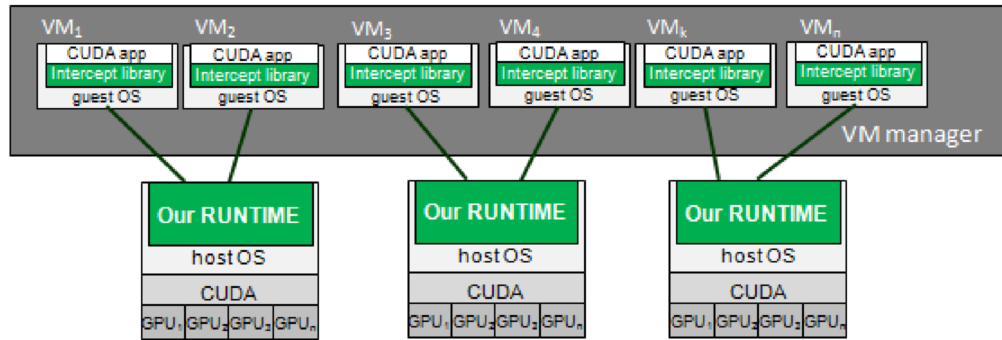
In this dissertation, we propose a novel programming model that provides both a simple memory model and an intuitive programming abstraction to increase the programmability of CPU-GPU clusters. In particular, our goal is to derive a programming model that: (i) provides uniform access to resources, (ii) provides simple and intuitive memory model, and (iii) facilitates embedding load balancing schemes. We will use our proposed programming model as a substrate to derive high-level programming models that hide the complexity of heterogeneous clusters and provide flexibility to manage CPUs and GPUs for concurrent applications.

CHAPTER 3

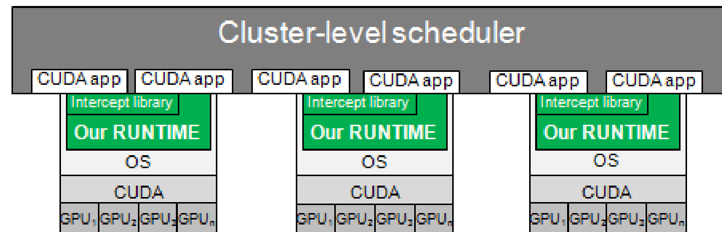
NODE-LEVEL RUNTIME

3.1 Objectives

The overall goal of this work is to provide a runtime component that allows multiple applications to run concurrently on a heterogeneous cluster whose nodes comprise CPUs and GPUs. We foresee the use of our runtime system in two scenarios (Figure 3-1): (i) in combination with VM-based cloud computing services (e.g.: Eucalyptus [61]), and (ii) in combination with HPC cluster resource managers (e.g.: TORQUE [8]). In both cases a cluster level scheduler assigns VMs or jobs to heterogeneous compute nodes. Our runtime component is replicated on each node and schedules library calls originated by applications on the available GPUs so as to optimize the overall performance. Our framework must allow integration with cluster-level schedulers intended for both homogeneous and heterogeneous clusters (the former oblivious of the presence of GPUs).



(a)



(b)

Figure 3-1 – Two deployment scenarios for our runtime: (a) VM-based cloud computing service and (b) HPC cluster resource manager.

Note that heterogeneous clusters that include GPUs require scheduling at two granularities: on one hand, jobs must be mapped onto compute nodes (*coarse-grained scheduling*); on the other, specific library calls must be mapped onto GPUs (*fine-grained scheduling*). Existing cluster-level schedulers perform coarse-grained scheduling; whereas our runtime performs fine-grained scheduling. The two schedulers may interact in two ways. First, the cluster-level scheduler may be completely oblivious of the GPUs installed on each node. In case of overloaded GPUs, the node-level runtime may offload the computation to other nodes. To this end, the runtime system must include a node-to-node communication mechanism enabling inter-node code and data transfer. Alternatively the node-level runtime may expose some information to the cluster-level scheduler (e.g.: number of GPUs, load level, etc.), so as to guide the cluster-level

scheduling decisions. While the first form of interaction may lead to suboptimal scheduling decisions, it allows a straightforward integration with existing cluster resource managers and VM-based cloud computing services targeting homogeneous clusters.

Here, we recall the most significant limitations of previous work as follows. Until recently, GPUs could not be accessed from applications executing within VMs. Several projects –GViM [27], vCUDA [28], rCUDA [29] and gVirtuS [30]-- have addressed this issue for applications using the CUDA Runtime API to access GPUs. The general approach is to use *API remoting* to bridge two different OS spaces: the guest-OS where the applications run and the host- OS where the GPUs reside. In particular, API remoting is implemented by introducing an interposed *front-end* library in the guest-OS space and a *back-end* daemon in the host-OS. The frontend library, which overrides the CUDA Runtime API, intercepts CUDA calls and redirects them to the back-end through a socket interface. In turn, the back-end issues those calls to the CUDA runtime. Note that this mechanism provides GPU *visibility* from within VMs, but does not add any form of *abstraction*. In fact, applications still use CUDA Runtime primitives to direct their calls to specific GPUs residing on the host where the VMs are deployed. Moreover, the bare use of the scheduling mechanisms offered by the CUDA Runtime may not be optimal when multiple or multi-threaded applications are mapped onto a single GPU. In this work, we aim to design a runtime that provides *abstraction* and *sharing* of GPUs while allowing *isolation* of concurrent applications. In addition, the runtime must be *flexible* in terms of *scheduling policies* and allow *dynamic binding* of applications to GPUs. Finally, the runtime must support *dynamic upgrade* and *downgrade* of GPUs and be *resilient to GPU failures*. To overcome these limits, we aim at providing the following mechanisms.

- *Abstraction*--GPUs installed in the cluster need to be abstracted (or hidden) from the user's direct access. GPU programming APIs generally require the application programmer explicitly to select the target GPU (for example, using the CUDA runtime `cudaSetDevice` primitive). This situation gives the application control of the number of GPU devices to use. Our design masks the explicit procurement of GPUs, thus allowing a transparent mapping of applications onto GPUs. As a side effect, applications can be efficiently mapped onto a number of devices different from that for which they have been originally programmed. Note that this abstraction is coherent with the traditional parallel programming model for general purpose processors. When a user writes a multithreaded program, for example, he targets a generic multi-core processor. At runtime, the operating system distributes processing threads onto the available cores.
- *GPU Sharing*–As mentioned above, applications targeting heterogeneous nodes alternate general-purpose CPU code with library calls redirected and executed on GPUs. In the presence of multi-tenancy, assigning each application a dedicated GPU device for the entire lifetime of the application may not be optimal in that it may lead to resource underutilization. GPU sharing is an obvious way to improve resource utilization. However, sharing must be done judiciously; excessive sharing may lead to high overhead and be counterproductive.
- *Isolation* – In the presence of resource sharing, concurrent applications must run in complete isolation from one another. In other words, each application must have the illusion of running on a dedicated device. State-of-the-art runtime support for GPUs

provides partial isolation of different process contexts. In particular, each process is assigned its own process space on the GPU; however, GPU sharing is possible only as long as the cumulative memory requirements of different applications do not exceed the physical capacity of the GPU. Our work aims to handle such memory issues seamlessly, allowing GPU sharing irrespective of the overall memory requirements of the applications. In other words, we want to extend the concept of *virtual memory* to GPUs.

- *Configurable Scheduling*–The quality of a scheduling policy depends on the objective function and assumptions about the workload. A simple first-come-first-served scheduling algorithm can be adequate in the absence of profiling information. A credit-based scheduling algorithm may be more suitable to settings that include fairness in the objective function. Further, a scheduling algorithm that prioritizes short running applications can be preferable if profiling information is available. Yet another scheduling policy may be adopted in the presence of expected quality of service requirements (e.g.,: execution deadlines). Our goal is to provide a runtime system that can easily accommodate different scheduling algorithms.
- *Dynamic Binding*–In existing runtime systems (including the CUDA runtime) the mapping of GPU kernels to GPU devices is static, or programmer-defined. A dynamic application-to-GPU binding may be preferable in several scenarios. First, let us consider the situation of a node having GPU devices with different compute capabilities. Existing work in the context of heterogeneous multi-core systems [62] has shown that performance can be optimized by maximizing the overall processor

utilization while favoring the use of more powerful cores. The application of this concept to nodes equipped with different GPUs suggests that the system throughput can be maximized by dynamically migrating application threads from less to more powerful GPUs as they become idle. Second, dynamic binding can help when GPUs are shared by applications cumulatively exceeding the memory capacity. In fact, dynamically migrating application threads to different devices may minimize waiting times. Finally, resuming application threads on different devices allows load balancing when GPUs are added or removed from the system (*dynamic upgrade and downgrade*) and is beneficial in case of *GPU failures* (by preventing a whole application restart).

- *Checkpoint-Restart Capability*–Along with dynamic binding, our runtime provides a checkpoint-restart mechanism that allows efficiently redirecting an application thread to a different GPU. A checkpoint can be explicitly specified by the user, or automatically triggered by the runtime. For example, the runtime may monitor the execution time of particular library calls (e.g. kernel functions) on a GPU. An automatic checkpoint may be advisable after long-running kernel calls to decrease the restart penalty in case of GPU failures. Note that this kind of checkpoint is inserted dynamically at runtime.

From an application perspective, we target two categories of applications: single-process GPU applications and multi-process GPU applications (distributed applications).

3.2 Reference Architecture

The overall reference architecture is represented in Figure 3-1. The underlying hardware platform consists of a cluster of heterogeneous nodes. Each node has one or more multi-core processors and a number of GPUs. The operating system performs scheduling and resource management on the general-purpose processors. Access to the GPUs is mediated by the CUDA driver and runtime library API. Our runtime performs scheduling and resource management on the available GPUs. Each GPU has a device memory. Among others, the CUDA runtime library contains functions to: (i) target a specific device (`cudaSetDevice`), (ii) allocate and de-allocate device memory (e.g., `cudaMalloc/Free`), (iii) perform data transfers between the general purpose processor and the GPU devices (e.g., `cudaMemcpy`), (iv) transfer code onto the GPUs (the internal functions (`cudaRegisterFunction/FatBinary`), and (v) trigger the execution of user-written kernels (`cudaConfigureCall` and `cudaLaunch`).

In addition, the CUDA runtime offers some CPU multithreading support. For example, CUDA 3.2 associates a CUDA context to each application thread. Several contexts can coexist on the GPU. Each of them has a dedicated virtual address space, contains references to textures, modules and other entities, and is used for error handling. CUDA contexts allow different application threads to time-share the GPU processing cores and space-share the GPU memory. In CUDA 4.0, the use of CUDA contexts is slightly modified to allow data sharing and concurrent kernel execution across threads belonging to the same application. As mentioned in Section 1, with both versions of the CUDA runtime, the number of parallel CUDA contexts that can be supported at runtime is limited by the device memory capacity.

As shown in Figure 3-1, our runtime component interacts with a cluster-level scheduler that operates at the node level and must be installed on all the nodes of the cluster. The cluster-level scheduler maps jobs onto compute nodes. During execution, the GPU library calls issued by applications are intercepted by our frontend library and redirected to our runtime daemon on the node where the job has been scheduled. Since our runtime is a stand-alone process, a mechanism for inter-process communication between the job and our runtime demon is needed. In our prototype, we use the socket-based communication framework provided as part of the open-source project gVirtuS [30, 63]. This framework relies on *afunix* sockets in a non-virtualized environment and on proprietary *VM-sockets* in a virtualized one.

In the presence of nodes with different hardware setups, simple cluster-level scheduling policies may lead to queuing on nodes containing a lower number of GPUs (or assigned a higher number of jobs targeting GPU). To tackle this problem, we allow nodes to offload GPU library calls to other nodes in the cluster. For this purpose, we introduce inter-node communication between our runtime components. Note that this mechanism operates at the granularity of GPU library calls and does not affect the portion of the job running on CPU.

3.3 Scheduling Policies

3.3.1 Batch Scheduling

Most of the cluster schedulers in use today, such as the open-source TORQUE and SLURM, perform batch scheduling. Specifically, they allocate to each application as many GPUs as requested and do so for the whole application lifetime. Further, by directly relying on the CUDA runtime, they do not provide GPU sharing mechanisms.

This form of batch scheduling has two major limitations. First, when the number of processes requesting GPUs is larger than the number of available GPU devices, batch schedulers fail to allocate the requested resources and to execute the application. Not only does this deficiency represent a lack of scheduling flexibility, but it also exposes to the user the configuration of the underlying architecture. Second, this scheme leads to resource underutilization in the presence of long CPU execution phases and load-imbalance, thus reducing the throughput of the whole cluster. This situation is illustrated in Figure 3-2(a), which depicts the batch scheduling of three applications (*A*, *B*, and *C*) on four GPUs. Application *A* consists of four processes, each going through two GPU phases; applications *B* and *C* consist of two processes, each including a single GPU phase. All applications present a global synchronization at the end of each GPU phase. In the figure, A_{jk} represents the k_{th} GPU phase of process *j* belonging to application *A*; black blocks represent idle GPU times. As can be seen, processes A_1, A_2 and A_4 must wait for process A_3 to complete the first phase before proceeding to the next one. As a result, GPU_0, GPU_1 and GPU_3 remain idle for part of the execution. Similar considerations apply in the next execution phases. Such a problem is common to batch schedulers that do not allow applications to time-share GPUs.

3.3.2 Controlled n-way GPU Sharing

Our runtime system allows multiple processes to time-share GPUs even in the presence of conflicting memory requirements. Controlled GPU sharing among n processes *(n-way GPU sharing*) is enabled by associating to each physical GPU a predefined number n of virtual GPUs (*vGPUs*). Processes are mapped to virtual GPUs, and vGPUs time-share the underlying physical GPU. Thus, processes mapped onto vGPUs associated to the same physical GPU will time-share the GPU device and execute concurrently. Processes are mapped to vGPUs in a round-robin fashion, prioritizing idle GPU devices. We have shown that this mechanism is particularly beneficial in case of single-process applications that alternate between CPU and GPU execution phases. Specifically, GPU time-sharing allows hiding CPU execution phases behind GPU execution phases of co-scheduled applications.

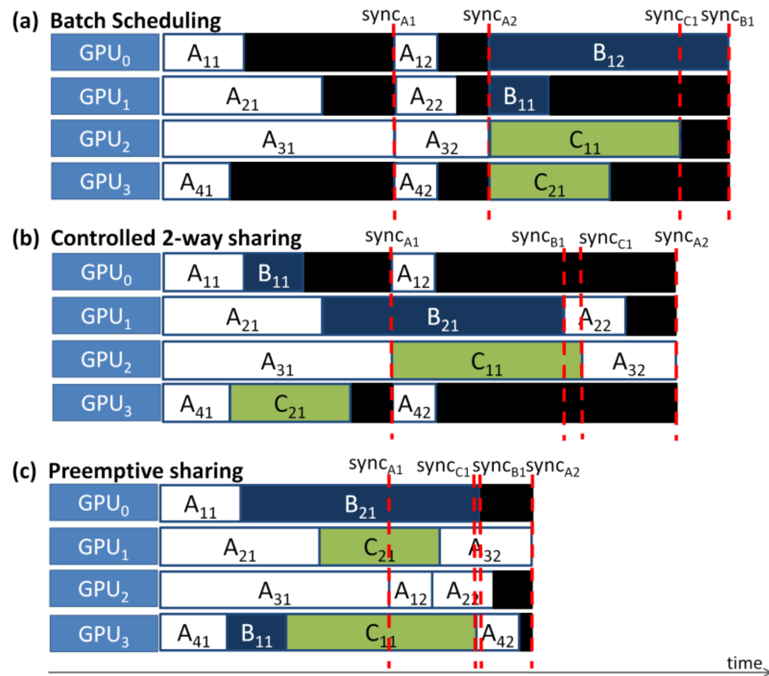


Figure 3-2 – Operation of different scheduling mechanisms in the presence of multi-tasks distributed applications with synchronizations and *intra-application* imbalance. Application A consists of 4 tasks and includes 2 GPU execution phases, with a global synchronization at the end of each. Application B and C consist of two tasks and a single GPU execution phase, also ending with a global synchronization. Synchronizations are represented through (red) dashed vertical lines. A_{jk} represents k^{th} GPU phase of task j belonging to application A. Idle GPU times are represented in black.

The proposed mechanism has an important limitation: the associated scheduling scheme focuses on single-process applications and does not consider synchronization issues related to multi-process applications. In the runtime system described in [1], once scheduled for execution, a process is mapped to a vGPU for its whole life-time, unless a more powerful GPU becomes available or a memory swap request occurs. As a consequence, in order for a multi-process application to proceed with its execution, its processes must all be scheduled onto vGPUs at the same time. This procedure may cause performance problems in highly utilized systems. Although the framework is capable of

migrating processes to different GPUs, such migration occurs only in the presence of conflicting memory requirements or when some GPUs become idle but does not take into account performance considerations related to multi-process applications.

In Figure 3-2(b) we show how the runtime system proposed in [1] would schedule the applications A, B and C described above, assuming two-way sharing (that is, two vGPUs per physical GPU). Note that the processes are mapped to vGPUs in a round-robin fashion, prioritizing idle GPUs. Assuming that the processes are initially queued as *[A1, A2, A3, A4, B1, B2, C1, C2]*, processes *A1* and *B1* will share $GPU_{0;}$ processes *A2* and *B2* will share $GPU_{1;}$ processes *A3* and *C1* will share *GPU2,* and processes *A4* and *C2* will share GPU_3. As can be seen, sharing allows hiding idle time of one process behind the GPU phase of a co-scheduled process. For example, *B1*, *B2* and *C2* can, in this case, start executing while *A1*, *A2* and *A4* wait for *A3* to reach the first synchronization point $sync_{A1}$. However, some GPU underutilization still takes place, especially when processes belonging to the same application exhibit significant imbalance in their execution time. The interested reader can verify that a better schedule would have resulted from co-locating processes *A2* and *C1*, processes *A3* and *C2* and processes *A4* and *B2*. However, the proposed runtime does not have the capability of automatically making such a scheduling decision.

We conclude with two observations. First, despite its limitations, controlled GPU sharing has allowed a better schedule than batch scheduling. Second, one could think of increasing the scheduling flexibility by allowing more processes to share the same GPU (by increasing the number of vGPUs). However, as described in [1], this is not always the optimal choice, since increasing the number of processes mapped to the same physical

GPU increases the probability of memory conflicts (thus causing swapping overheads). In addition, scheduling inefficiencies would arise in any event when increasing the number of processes per application.

3.3.3 Preemptive Sharing

We introduce the concept of *preemptive GPU sharing* as a mechanism to further improve the performance by reducing the idle time of the GPU devices. The basic idea is the following: processes that underutilize the GPU should be *preempted* from use to allow other processes to execute. Processes waiting at a synchronization point are good candidates from preemption. Once the synchronization has been performed, preempted processes can be rescheduled for execution on the same or on a different GPU.

In Figure 3-2(c) we show how preemptive GPU sharing would schedule the applications *A*, *B* and *C* discussed above on four GPUs. This time, we assume that no *n*-way GPU sharing takes place. Pending processes are scheduled for execution as soon as a GPU becomes idle. As can be seen in this case, the scheduler preempts processes A_1, A_2 and A_4 while waiting for A_3 to reach the synchronization point $sync_{A1}$ and lets B_2, C_2 and B_1 execution on GPU_0, GPU_1 and GPU_3, respectively. When the short-running B_1 completes the execution of its GPU phase B_{11}, it is also preempted, and GPU_3 is assigned to C_1. The idle times of all GPUs are minimized. From this example, we can see that preemption allows effective hiding of the idle time of one application behind the execution time of other applications. This scheme provides performance benefits not only over batch scheduling, but also over controlled *n*-way GPU sharing.

3.4 Use Cases for Preemption

At least three scenarios exist that make preemptive GPU sharing preferable to both batch scheduling and controlled *n*-way sharing (from now on, we will omit "controlled" for simplicity).

All these scenarios use cases that involve multi-processes (or multi-threaded) applications with synchronization.

SCENARIO 1: When processes belonging to the same application exhibit significant imbalance in their execution time and GPU utilization: this is a form of *intra-application imbalance*.

SCENARIO 2: When the workload composition is such as to prevent all processes belonging to the same application to be scheduled on the available GPUs at the same time: this is a form of *inter-application imbalance*.

SCENARIO 3: When an application has more processes than physical GPUs (in case of batch scheduling) or than virtual GPUs (in case of *n*-way sharing).

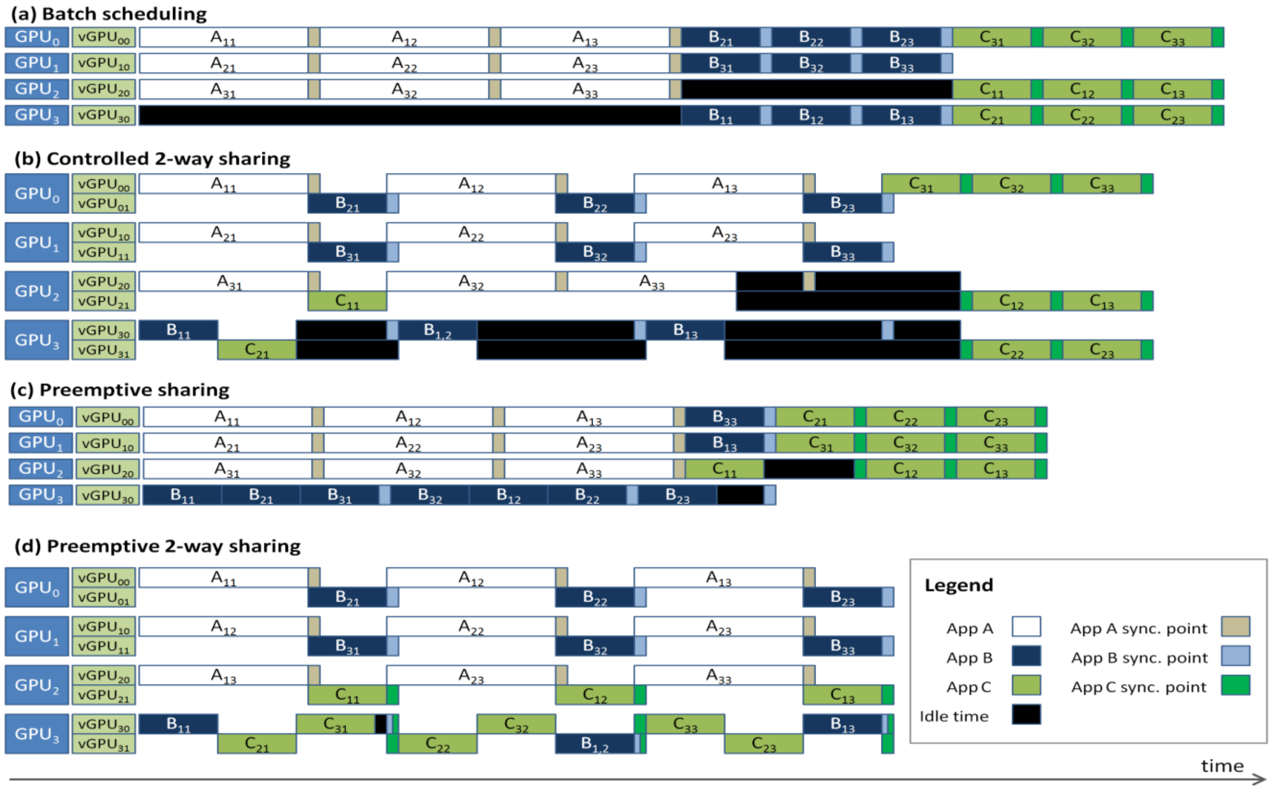


Figure 3-3 – Operation of different scheduling mechanisms in the presence of multi-tasks applications with synchronization and *inter-application imbalance*. Application *A*, *B*, and *C* consist of 3 tasks and include 3 GPU execution phases, with a global synchronization at the end of each. A_{jk} represents the kth GPU execution phase of task *j* belonging to application *A*. Idle GPU times are represented in black.

Scenario 1 (intra-application imbalance) is exemplified in Figure 3-2 and has been already described. Scenario 2 (inter-application imbalance) is represented in Figure 3-3. In this example, we assume again to have three applications *A*, *B* and *C* and four GPUs (GPU_0-GPU_3). The applications consist of three processes, each executing three GPU phases; again, there is a global synchronization at the end of each GPU phase. This time, however, each application is internally balanced, i.e, processes belonging to the same application have the same execution time in each execution phase. We assume that, at

each synchronization point, some CPU code executes. The inter-application imbalance is due to the fact that each application has three processes but the system has four GPUs.

As can be seen in Figure 3-3(a), inter-application imbalance causes GPU underutilization when batch scheduling is performed. In fact, while *A* runs on GPU_0-GPU_2, GPU_3 stays idle. This is because both *B* and *C* require three GPUs, and are, therefore, queued until such GPU resources become available. This situation occurs again later when scheduling application *C* (GPU_2 stays idle until *B* completes execution and frees two more GPUs).

A better schedule is achieved by two-way sharing, as shown in Figure 3-3(b). For illustration, we also show the virtual GPUs associated to each physical device. In this case, some waiting times are hidden by allowing two processes to share a GPU (for example, GPU_0 is initially shared by A_1 and B_2, GPU_1 is initially shared by A_2 and B_3, and so on.). Further, GPU sharing allows hiding the CPU execution of one process behind the GPU execution of a co-located process. For example, B_2 can run on GPU_0 while A_1 performs CPU computation (which includes synchronization code), and vice-versa. However, the presence of synchronization within the three applications still leads to GPU idle times, which ultimately affects the performance.

An even better schedule is achieved by using preemptive GPU sharing, as shown in Figure 3-3(c). In this case, the GPU idle time is minimized by preempting processes that are inactive waiting at synchronization points and by remapping them to GPUs after the synchronization has been performed. Further, an optimal schedule is achieved by combining two and one-way sharing and preemption, as shown in Figure 3-3(d). This solution benefits from the advantages of both schemes: on one hand, CPU execution

phases of one process (including data communication and synchronization) are hidden behind the GPU execution phases of a co-located process; and on the other hand, the use of preemption allows minimizing the GPU idle times. We observe that further increasing the level of GPU sharing (by associating to each physical device a larger number of vGPUs) would not bring additional benefits. In fact, the increased probability of conflicting memory requirements among co-located processes and the added overhead would have detrimental effects on the performance.

Scenario 3 can be explained as follows. As detailed above, batch scheduling allocates to each application as many GPUs as requested by its processes and does not allow GPU sharing. Thus, if one application consists of more processes than the physical GPUs available in the system, such application will not be serviced. Controlled n-way sharing reduces this problem by allowing multiple processes to share a physical GPU and by reassigning a GPU as soon as it becomes idle. However, once scheduled, a process will be mapped onto a GPU for its whole execution time. This works well in the absence of synchronization within the applications. However, let us assume to perform n-way sharing of k GPU devices, and to have an application that presents global synchronizations and p processes, with $p > nk$. In this case, the execution will reach a deadlock situation such that processes 1 to nk will be idle waiting on the global synchronization, while processes $nk+1$ to p will be waiting for a free GPU. GPU preemption allows avoiding this problem.

3.5 System Design

In this section, we describe the design of our proposed runtime. Our prototype implementation targets NVIDIA GPUs programmed through the CUDA runtime API.

3.5.1 Overall Design

The overall design of our runtime is illustrated in Figure 3-4. The basic components are: *connection manager*, *dispatcher, virtual-GPUs* (vGPUs), and *memory manager*. As mentioned before, when applications execute on the CPU, library calls directed to the CUDA runtime are intercepted by a frontend library and redirected to our runtime. We say that each application establishes a connection with the runtime and uses the connection to issue a sequence of CUDA calls and receive their return code. Multiple applications establish concurrent connections. The connection manager accepts and enqueues incoming connections. The dispatcher dequeues pending connections and schedules their calls on the available GPUs. If the devices on the node are overloaded, the dispatcher may offload some connections to other nodes using an inter-node communication mechanism. To allow controlled GPU sharing, each GPU has an associated set of virtual-GPUs. The dispatcher schedules applications onto GPUs by *binding* their connections to the corresponding virtual-GPUs. Applications bound to virtual-GPU $vGPU_{ik}$ share GPU_i. Finally, the memory manager provides a virtual memory abstraction to applications. Dispatcher and virtual-GPUs interact with the memory manager to enable: (i) GPU sharing in the presence of concurrent applications with conflicting memory requirements, (ii) load balancing in case of GPU with different capabilities, GPU addition and removal, (iii) GPU fault tolerance, and (iv) checkpoint-restart capabilities.

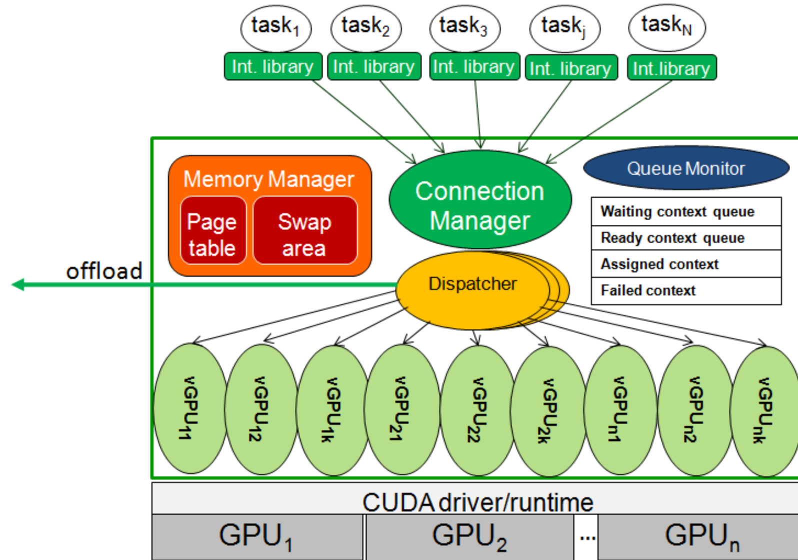


Figure 3-4 – Overall design of the runtime

3.5.2 Context Queues

The runtime system uses four context queues. (i) The *assigned context queue* contains a list of contexts that have been already scheduled on a GPU resource. (ii) The *ready context queue* stores contexts that have pending CUDA calls and are ready to be scheduled on a GPU resource. (iii) The *waiting context queue* stores contexts that have been preempted and unassigned from a GPU resource. (iv) The *failed context queue* contains contexts that have reported some CUDA error and need recovery actions.

3.5.3 Connection Manager

When used natively, the CUDA runtime spawns a CUDA context on the GPU for each application process. Different application processes can be directed to different GPUs by using the `cudaSetDevice` primitive. One of the goals of our runtime is to preserve the CUDA semantics. To this end, our frontend library opens a separate connection for each

application thread. CUDA calls belonging to different connections can, therefore, be served independently either on the same or on distinct GPUs. The connection manager enqueues connections generated by concurrent application processes in a *pending connections* list.

3.5.4 Dispatcher

The primary function of the dispatcher is to schedule CUDA calls issued by application threads onto GPUs. The dispatcher can be configured to use different scheduling algorithms: first-come, first-served, shortest-job-first, credit-based scheduling, etc. Some scheduling algorithms (e.g., shortest-job-first) require the dispatcher to make scheduling decisions based on the kernels executed by the applications, their parameters, and their execution configuration. Higher resource utilization and better performance can be achieved by supporting dynamic binding of applications to GPUs: the dispatcher must be able to modify the application-to-GPU mapping between kernel calls and to unbind applications from GPUs during their CPU-phases. These scheduling actions must be hidden from the users.

To enable informed scheduling decisions, the dispatcher must be able to delay application-to-GPU binding until the first kernel launch is invoked. Unfortunately, the very first CUDA calls issued by a CUDA application are not kernel launches, but synchronous internal routines used to register the GPU machine code (`_cudaRegisterFatBinary`), kernel functions (`_cudaRegisterFunction`), variables and textures (`_cudaRegisterVar`, `_cudaRegisterSharedVar`, `_cudaRegisterShared` and `_cudaRegisterTexture`) to the CUDA runtime.

Moreover, kernel launches are never the first non-internal CUDA calls issued by application threads; at the very least, they must be preceded by memory allocations and data transfers. Before kernel launches can be invoked by the client, all of these previous calls must be serviced.

Two observations help us overcome this problem. First, registration functions are always issued to the runtime prior to CUDA contexts' creation on the GPU. Therefore, these internal calls can be safely issued by the dispatcher well before the corresponding applications are bound to virtual-GPUs. The same holds for device management functions, some of which are ignored by our runtime (e.g. `cudaSetDevice`) or overridden (e.g., `cudaGetDeviceCount` will return the number of virtual, not physical, GPUs). Second, it is possible to delay GPU memory operations until the related data are accessed within kernel calls;the runtime responds to memory allocation requests by returning *virtual addresses*, and these virtual pointers are mapped to real device pointers at a later stage.

In summary, the dispatcher dequeues application threads from the list of *pending connections*, and handles them as follows. First, it issues registration functions to the CUDA runtime. Second, it services device management functions (and typically overrides them so as to hide the hardware setup of the node from the users). Third, it handles memory operations with the aid of the memory manager. In particular, the dispatcher does not issue memory operations directly to the CUDA runtime, but instead operates entirely in terms of virtual addresses generated by the memory manager. Fourth, if there are any free virtual-GPUs, the dispatcher schedules application threads to virtual-

GPUs (and enqueues them in the list of *assigned contexts*). If all virtual-GPUs are busy, application threads are enqueued in the list of *waiting contexts* for later scheduling. In addition, any failure during the execution of an application thread will cause it to be enqueued in a list of *failed contexts*, which is used by the dispatcher for recovery.

To prevent the dispatcher from being a bottleneck, its implementation is multithreaded; each dispatcher thread processes a different connection. All queues used within the runtime (*pending connections*; *waiting, assigned and failed contexts*) are accessed using mutexes.

3.5.5 Virtual GPUs

In order to allow time-sharing of GPUs, we spawn a configurable number of virtual-GPUs for each GPU installed on the system. A virtual-GPU is essentially a worker thread that issues calls originated from within application threads to the CUDA runtime. Virtual-GPUs are statically bound to physical GPUs through a `cudaSetDevice` invoked at system startup. Each virtual-GPU can service one application thread at a time. A virtual-GPU is idle when no application thread is bound to it and is active otherwise. Note that, since our runtime maps application threads onto virtual-GPUs and the CUDA runtime spawns a CUDA context for each virtual-GPU, this infrastructure preserves the semantics of the CUDA runtime. We experimentally observed (see Section 3.7) that the CUDA runtime cannot handle an arbitrary number of concurrent threads. Therefore, limiting the number of virtual-GPUs prevents our framework from overloading the CUDA runtime, and allows proper operation even in the presence of a large number of CUDA applications.

3.5.6 Memory Manager

The goal of the memory manager is to provide a virtual memory abstraction for GPUs. Two ideas are at the basis of the design. First, applications will not see device addresses returned by the CUDA runtime, but they will see virtual addresses generated by the runtime. Second, data resides in the host memory and is moved to the device only on demand. In this way, *the host* memory represents a lower level in the memory hierarchy: when some data must be moved to the device memory but the device memory capacity is exceeded, the memory manager *swaps* data from the device memory to the host memory. We allow memory swapping in two situations: (i) within a single application, and (ii) in the presence of multi-tenancy. The latter scenario is characterized by the presence of concurrent applications, each of whose memory footprints in isolation would fit within the device memory but whose aggregate memory requirements exceed the GPU memory capacity. In addition, the swap functionality allows an application to migrate from a less capable to a more capable GPU when the latter becomes available.

Host-to-device data transfers deferral must be done judiciously. Data transfers *preceding* the first kernel call cannot overlap with GPU computation and can, thus, be deferred without incurring performance losses. After the first kernel call, application-to-GPU binding is known; and our runtime can be configured to either defer or not defer data transfers. Not deferring allows computation-communication overlapping at the expenses of an increased swap overhead; deferring has the opposite effect.

The memory manager has two components: a *page table*[1], and a *swap area*. The page table stores the address translation, and the swap area contains not yet allocated or

swapped-out GPU data. The main data structures used in the memory manager are the following.

```
/* PAGE TABLE*/
typedef struct {
    void      * virtual_ptr;
    void      * swap_ptr;
    void      * device_ptr;
    size_t      size;
    bool        isAllocated;
    bool        toCopy2Dev;
    bool        toCopy2Swap;
    entry_t     type;
    void      * param
    nesting_t   nested;
} PageTableEntry

std::map<Context *, list<PageTableEntry *> *> PageTable;

/* CAPACITY AND UTILIZATION of AVAILABLE GPUs */
int numGPUs;
uint64_t * CapacityList;
uint64_t * MemAvaiList;
std::map<Context *, size_t> MemUsage;
```

Each page table entry (PTE), which is created upon a memory allocation operation, contains three pointers: the virtual pointer that is returned to the application (`virtual_ptr`), the pointer of the data in the swap area (`swap_ptr`), and, if the data are resident on the device, the device pointer (`device_ptr`). In addition, each entry has a size, a type, and possible additional parameters (`params` and `nested`). Finally, the flags `isAllocated`, `toCopy2Dev`, and `toCopy2Host` are used to guide device memory allocations, de-allocations and data transfers, as well as indicate whether the PTE has been allocated on device, whether the actual data reside only on the host, and whether the actual data reside only on device, respectively. The state transitions of the three flags depending on the call invoked by the application are illustrated in Figure 3-5. In particular, `malloc` represents any allocation operation (`cudaMalloc`, `cudaMallocArray`, etc.), whereas $\texttt{copy}_{\texttt{DH}}$ and $\texttt{copy}_{\texttt{HD}}$ represent any device-host and host-device data transfer function (`cudaMemcpy`, `cudaMemcpy2D`, etc.), respectively.

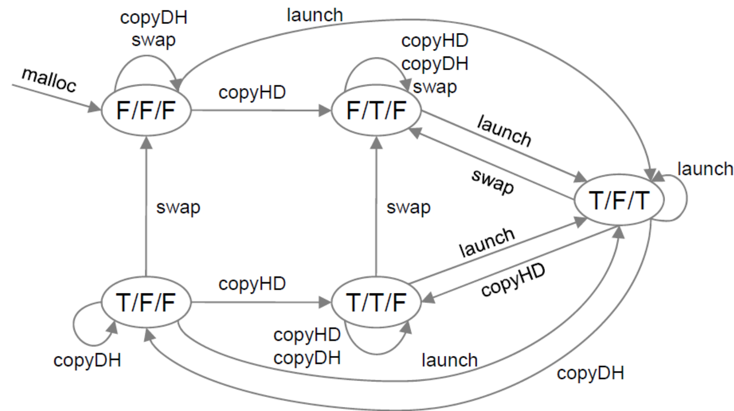


Figure 3-5 – State diagram showing the transition of `isAllocated/toCopy2Dev/toCopy2Swap` flags

Figure 3-5 assumes data transfer deferral and that all data referenced in a kernel launch can be modified by the kernel execution; a more fine-grained handling is possible if the information about read-only and read-write parameters is available. The attributes `type` and `params` allow distinguishing different kinds of memory allocations and data transfers associated with the entry. The nested attribute indicates whether the virtual address points to a nested data structure, or whether it is a member of it. Nested data structures must be declared to the runtime using a specific runtime API call, and are associated additional attributes describing their structure. These attributes are used by the memory manager in order to ensure consistency between virtual and device pointers within nested structures.

Each application thread (or *context*) has an associated list of PTEs; the page table contains all the PTEs for all the active and pending contexts in the node. In addition, the memory manager keeps track of the capacity and the memory availability of each GPU (`CapacityList` and `MemAvailList`) and of the memory usage of each context (`MemUsage`). This information is used to determine whether binding an application thread to a GPU can potentially lead to exceeding its memory capacity.

Routine issued	Action performed by the node-level scheduler	Error returned by the node-level scheduler
Malloc	Create VM_entry	A virtual address cannot be assigned.
	Allocate memory region in swap area	Memory region cannot be allocated.
$Copy_{HD}$	Check valid VM_entry	No valid VM_entry
	Move data to memory region	Swap-data size mismatch
$Copy_{DH}$	Check valid VM_entry	No valid VM_entry
	If (`VM_entry.toCopy2Swap`) `cudaMemcpy`$_{DH}$	-
Free	Check valid VM_entry	No valid VM_entry
	De-allocate memory region in swap area	Cannot de-allocate memory region
	If (`VM_entry.isAllocated`) `cudaFree`	-
Launch	Check valid VM_entry	No valid VM_entry
	If (!`VM_entry.isAllocated`) `cudaMalloc`	-
	If (`VM_entry.toCopy2Dev`) `cudaMemcpy`$_{HD}$	-
	`cudaLaunch`	-
Swap	Check valid VM_entry	No valid VM_entry
	If (`VM_entry.toCopy2Swap`) `cudaMemcpy`$_{DH}$	-
	If (`VM_entry.isAllocated`) `cudaFree`	-

Table 3-1 – For each routine issued, actions performed by the node-level scheduler and possible errors returned. A black in the third column indicates any error generated by the CUDA runtime (i.e. result ≠ cudaSuccess). VM_entry = Virtual Memory Entry.

Table 3-1 shows the actions performed by the runtime for each memory-related call invoked by the application. For simplicity, we show the data transfer deferral configuration. Note that, in this case, malloc and $copy_{HD}$ (data copy from host to device) do not trigger any CUDA runtime actions. *Swap* is an internal function that is triggered by the runtime when some data must be swapped from device to host memory to make room for data on the GPU. Like malloc; *swap* operates on a single page table entry. Two scenarios are possible: *intra-application swap* and *inter-application swap*. Independent of the kind, the swap operation can be triggered by the runtime while trying to allocate device memory to execute a kernel launch. Memory operations on nested structures will be extended also to their PTE members.

Intra-application swap–Consider the following sequence of calls coming from the same application *app*, where matmul is a matrix multiplication kernel for square matrices.

```
1. malloc(&A_d, size);
2. malloc(&B_d, size);
3. malloc(&C_d, size);
4. copyHD(A_d, A_h, size);
5. matmul(A_d, A_d, B_d);      //B_d = A_d * A_d
6. matmul(B_d, B_d, C_d);      //C_d = B_d * B_d
7. copyDH(B_h, B_d, size);
8. copyDH(c_h, C_d, size);
```

If the above application is run on the bare CUDA runtime and the data sizes are such that only two matrices fit the device memory, the execution will fail on the third instruction (that is, when trying to allocate the third matrix). On the other hand, when our

runtime is used, no memory allocation is performed until the first kernel launch (instruction 5). Previous instructions update only page table and swap memory. Instruction 5 will cause the allocation of matrices *A_d* and *B_d,* and the data transfer of *A_h* to *A_d*, and will execute properly. During execution of instruction 6, the runtime will detect the need for freeing device memory. Before trying to swap and unbind other applications from the GPU, the runtime will analyze the page table of *app* and detect that data *A_d*, not required by instruction 6, can be swapped to host. This situation will allow the application to complete with no error. In summary, *intra-application swap enables the execution of applications that would fail on the CUDA runtime even if run in isolation*. In other words, the maximum memory footprint of the "larger" kernel (rather than the overall memory footprint of the application) will determine whether the application can correctly run on the device.

Inter-application swap – This kind of swap may take place when concurrent applications mapped onto the same device have conflicting memory requirements. In particular, if device memory cannot be allocated and intra-application swap is not possible, the memory manager will be queried for applications running on the same GPU and using the amount of memory required. If such an application exists, it will be asked to swap. The application may or may not accept the request: for instance, an application running in a CPU phase with no pending requests may swap, but an application in the middle of a kernel call may not. If no application honors the swap request, the calling application will unbind from the virtual-GPU and retry later. Otherwise, all the page table entries belonging to the application that accepts the request will be swapped, and such application will be temporarily unbound from the GPU. There may be situations where

multiple applications must swap for the required memory to be freed. To reduce complexity and avoid inefficiencies, we do not trigger the swap in these situations. Note that inter-application swap implies coordination among virtual-GPUs and, as a consequence, has a higher overhead than intra-application swap. To avoid dead-locks, synchronization is required while accessing the page table. Finally, note that enabling swaps only during CPU phases allows GPU intensive applications to make full use of the GPU.

To determine whether a memory allocation can be serviced, the runtime will first use the memory utilization data in the memory manager (`CapacityList`, `MemAvailList`, and `MemUsage`). However, because of possible memory fragmentation on GPU, the runtime may need to use the return code of the GPU memory allocation function to ensure that the request can be honored. Moreover, there may be cases where only some GPUs have the required memory capacity.

Finally, we point out two additional benefits of our design. First, bad memory operations (for instance, data transfers beyond the boundary of an allocated area) can be detected by the memory manager without overloading the CUDA runtime with calls that would fail. Second, multiple data copy operations within the same allocated area (i.e., the same page table entry) will trigger a single, bulk memory transfer to the device memory.

3.5.7 Fault Tolerance & Checkpoint-Restart

The memory manager provides an implicit checkpoint capability that allows load balancing if more powerful GPU become idle, if GPUs are dynamically added and removed from the system, and recovery in case of GPU failures. For each application

thread, the page table and the swap memory contain the state of the device memory. In addition, an internal data structure (called *Context*) contains other state information, such as a link to the *connection* object, the information about the last device call performed, and, if the application thread fails, the error code. With this state information, dynamic binding allows redirecting contexts to different GPUs and resuming their operation. The dispatcher will monitor the availability of the devices and schedule contexts from the *failed contexts* list (in case of GPU failure or removal) and unbind and reschedule applications from the *assigned contexts* list in case of GPU addition. Our mechanism can be combined with BLCR [64] in order to enable these mechanisms also after a full restart of a node. Finally, our runtime has an internal checkpointing primitive that can be dynamically triggered after long running kernels, to allow fast recovery in case of failures.

3.5.8 Inter-node offloading

If the GPUs installed on a node are overloaded, our runtime can offload some application threads to other nodes. Note that this mechanism allows transferring only the CUDA calls originating within an application and not its CPU phases. In particular, the runtime redirects application threads in the list of *pending connections* to other nodes using a TCP socket interface. A measure of the load on the system is provided by the size of the list of *pending connections.* We allow the dispatcher to process *pending connections* only if the number of *pending contexts* is below a given threshold.

3.6 Supporting Preemptive GPU Sharing

3.6.1 Defining the Preemption Policy

We want to introduce a preemption mechanism that allows efficient GPU sharing for multi-process applications. To this end, we must define a preemption policy to be used: we must determine *when* preemption must be triggered. Since we are designing a light-weight operating system, we start by considering common OS preemption mechanisms. In operating systems, preemption is commonly implemented using the concept of time-quantum. Each process executes for a period of time, usually in the order of hundreds of micro-seconds. When the time-quantum expires, the process that currently occupies the processor is switched off from the processor, and its state and registers are saved in its process-control-block.

This approach, however, would be inefficient for our application. The motivations can be summarized as follows. First, GPU preemption can incur substantial overhead: a context-switch for GPU involves data transfers between host and device and the replay of all initializations routines (e.g., `_cudaRegisterFatBinary`, `_cudaRegisterFunction`, etc.) to the CUDA runtime. Thus, to avoid inefficiencies, the preemption rate should be kept low. Second, GPU preemption should be performed between kernel calls and not in the middle of a kernel's execution. Since different kernels may have different execution times, using a time quantum approach may be problematic.

Instead, we consider the following two preemption policies.

1. Maximum idle time-driven preemption–In this approach, assigned contexts are monitored for their inactivity period. If a context does not utilize the GPU for a

predefined period (the *maximum idle time*), then it will be preempted. The idea behind this policy is that processes waiting at synchronization points will stop issuing CUDA calls and, therefore, their associated contexts will become idle. During this inactivity period, the GPU can be yielded to other ready contexts. The main advantage of this policy is its relatively simple implementation; the disadvantages are the need for mechanisms to measure the inactivity period and to tune the maximum idle time parameter.

2. Synchronization call-driven preemption–In this approach, contexts are preempted when they invoke synchronization routines, such as MPI collective calls. The main advantage of this approach is that GPU is yielded immediately upon reaching a synchronization point and not after an idle time. The main disadvantages of this policy are two. First, it requires our runtime to intercept and handle not just CUDA calls, but also MPI calls (and possibly other multi-threading synchronization routines). Second, unless complex bookkeeping of the communicating processes is performed, this policy may lead to unnecessary preemptions (for example, preemption may occur also when the last process involved in the synchronization reaches the synchronization point).

We implemented both approaches, but achieved better results with the first one. In fact, the second approach was penalized by the overhead due to unnecessary preemptions. In the remainder of the paper, we will only present results obtained with the first approach.

3.6.2 Implementation

When a user submits a multi-process application to the system, the runtime receives a connection from each process and creates a context for each of them. The contexts are

initially put into the ready queue by the connection manager, where they wait for the dispatcher to perform scheduling operations. When a context from a multi-process application scheduled onto a vGPU reaches a global synchronization point, it may exhibit inactivity periods in two situations. First, it may need to wait for other slower contexts to also reach the synchronization point even if they are all concurrently scheduled onto vGPUs. Second, some of the other contexts in the same application may still be unassigned. In the latter case, there are not enough vGPUs to accommodate all the contexts. We describe the changes made to the system to support preemption and address these situations.

We implemented in our runtime system the preemption cycle represented in Figure 3-6. In particular, virtual GPUs are extended to actively monitor the inactivity period of their assigned context. To reduce the overhead, we activate this monitoring routine only when the ready context queue is not empty. In other words, preemption is inactivated when there are no pending contexts waiting for a GPU resource. If monitoring is enabled, and a context's inactivity period exceeds the maximum idle time; then, the corresponding vGPU will preempt such context.

Upon preemption, the following actions are performed. First, the vGPU stops executing the context (that is, it stops issuing its calls to the CUDA runtime). Second, the vGPU saves the state of the context, by transferring all data residing on the GPU back to the virtual memory space of the context on CPU. Third, the vGPU moves the context to the waiting queue. The context will now be periodically monitored by the queue monitor to check whether it has completed synchronization and is ready to be moved back to the ready queue. The queue monitor accomplished this action by observing whether the

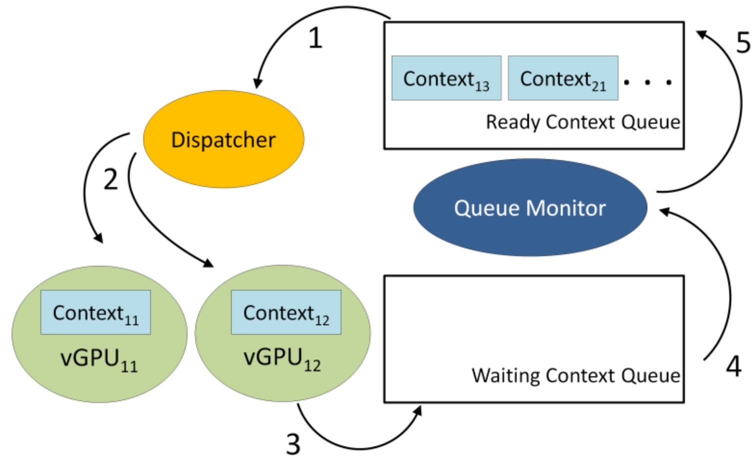


Figure 3-6 – Preemption cycle

context has more CUDA tasks issued through the frontend library. If the waiting queue contains several contexts, the queue monitor observes them in a round-robin fashion and moves the context with more CUDA tasks back to the ready queue. The dispatchers can then re-schedule the context to the same or to a different vGPU.

We further clarify this mechanism with an example, as illustrated in Figure 3-6. Suppose that we have a multi-process and a single-process application. The multi-process application consists of three contexts: $Context_{11}$, $Context_{12}$ and $Context_{13}$. The single-process application consists of a single context: $Context_{21}$. Initially, the dispatcher fetches $Context_{11}$ and $Context_{12}$ from the ready queue (step 1) and then schedules the two contexts to the vGPUs (step 2). These two contexts execute until they reach a global synchronization point. At this point, $Context_{11}$ and $Context_{12}$ become idle waiting for $Context_{13}$, which is in turn waiting in the ready queue. After the maximum idle time period elapses, the vGPUs determine that preemption is necessary because the ready queue is not empty (it contains $Context_{13}$ and $Context_{21}$). The vGPUs, therefore, preempt $Context_{11}$ and $Context_{12}$ and move them to the waiting queue (step 3). At this point, the

dispatcher can schedule $Context_{13}$ and $Context_{21}$ onto the two vGPUs. As $Context_{13}$ and $Context_{21}$ progress, $Context_{11}$ and $Context_{12}$ are monitored for readiness by the queue monitor (step 4). After all contexts belonging to the multi-process application synchronize, their processes will start issuing CUDA calls. Eventually, $Context_{11}$ and $Context_{12}$ are moved by the queue monitor back to the ready queue (step 5). The preemption cycle repeats until the pending queue becomes empty or until the multi-process application terminates.

3.7 Experimental Results

The experiments are divided into two parts. The first group of experiments (section 3.7.1), focus on single-process applications, while the second group (section 3.7.2) focuses on multi-process applications.

3.7.1 Single-process Application

3.7.1.1 Hardware Setup

The system used in our node-level experiments includes eight Intel Xeon E5620 processors running at 2.40 GHz and is equipped with 48 GB of main memory and three NVIDIA Fermi GPUs (two Tesla C2050s and one Tesla C1060). Each Tesla C2050 has 14 streaming multiprocessors (SMs) with 32 cores per SM, each running at 1.15 GHz, and three GB of device memory. The Tesla C1060 has 30 SMs with eight cores per SM, and four GB of device memory. In one experiment, we replaced the Tesla C1060 with the less powerful NVIDIA Quadro 2000 GPU, equipped with four 48-core SMs and one GB of device memory. In our cluster-level experiments, we used an additional node with the same CPU configuration but equipped with a single Tesla C1060 GPU card.

3.7.1.2 Benchmark

The benchmark applications used in our experiments are listed in Table 3-2. These applications, obtained from Rodinia Benchmark Suite [65] and NVIDIA's CUDA SDK, cover several application domains, and differ in their memory occupancy, their GPU intensity and their interleaving of computation between CPU and GPU. We divide the workload into two categories: *short-running* and *long-running* applications. When using a Tesla C2050 GPU, the former report a running time between three and five seconds

Program	**Description**	**Kernel calls #**
Short-running applications		
Back Propagation (*BP*)	Training of 20 neural networks with 64K nodes per input layer	40
Breadth-First Search (*BFS*)	Traversal of graph with 1M nodes	24
HotSpot (*HS*)	Thermal simulation of 1M grids	1
Needleman-Wunsch (*NW*)	DNA sequence alignment of 2K potential pairs of sequences	256
Scalar Product (*SP*)	Scalar product of vector pair (512 vector pairs of 1M elements)	1
Matrix Transpose (*MT*)	Transpose (384x384) matrix	816
Parallel Reduction (*PR*)	Parallel reduction of 4M elements	801
Scan (*SC*)	Parallel prefix sum of 260K elements	3,300
Black Scholes – small (*BS-S*)	Processing of 4M financial options	256
Vector Addition (*VA*)	100M-element vector addition	1
Long-running applications		
Small Matrix Multiplication (*MM-S*)	200 matrix multiplication of 2Kx2K square matrices and variable CPU phases	200
Large Matrix Multiplication (*MM-L*)	10 matrix multiplications of 10Kx10K square matrices and variable CPU phases	10
Black Scholes – large (*BS-L*)	Processing of 40M financial options	256

Table 3-2 – Benchmark programs

each, and the latter between 30 and 90 seconds (depending on the CPU phase injected– see Section 3.7.1.3). In the third column of Table 3-2, we report the number of kernel calls performed by each application. All short-running applications and *BS-L* are GPU intensive and have memory requirements well below the capacity of the GPUs in use. *MM-S* and *MM-L* are injected CPU phases of different length; *MM-L* has high memory requirements.

3.7.1.3 Node-level Experiments

Overhead Evaluation–First, we measured the overhead of our framework with respect to the CUDA runtime. We allowed our runtime to use only one physical GPU and varied the number of virtual GPUs (*vGPUs*). The execution time of the bare CUDA runtime gives a lower bound that allows us to quantify the overhead associated with our framework. The data in Figure 3-7 were obtained by randomly drawing jobs from the pool of short-running applications in Table 3-2 and averaging the results over ten runs. To ensure apple-to-apple comparison, we run each of the randomly drawn combination of jobs on all 5 reported configurations (bare CUDA runtime and our runtime using 1, 2, 4 and 8 *vGPUs*).

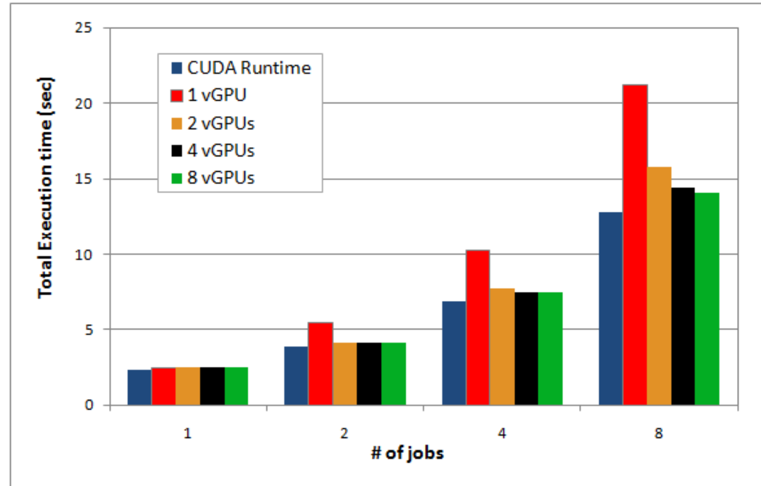


Figure 3-7 – Execution time reported with a variable number of short-running jobs on a node with one GPU. The bare CUDA runtime is compared with our runtime

Since our experiments showed that the CUDA runtime cannot handle more than eight concurrent CUDA contexts, we limited the number of jobs to eight. As can be seen in Figure 3-7, the total execution time of our runtime approaches the lower limit (CUDA runtime) as we increase the number of *vGPUs*. Increasing the number of *vGPUs* means increasing the sharing of the physical GPU, thus amortizing the overhead of the framework (which, in the worst case, accounts for about 10% of the execution time). Note that the percentage overhead would decrease on long-running applications.

Benefits of GPU Sharing – In our second set of experiments, we evaluated the effect of GPU sharing in the presence of more (three) physical GPUs. We used the same workload as in previous experiment and again varied the number of *vGPUs* per device. We recall that the number of *vGPUs* represents the number of jobs that can time-share a GPU.

As mentioned in the previous section, we found that the CUDA runtime does not currently support more than eight concurrent jobs stably. Therefore, we do not report

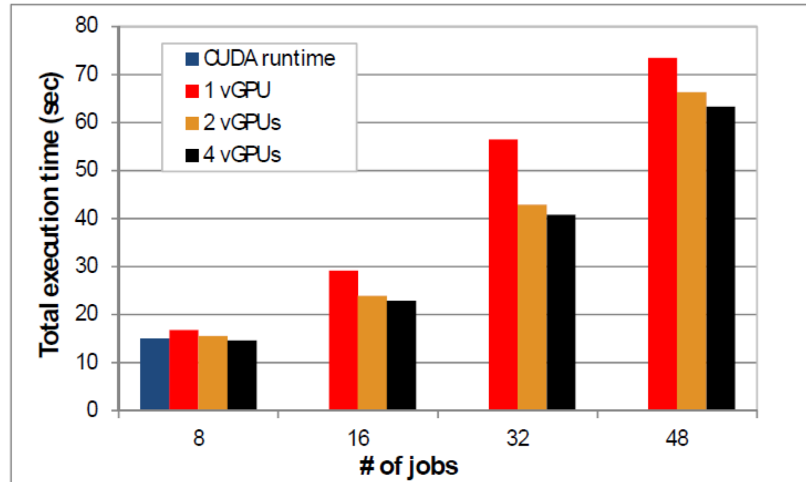


Figure 3-8 – Execution time reported with a variable number of short-running jobs on a node with three GPUs. The bare CUDA runtime cannot handle more than eight concurrent jobs.

results using the bare CUDA runtime beyond eight jobs. Figure 3-8 shows that, when using four *vGPUs* per device, our runtime reports some performance gain compared to the bare CUDA runtime. In fact, the overhead of our framework is compensated by its ability to load balance jobs on different physical GPUs. When running higher number of concurrent jobs, our results confirm our previous finding that increasing the amount of GPU sharing positively impacts the performances. However, we do not observe significant performance improvements when more than four *vGPUs* are employed. We believe that four *vGPUs* per device provide a good compromise between resource sharing and runtime overhead, and we use this setting in the rest of our experiments.

Conflicting Memory Needs: Effect of Swapping–The effect of swapping can be evaluated by using memory-hungry applications. To this end, we considered large matrix multiplication (*MM-L*). This benchmark program performs ten square matrix multiplications on randomly generated matrices. We set the data set size so to have conflicting memory requirements when more than two jobs are mapped onto the same

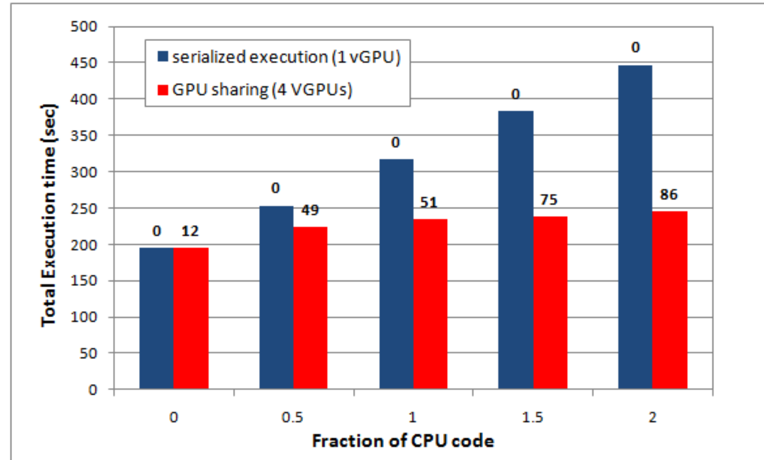


Figure 3-9 – 36 *MM-L* jobs (with conflicting memory requirements) are run on a node with three GPUs. The fraction of CPU code in the workload is varied. We indicate the number of *swap* operations occurred on top of each bar

GPU. In addition, we injected in the matrix multiplication benchmark CPU phases of various size. CPU phases are interleaved with kernel calls and simulate different levels of post-processing on the product of the matrix multiplication.

The effect of swapping is evaluated by running 36 *MM-L* jobs concurrently. In order to compare the swapping and no-swapping cases, we conducted experiments with one *vGPU* (no swapping required) and four *vGPUs* (swapping required). We recall that, in the one *vGPU* case, jobs run sequentially on a physical GPU; and, therefore, there is no memory contention. In the experiment, the fraction of CPU work is varied while maintaining the level of GPU work. Figure 3-9 shows that the total execution time grows linearly with the fraction of CPU work in the case of serialized execution (1 *vGPU*). In the case of GPU sharing (4 *vGPUs*), the overall execution time is kept constant even if the amount of work in each job increases. In fact, swapping can effectively reduce the total execution time by hiding the CPU-driven latency. In the chart, the number on the top of each bar indicates the swap operations occurred during execution. This experiment

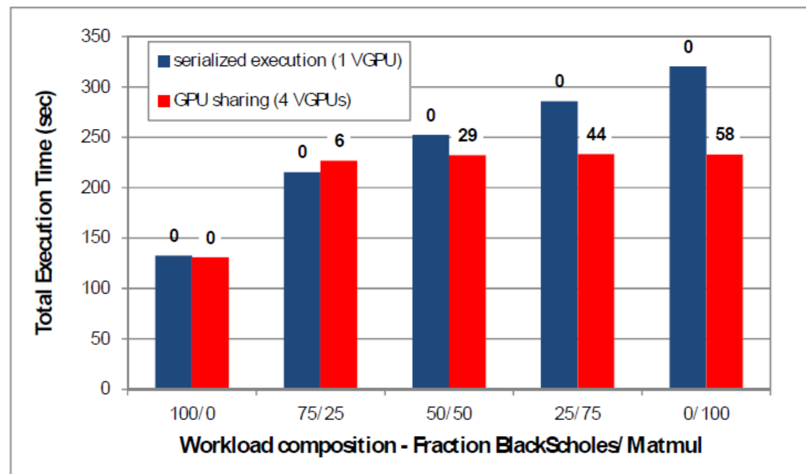


Figure 3-10 – 36 jobs (BS-L and MM-L) are run on a node with three GPUs. The workload composition is varied. We indicate the number of *swap* operations that occurred on top of each bar.

demonstrates that our swapping mechanism can effectively resolve resource conflicts among the concurrently running applications. In addition, despite its overhead, this mechanism provides performance improvement to applications with a considerable fraction of CPU work.

We next investigated the performance of our runtime when combining applications with different amount of CPU work. In particular, we mixed *BS-L* with *MM-L* at different ratios (Figure 3-10). *BS-L* is a GPU-intensive application with short CPU phases, whereas *MM-L* was set to have a fraction of CPU work equal to 1. The memory requirements of *BS-L* are below those of *MM-L*. Again, we ran 36 jobs concurrently. The results of these experiments are shown in Figure 3-10. Again, the number on the top of each bar indicates the number of swap operations that occurred during execution. As one might expect, the performance gain from GPU sharing increases as *MM-L* becomes dominant. Because *BS-L* is a GPU intensive application and swapping adds additional

overhead; this results in a longer execution time for four *vGPUs* at a 75/25 mix of *BS-L* and *MM-L*.

Benefit of Dynamic Load Balancing–In Figure 3-11, we show the results of experiments performed on an unbalanced node that contains two *fast* and one *slow* GPUs: two Tesla C2050s and one Quadro 2000, respectively. In one setting, our runtime performs load balancing as follows. The dispatcher keeps track of fast GPUs becoming idle; and, in the absence of pending jobs, it migrates running jobs from slow to fast GPUs.

The experiments are conducted on *MM-S* jobs with varying CPU fraction and using four *vGPUs* per device. The number of jobs migrated is reported on top of each bar. As can be seen, despite the overhead due to job migration, load balancing through dynamic binding of jobs to GPUs is an effective way to improve the performances of an unbalanced system. This holds especially in the presence of small batches of jobs and of

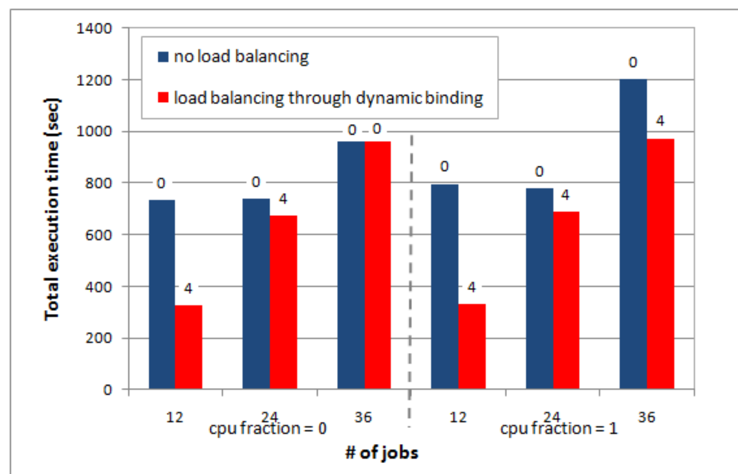


Figure 3-11 – Unbalanced node with two Tesla C2050s and one Quadro 2000: effect of load balancing through dynamic binding. The number of *MM-S* jobs migrated to *fast* GPUs is reported on top of each bar.

applications alternating CPU and GPU phases. As the number of concurrent jobs increases, the system performs load balancing by scheduling on fast GPUs pending jobs rather than by migrating jobs already running on slow devices.

3.7.1.4 Cluster-level Experiments

We have integrated our runtime with TORQUE, a cluster-level scheduler that can be used to run GPU jobs on heterogeneous clusters. In this section, we show experiments performed on a cluster of three nodes. The jobs are submitted at a head node and executed on two compute nodes. The hardware configuration of the compute nodes is described in Section 3.7.1.1. Having a three and a single-GPU compute node, our cluster is unbalanced.

When TORQUE is used on a cluster equipped with GPUs, it relies on the CUDA runtime to execute GPU calls. Since the CUDA runtime does not provide adequate support to concurrency, TORQUE does not allow any form of GPU sharing across jobs. Therefore, when configured to use compute nodes equipped with GPUs, TORQUE serializes the execution of concurrent jobs by enqueuing them on the head node and submitting them to the compute nodes only when a GPU becomes available. *By coupling TORQUE with our runtime system, we are able to provide GPU sharing to concurrent jobs.*

When coupling TORQUE with our runtime, we conducted experiments with three settings. In all cases, to force TORQUE to submit to the compute nodes more jobs than available GPUs, we hid from TORQUE the presence of GPUs and handled it only within our runtime. In the first setting, our runtime was configured to use only one *vGPU* per device, and, therefore, to serialize the execution of concurrent jobs. In the second setting,

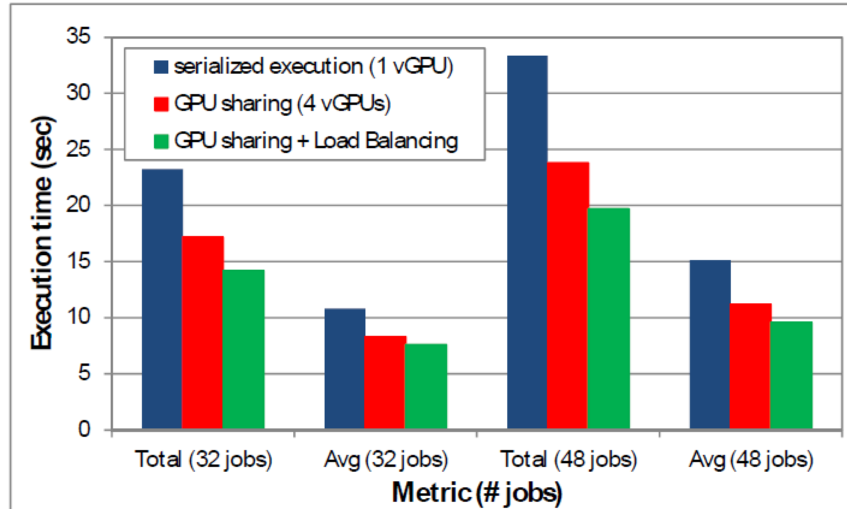


Figure 3-12 – Two-node cluster using TORQUE: effect of GPU sharing and load balancing via inter-node offloading in the presence of short-running jobs and in the absence of conflicting memory requirements.

we allowed GPU sharing by using four *vGPUs* per device. In the third setting, we additionally enabled load balancing across compute nodes by allowing inter-node communication and offloading. We also performed experiments using TORQUE natively on the bare CUDA runtime. However, the results reported using this configuration are far worse than those reported using TORQUE in combination with our runtime. Therefore, we show the use of our runtime with one *vGPU* per device as an example of no GPU sharing.

In Figure 3-12 – Two-node cluster using TORQUE: effect of GPU sharing and load balancing via inter-node offloading in the presence of short-running jobs and in the absence of conflicting memory requirements.,, we show experiments conducted using a variable number of short-running jobs drawn from the applications in Table 3-2. In this set of experiments, jobs do not exhibit conflicting memory requirements. Again, we average the results reported over ten runs. As can be seen, GPU sharing allows up to a

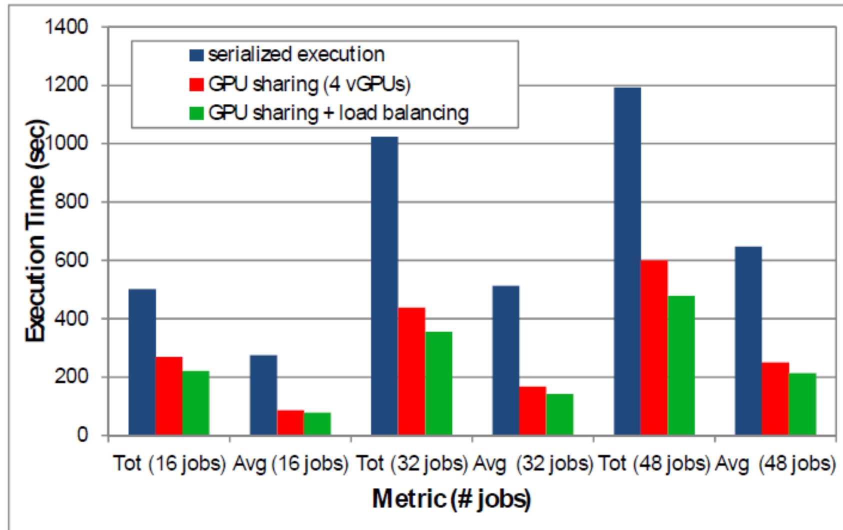


Figure 3-13 – Two-node cluster using TORQUE: effect of GPU sharing and load balancing via inter-node offloading in the presence of long-running jobs and conflicting memory requirements.

28% performance improvement over serialized execution. However, TORQUE, which relies on our runtime and is unaware of the number and location of the GPUs in the cluster, divides the workload equally between the two nodes. Thus, the node with only one GPU is overloaded compared to the other node with three GPUs. When, in addition to GPU sharing, we allow load balancing through our inter-node offloading technique, the overall throughput is further improved by up to 18%.

Finally, we want to show the benefits of our runtime system in a cluster in the presence of jobs with conflicting memory requirements. To this end, we run 16, 32 and 48 *BS-L* and MM-L jobs (25/75 distribution). We recall that these two applications have long runtimes. The results of this experiment are shown in Figure 3-13. Again, serialized execution allows avoiding memory conflicts. From the figure, it is clear that allowing jobs to share GPUs increases the throughput significantly (up to 50%) despite the overhead due to the need for swap operations. Moreover, in the presence of load

imbalances, the execution is further accelerated by allowing the overloaded node to offload the excess jobs remotely.

3.7.2 Multi-process Application

The goal of our experiments is to evaluate our proposed preemptive GPU sharing scheme and compare it with batch scheduling and with controlled *n*-way GPU sharing using a collection of benchmark programs with various characteristics. In particular, we want to evaluate how the number of processes per application, the duration of the CPU and GPU computation within the application, the communication pattern, and the degree of intra- and inter-application imbalance affect the performance of the considered scheduling and sharing schemes.

3.7.2.1 Benchmark Description

Since there is no established benchmark suite for applications that combine MPI and CUDA library calls, we decided to write a benchmark generator for such programs. Our benchmark generator is a C++ tool that automatically generates MPI-CUDA applications according to user-specified parameters.

Parameters	Description	Settings in the experiments		
		Preliminary	*Node-level*	*Cluster-level*
Number of processes	Number of MPI processes	4	4	4, 6, 8
Communication type	MPI communication type	All-to-all	Broadcast, Scatter-Gather, Barrier synchron.	Scatter-Gather
Number of iterations	Number of iterations executed by each MPI process	200	200 (0.3 sec/kernel), 20 (3.0 sec/kernel)	200
Number of execution phases	Number of phases per iteration	1	1	1
GPU phase duration	Duration of per-phase GPU computation	0.5 sec	0.3 sec - 3.0 sec	0.3 sec
CPU phase duration	Duration of per-phase CPU computation	0	0%, 50%, 100% the GPU phase duration	0
Size of GPU transfers	Size of memory transfers between CPU and GPU	100KB	100KB	100KB
Size of MPI transfers	Size of data transferred by each communication primitive	400KB	400KB (broadcast, gather-scatter) 0KB otherwise	100KB * # of processes

Table 3-3 - Description and setting of the parameters of our benchmark generator

To define the basic structure of the applications to be generated, we analyzed the structure of the NAS Parallel Benchmarks[1], which include a variety of MPI applications. We observed that several such applications are iterative. Every iteration consists of one or more execution phases; in turn, each execution phase contains some communication and some computation code. We decided to generalize this computation pattern by allowing offloading some computation to the GPU. Specifically, in our model, every execution phase consists of some communication, some CPU and some GPU computation (which, in turns, includes data transfers between host and device and some GPU kernel invocations).

The parameters that drive the generation of the benchmark applications are listed in Table 3-3. In the table, all parameters (but the first) refer to a single MPI process. As

[1] http://www.nas.nasa.gov/publications/npb.html

can be seen, the generated applications mainly differ in their communication patterns, their CPU and GPU computation intensity, their interleaving of computation between CPU and GPU, and their duration. CPU and GPU computation are currently implemented using a number of vector additions of various sizes. To realize the specified CPU and GPU durations, we dynamically modify the number of vector additions executed (we have profiled the CPU and GPU execution time of vector additions of predefined sizes).

Table 3-3 also specifies the parameter settings used in the experiments presented in this work. The duration of the execution phases and the size of the MPI data transfers were chosen according to profile information collected in the analysis of the NAS Parallel Benchmarks. We assumed that, in each execution phase, the computation can take place either completely on GPU, or partially on CPU (to this purpose, we varied the CPU phase duration as shown in the table). In our preliminary experiments, we used the all-to-all communication pattern. However, in most of our experiments we focused on three communication patterns: broadcast, scatter-gather communication and barrier synchronization, which are characteristics of applications such as distributed matrix multiplication and N-body simulation.

3.7.2.2 Experimental Setup & Evaluation Metric

Hardware setup–The experiments described in Section 0 and 3.7.2.4 were performed on a single node; the ones presented in Section 3.7.2.5 were performed on a two-node cluster. The hardware setup of our cluster is summarized in Table 3-4. Single-node experiments were conducted on $Node_1$.

Software setup–All experiments described in this Section were performed using the runtime system. Batch scheduling is implemented by setting the number of virtual

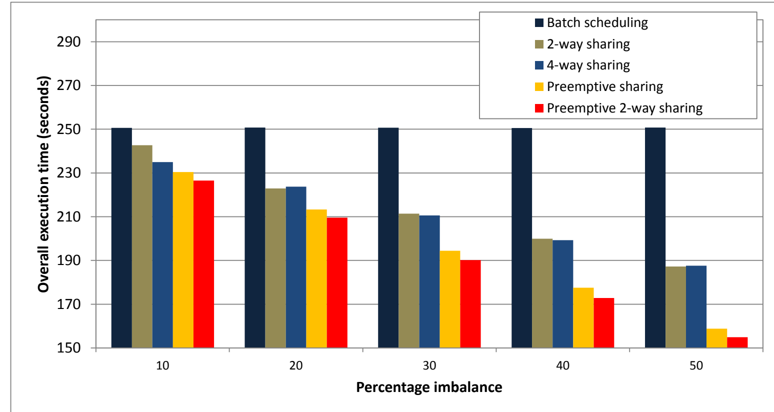
GPUs per physical GPU to 1. As far as the implementation of MPI is concerned, we use MPICH2 (http://www.mpich.org).

Maximum Idle Time Setting–Our proposed preemptive GPU sharing scheme requires setting the *maximum idle time* parameter, which indicates how long a GPU should remain idle before the runtime system performs a preemption operation. Setting this parameter involves evaluating a trade-off between runtime overhead and GPU underutilization. If the maximum idle time is set to a large value, preemption will be rarely invoked--leading to minimal runtime overhead but possibly to GPU underutilization. Conversely, if this parameter's value is too low, preemption operations may be triggered too frequently, causing the runtime overhead to outweigh the benefits from an increased GPU utilization. To set this parameter, we have measured the runtime overhead due to preemption and involved with binding a process to a virtual GPU. We have found that setting the maximum idle time to 0.01 seconds allows a good trade-off between runtime overhead and GPU underutilization, and we have used this setting in all experiments presented in this section.

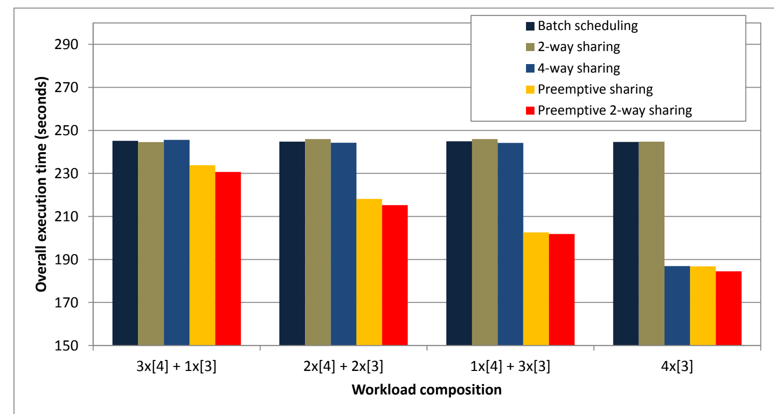
Evaluation Metric–In all our experiments, we have measured the *overall execution time* for running a variable number of MPI jobs on a single node or on a two-node cluster. In the charts presented in Section 3.7.2.4, we show the *speedup* (in terms of overall execution time) reported by *n*-way and preemptive GPU sharing over simple batch scheduling.

Node	Attributes	Values
$Node_1$	*# CPU cores*	8
	CPU cores	Intel Xeon® E5620, 2.4 GHz 12 MB cache
	Main memory	48 GB
	Operating System	CentOS 5
	CUDA version	3.2
	# GPUs	4
	GPUs	Nvidia GeForce GTX 480 15 SM x 32 cores 1 GB Global memory
$Node_2$	*# CPU cores*	12
	CPU cores	Intel Xeon ® E5-2620, 2.00GHz 15 MB cache
	Main memory	64 GB
	Operating System	CentOS 6
	CUDA version	3.2
	# GPUs	3
	GPUs	Nvidia Tesla C2070 14 SM x 32 cores ~6 GB Global memory
		Nvidia Tesla C2075 14 SM x 32 cores ~6 GB Global memory
		Nvidia Tesla C2050 14 SM x 32 cores ~3 GB Global memory

Table 3-4 – Characteristics of the nodes



(a) – Intra-application imbalance



(b) – Inter-application imbalance

Figure 3-14 – Preliminary experiments: comparison among five scheduling and sharing schemes for a 4-job workload including *all-to-all* communication primitives. In (5-a), each job consists of four processes, and the *GPU phase duration* parameter of one of these processes is higher than that of the other three by a factor *percentage imbalance*. In (5-b), workload composition $j_1 \times [p_1] + j_2 \times [p_2]$ indicates that j_1 jobs consist of p_1 processes and j_2 jobs consists of p_2 processes.

3.7.2.3 Preliminary Experiments

Before exploring different computation and communication patterns, we compared the considered scheduling and GPU sharing schemes using a 4-job workload that employs the `MPI_Alltoall` communication primitive (see column 3 in Table 3-3). We recall that this set of experiments was performed on the 4-GPU *Node*$_1$ machine. We considered

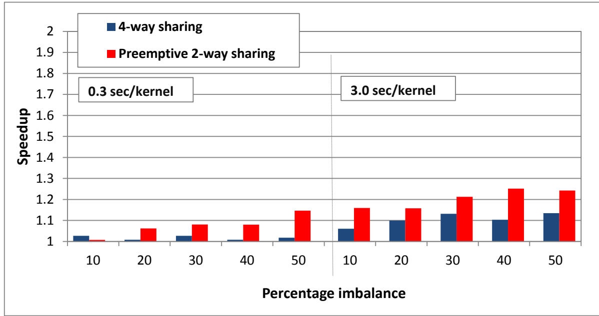
the first two situations described in Section 3.4: intra- and inter-application imbalance. In both cases, we compared batch scheduling, two- and four-way sharing, preemptive sharing, and the combination of preemption and two-way sharing (preemptive two-way sharing). The results are shown in Figure 3-14.

Figure 3-14(a) covers the *intra-application imbalance* case. In this set of experiments, every job consists of four processes. Intra-application imbalance is due to the fact that one process is slower than the other three. Specifically, its *GPU phase duration* parameter (0.5 sec) is set to be higher than that of the other three processes by a factor *percentage imbalance*. This factor is varied along the *x*-axis of Figure 3-14(a) from 10% to 50%, thus leading to an increasing amount of intra-application imbalance.

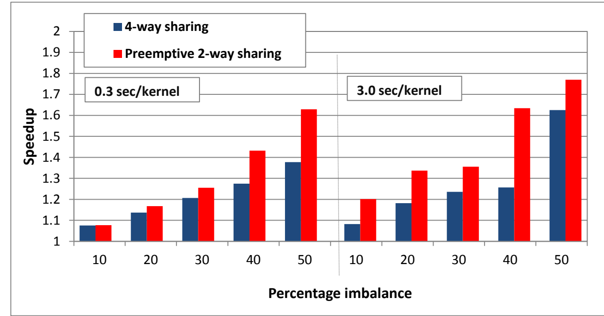
Figure 3-14(b) covers the inter-application imbalance situation. In this case, all the processes within each job have the same CPU and GPU execution time. However, the workload composition is now heterogeneous: the number of processes is not uniform across the jobs. In particular, workload composition $j_1 \times [p_1] + j_2 \times [p_2]$ indicates that j_1 jobs consist of p_1 processes, and j_2 jobs consists of p_2 processes. Note that all configurations contain jobs consisting of three processes. Since the node has four GPUs, these jobs cause imbalance. In particular, the number of jobs with three processes is varied along the x-axis, leading to an increasing amount of inter-application imbalance.

From these preliminary experiments, we can make the following observations. First, batch scheduling fails to capture both intra- and inter-application imbalance and to correct their negative effects on the performance. In fact, when using batch scheduling, the overall execution time stays constant across all the experiments and depends on the slowest job. Second, in case of intra-application imbalance, *n*-way sharing leads to

performance improvements over batch scheduling; and this gain increases with the imbalance. In fact, GPU sharing allows the idle time of one process to be hidden behind the GPU execution of co-located processes. In case of inter-application imbalances, *n*-way sharing is beneficial only when the workload is highly imbalanced and when 4 processes are allowed to share the same GPU device. Third, by increasing the GPU utilization, preemptive GPU sharing greatly outperforms the other schemes in both the intra- and the inter-application imbalance scenarios. Again, the performance gain increases with the degree of imbalance. Finally, best performance can be achieved by combining 2-way sharing and preemption (preemptive 2-way sharing case). This trend has been observed in all experiments performed. For readability, in the remainder of this section we will consider only batch scheduling, 4-way sharing and preemptive 2-way sharing, and report the speedup of these two sharing schemes over batch scheduling.

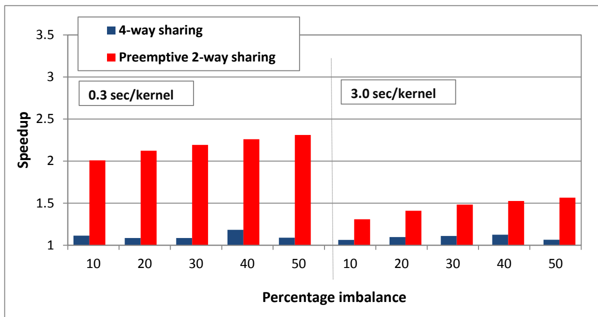


(a) Worst case

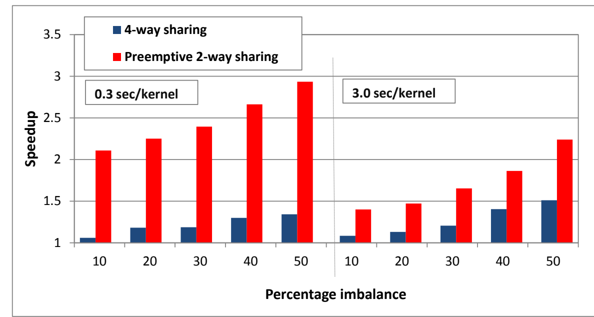


(b) Best case

Figure 3-15 Intra-application imbalance--speedup for broadcast communication pattern

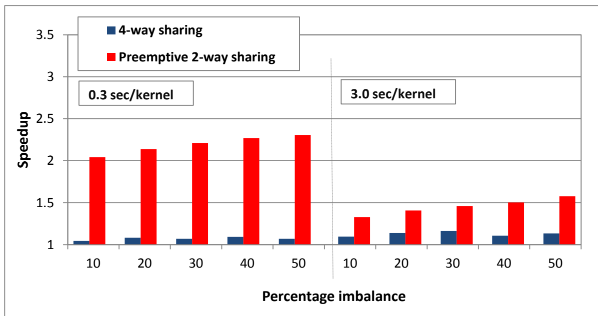


(a) Worst case

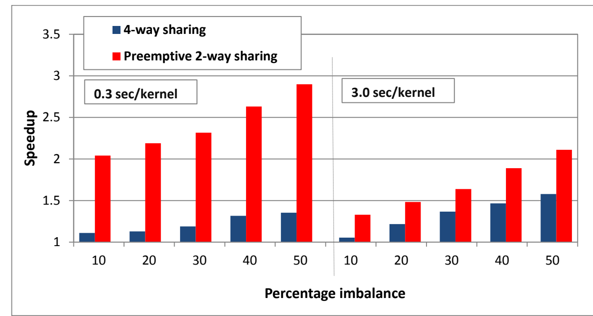


(b) Best case

Figure 3-16 Intra-application imbalance--speedup for scatter-gather communication pattern

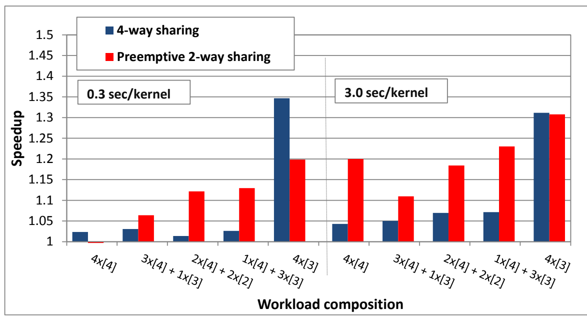


(a) Worst case

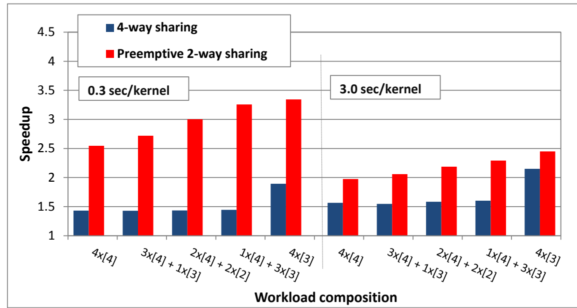


(b) Best case

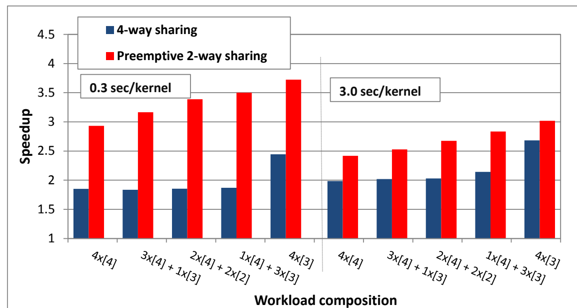
Figure 3-17 Intra-application imbalance--speedup for barrier synchronization pattern



(a) 0% CPU phase

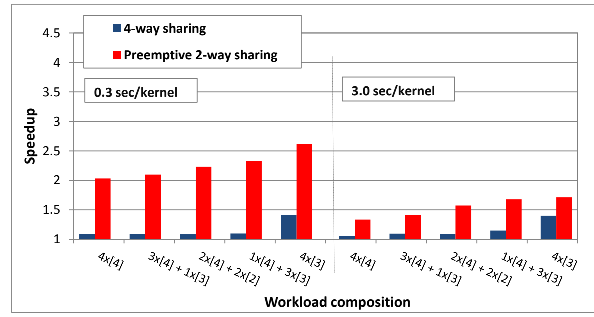


(b) 50% CPU phase

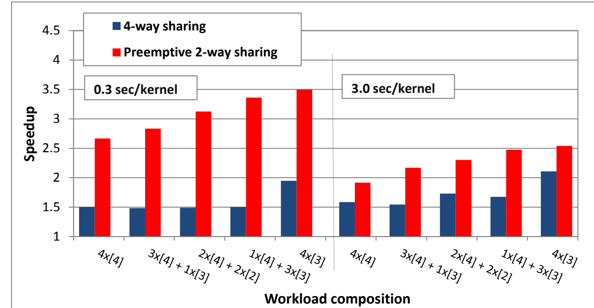


(c) 100% CPU phase

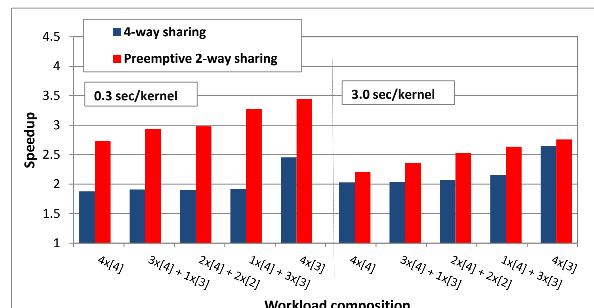
Figure 3-18 Inter-application imbalance-- speedup for broadcast pattern



(a) 0% CPU phase

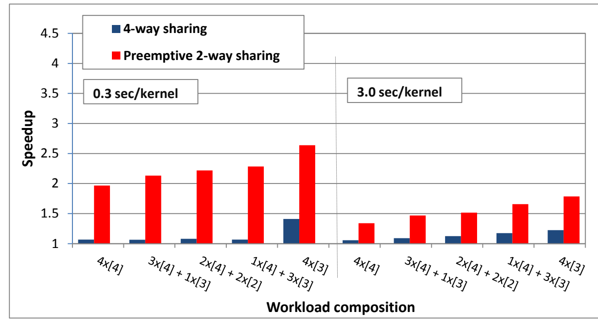


(b) 50% CPU phase

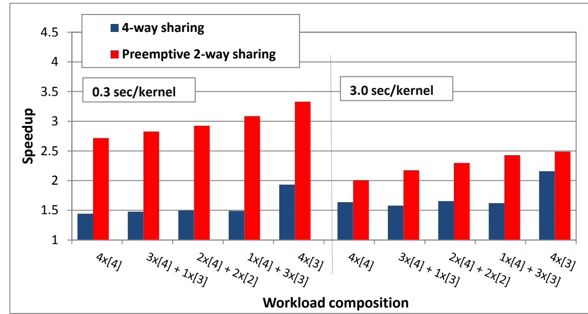


(c) 100% CPU phase

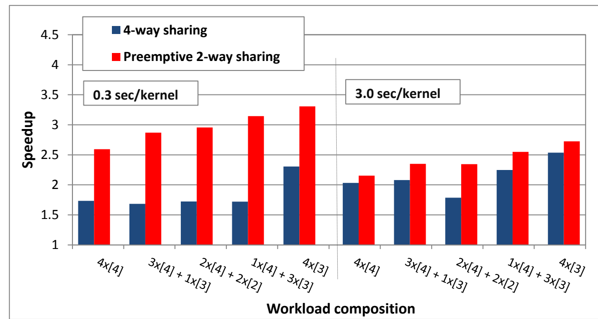
Figure 3-19 Inter-application imbalance-- speedup for scatter-gather pattern



(a) 0% CPU phase



(b) 50% CPU phase



(c) 100% CPU phase

Figure 3-20 - Inter-application imbalance--speedup for barrier pattern

3.7.2.4 Node-level Experiments

In this section, we show a set of node-level experiments meant to evaluate the proposed scheduling and sharing schemes on workloads characterized by a variety of computation and communication patterns. The parameter settings of the workload generator used for this group of experiments are shown in the fourth column of Table 3-3. As can be seen, the exploration space can be described as follows: First, we consider three collecting communication patterns, broadcast, scatter-gather and barrier synchronization. Second, we use shorter and longer GPU phases (namely, 0.3 and three sec, respectively). Third, we add an increasing amount of CPU computation to each execution phase: specifically, we consider three settings: one with no CPU computation, one where the CPU computation is half as long as the GPU computation, and one where the two have the same duration (CPU phase duration parameter set to 0%, 50% and 100%, respectively). We want every process to use the GPU for approximately 60 seconds in all experiments. To this end, we vary the number of iterations (20 or 200) depending on the GPU phase duration used. Note that this will affect the synchronization rate (since each iteration contains a communication/synchronization primitive). As in our preliminary experiments, we want to evaluate the speedup of the sharing schemes over batch scheduling in two situations: intra- and inter-application imbalance. In all cases, we use a workload consisting of four jobs.

Intra-application imbalance--In this first set of experiments, all jobs consist of four processes with no CPU computation (except for inter-process communication). The default kernel time is set either to 0.3 or to three seconds to allow two synchronization

rates. To introduce intra-application imbalance, we decrease the GPU phase duration of some processes in each job by 10 to 50 percent of the default kernel time. We consider two workload configurations, that we call *best case* and *worst case*. In the former, three out of four processes per application are made faster. In the latter, only one process per application is faster than the other three.

The results reported on workloads using the broadcast, the gather-scatter and the barrier communication patterns are shown in Figure 3-15, Figure 3-16 and Figure 3-17, respectively. As can be seen, preemption achieves better performance in almost every case. In a few cases (e.g., broadcast communication pattern & low imbalance percentage), the preemption overhead can have a dominant effect and deteriorate the performance. A speedup of 1.7X, 2.9X and 2.9X over the batch scheduling can be observed in the broadcast, scatter-gather and barrier communication cases, respectively. Batch scheduling and four-way sharing fail to capture the effect of heterogeneity, especially in *worst case* configurations. We also note that the speedup observed with broadcast communication is relatively small compared to that achieved with the other communication patterns. Processes that perform broadcast communication exhibit fewer inactivity periods than processes using other patterns. This is because the broadcast operation does not require every process to enter the communication routine at the same time in order to proceed. Different processes can progress independently, even under the imbalance. The overhead of the communication can, therefore, be adequately hidden even by the four-way sharing mechanism.

Inter-application imbalance–In this second set of experiments, there is no execution imbalance within each job. However, inter-application imbalance is introduced

by allowing different jobs in the workload to include a different number of processes. As in our preliminary experiments, we use mixes of jobs with four and three processes each, and vary the number of jobs consisting of three processes.

The results reported on workloads that use the broadcast, the gather-scatter and the barrier communication patterns are shown in Figure 3-18, Figure 3-19 and Figure 3-20, respectively. The workload composition along the *x*-axis of these charts can be read as in the preliminary experiments. We make the following observations: First, preemption can again effectively improve the performance by minimizing the GPU underutilization. This mechanism achieves up to a 3.5X and a 2.1X speedup over batch scheduling and four-way sharing, respectively. Second, when using the broadcast pattern, performance improvements can be observed in most of the configurations. As we have learned from the previous experiments, the broadcast pattern rarely blocks processes when performing communication. Therefore, four-way sharing is sufficient to hide the communication latency. In addition, the preemption overhead can be unjustified in case of balanced workload compositions (e.g. *4x[4]*). We further observe that, as we increase the percentage of the CPU phase, the speedup of preemption over four-way sharing decreases slightly. In fact, four-way sharing can overlap the GPU and CPU phases of different applications without incurring preemption overhead.

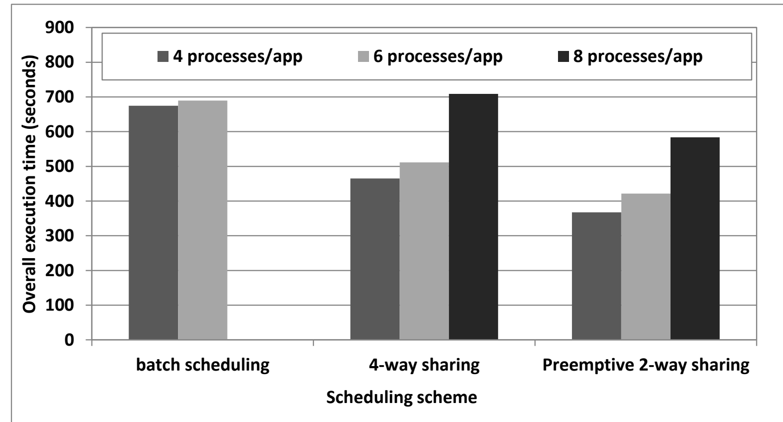


Figure 3-21 Overall execution time in cluster settings

3.7.2.5 Cluster-level Experiments

In this section, we show the results of experiments conducted on the two-node heterogeneous cluster of Table 3-4. The jobs, whose characteristics are summarized in the fifth column of Table 3-3, use the scatter-gather communication pattern, do not include CPU computation (except for MPI communication), and do not present intra-application imbalance. We conducted three sets of experiments using 4, 6 and 8 processes per job, respectively. In all experiments, the workload consists of eight jobs.

Figure 3-21 reports the execution time using batch scheduling, four-way sharing, and preemptive two-way sharing. The dataset in case of batch scheduling and eight processes per job is missing because our cluster has a total of 7 GPUs; and, as explained in Section 3.4 (scenario 3), batch scheduling cannot support applications requesting more GPUs than physically available in the system. As can be seen, four-way sharing and preemptive GPU sharing lead to a 25-30% and a 40-45% performance improvement over batch scheduling, respectively. In addition, both sharing mechanisms allow the execution

of jobs with more processes than available GPUs. As explained in Section 3.4, four-way sharing would not be able to support jobs with more than 7*4=28 processes. Preemption-based scheduling is the only mechanism able to support applications with any number of processes, even in the presence of synchronization.

CHAPTER 4

CLUSTER-LEVEL SCHEDULER

4.1 Objectives

We recall from the introduction that because of the widespread use of GPUs, popular cluster resource managers, such as TORQUE [8] and SLURM [9], have been recently extended with GPU support capabilities. These systems typically treat GPUs as dedicated resources (assigned to one application at a time) and rely on the GPU software stack (i.e., the CUDA driver and runtime) for the scheduling of GPU-accelerated kernels. These traditional cluster resource managers suffer from the following limitations.

First, they do not handle heterogeneity well. Nowadays clusters present heterogeneity at different granularities. First, a single node may comprise multiple GPUs with different compute capabilities and memory capacities. Second, different nodes may be equipped with different numbers and types of GPUs. Hence, jobs can experience both intra- and inter-node heterogeneity. In these environments, statically assigning jobs to GPUs in a dedicated fashion may lead to sub-optimal mapping of work-to-resources, thus limiting performance. In addition, in our previous work [2] we have shown that, in the presence of distributed applications, heterogeneity may cause imbalances among processes, which in turn lead to resource underutilization and sub-optimal performance.

Second, traditional schedulers treat GPUs as dedicated resources, rather than sharing them among users and applications. On the other hand, GPU sharing has proven to be an effective mechanism to increase both resource utilization and overall performance [1, 2, 34]. The potential of GPU sharing is also motivated by the increasing compute power of recent devices (the number of GPU cores, for example, has increased by a 6x factor from the Fermi to the Kepler architecture), while many GPU-accelerated applications have been optimized on less sophisticated devices. While previous work on GPU sharing has focused on node-level considerations, the integration of these results with cluster-level issues has received only limited attention.

Third, traditional schedulers provide limited GPU virtualization and expose information about the underlying cluster architecture to users. In many cases, users must decide the runtime configuration of their application depending on the available computing resources. In shared environments, however, the load of the cluster affects the optimal configuration. Some cluster resource managers require detailed profiling information to perform good scheduling decisions. In general, it is preferable for cluster resource managers to obtain profiling information dynamically and operate with limited user intervention.

Finally, traditional schedulers often do not provide easy ways to configure their scheduling policies. It is well known that the optimal scheduling policy depends on the performance objectives, the cluster's hardware and software configuration and the applications' characteristics. Administrators should be able to define custom scheduling policies that satisfy the users' needs and take the cluster configuration and the workload profile into consideration. Providing administrators with easy-to-user customization

facilities without requiring them to fully understand the architecture of the cluster-level scheduler is especially important in heterogeneous environments.

In this work, we make the following contributions.

- We design a new cluster-level scheduler and integrate it with our previously proposed node-level GPU virtualization runtime. This hierarchical framework allows better utilization of heterogeneous CPU-GPU clusters.
- Our proposed scheduling framework is configurable and allows administrators to easily define custom scheduling policies, so to meet specific requirements dependent on the cluster setup, the applications, and the user objectives.
- We use our scheduling API to define two application- and heterogeneity-aware scheduling policies: co-location- and latency-reduction based scheduling. We show that these schemes outperform batch scheduling on different applications and cluster configurations.

4.2 Cluster-level Scheduler

Our overall goal is to provide an efficient scheduling framework for CPU-GPU clusters. These heterogeneous environments require scheduling at two granularities: on one hand, jobs must be mapped onto compute nodes (coarse-grained scheduling); on the other, specific library calls must be mapped onto GPUs (fine-grained scheduling). In order to allow scalability, we design a hierarchical system, consisting of a cluster-level scheduler and multiple node-level runtime components installed on each compute node. The cluster-level scheduler is responsible for coarse-grained mapping of processes onto compute nodes, whereas the node-level components perform fine-grained scheduling of

GPU work onto the available devices. In our previous work [1, 2, 34] we have proposed a node-level runtime component that supports different GPU sharing and scheduling mechanisms. The main features and architectural characteristics of this node-level component are summarized in CHAPTER 3. In this section and the next, we discuss the features, architecture and interfaces of our newly designed cluster-level scheduler.

Our cluster-level scheduler is designed to closely interact with the node-level runtime component in order to schedule single- and multi-process GPU applications on heterogeneous clusters. The main goals of the cluster-level scheduler are: (i) to perform cluster-level scheduling decisions, (ii) to coordinate with the node-level runtime components, and (iii) to allow administrators to define customized scheduling policies. User-defined scheduling policies can have different optimization goals, such as improving the system throughput, the power consumption, or other Quality of Service metrics. In Section 4.3, we present two sample scheduling policies.

Our cluster-level scheduler employs a centralized model: a head-node accepts jobs from users and performs cluster-level scheduling decisions in a centralized fashion. The head-node, then, schedules jobs and processes onto different compute-nodes. A similar model is adopted by existing cluster resource managers such as TORQUE and SLURM. Differently from these systems, however, our scheduler is designed to explicitly target heterogeneous CPU-GPU clusters and to provide administrators with the ability to easily define and customize the scheduling policies. In order to better utilize the resources of heterogeneous clusters and allow scalability, our overall system performs scheduling in a hierarchical fashion and clearly separates cluster- and node-level scheduling responsibilities, which involve processes-to-nodes and GPU work-to-device mapping

decisions, respectively. An interface between the cluster- and node-level components allows the cluster-level scheduler to query node-level information such as the number of CPUs and GPUs available on the node, and its level of load (for example, the number of pending jobs). However, fine-grained control information (such as the load of specific GPUs) is not exposed to the cluster-level scheduler, but is used only for node-level decisions. This decoupled design allows scalability and easier reconfiguration and load balancing in case of dynamic changes in the cluster setup (e.g. upgrade and downgrade of nodes and GPU devices). Administrators can use the scheduling API described below to define their custom cluster-level scheduling policies, which are loaded in the scheduler as software modules. The scheduling API provides also routines to configure the GPU sharing and scheduling scheme used at the node-level. The cluster-level scheduling policies described in this paper rely on round-robin scheduling with preemptive sharing [2] to be performed at the node-level (however, they could also be configured to use different node-level policies).

The cluster-level scheduler running on the head-node services requests from users and responds to notifications from the runtime components running on the compute nodes. Users can submit to the head-node requests such as job execution, job status query, node status query, or job priority change. Notifications received from node-level runtime components include job-completion, performance-related information, node status, and attributes changes. Such information is also delivered to the administrator-defined scheduling library. The scheduling library can ignore these notifications or take appropriate actions based on the policy programmed by the administrator. For example, if the scheduling library observes that sharing GPUs on a node is degrading performance of

a job, it can immediately suspend a co-located job and reschedule it onto a different node. Thus, the feedback information from the node-level to the cluster-level scheduler allows cooperation between these two components.

4.2.1 Scheduler Architecture

Figure 4-1 shows the overall architecture of our cluster-level scheduler. The lower layer of the framework implements the communication mechanisms required by the cluster-level scheduler to interact with the users and the node-level runtime components installed on the compute nodes. This interaction can take place using a variety of communication channels: currently we provide socket communication for inter- and intra-node communication, and shared memory communication for intra-node communication; in the future, we plan to add an Infiniband communication module. The implementation of the different communication channels is abstracted by a communication abstraction layer, which provides a uniform view of the communication channels to the modules in the upper layers. On top of the communication layer, the architecture includes some scheduling-specific and some general purpose components. Specifically, the common modules provide protocols for data exchange between components, routines for low-level data structure manipulation, and other helper functions. The scheduling components include a scheduling API, (user-defined) shared libraries implementing the scheduling policies, and data structures storing job- and cluster -related information. The scheduling API is divided in two parts: routines that allow users to query information (such as the status of jobs and nodes), and routines exposed to the administrator to program the scheduling libraries.

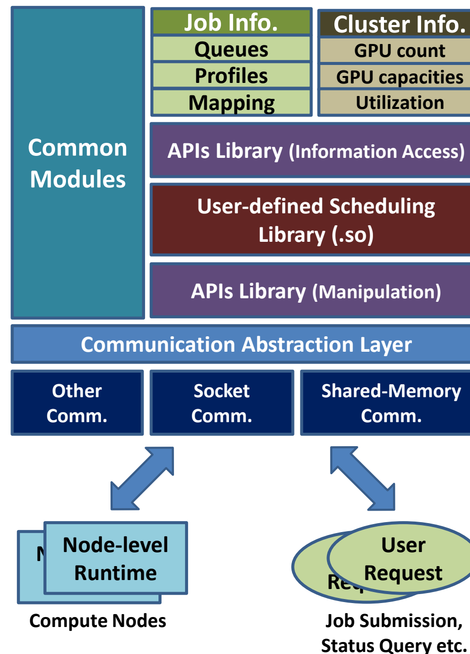


Figure 4-1 – Architecture of the cluster-level scheduler

As mentioned above, one of our design goals is to make our cluster-level scheduler easily configurable, and provide the users with an easy mechanism to define custom scheduling policies. The scheduling API, by providing routines to access and manipulate job- and node-specific information and local data structures (such as job queues), allows administrators to easily define new scheduling policies (or modify existing ones) without the need to know the implementation details of the cluster-level scheduler and node-level runtime components. User-defined scheduling policies are implemented as dynamic-linked libraries (or shared objects). Administrators can switch among policies by editing the configuration file of the scheduler to use the desired library. We provide two reference scheduling policies, which we describe in Section 4.3.

It is worth comparing our system with existing cluster resource managers, such as TORQUE. When using TORQUE, administrators can switch among policies by replacing

scheduling components like *pbs_sched*[2] or *Maui*[3] with their own scheduling modules. In this sense, our scheduling libraries are comparable to *pbs_sched* and *Maui*. However, constructing such scheduling components is not a trivial task and requires a good understanding of TORQUE's architecture. In particular, TORQUE consists of different components that run as daemons, interact through sockets, and do not share data structures. Differently from TORQUE, we aim to abstract the architecture of the scheduler and provide easy access to its data structures. In addition, TORQUE was not originally designed to handle GPU-jobs: TORQUE's design focuses on CPU resources and regards each GPU as an additional resource which is dedicated to a job and cannot be shared. Our proposed system allows GPU sharing and can be customized without great effort.

4.2.2 Scheduling API

Table 4-1 lists the main primitives of the scheduling API that must be used by administrators to create custom scheduling libraries. In particular, the scheduling interface provides two mechanisms: API and callback functions.

---

[2] http://linux.die.net/man/8/pbs_sched

[3] http://docs.adaptivecomputing.com/maui

API Functions	Description
General	
mccsSetCallBack()	Registers callback function for a given event.
mccsReport()	Prints message to *stderr*, *stdout* or log file.
mccsClean()	Releases all resources allocated to the scheduling library.
Node-level Runtime Management	
mccsGetNodeList()	Retrieves the list of compute nodes
mccsGetNode()	Retrieves a node data object based on name or attributes.
mccsPingNode()	Sends ping signal to a node to determine its availability.
mccsQueryNodeInfo()	Retrieves node configuration information such as: # of CPUs/GPUs, memory capacity.
mccsCreateNode()	Creates a data object used to represent a node.
mccsAddNode()	Adds a node data object to the list of compute nodes.
mccsCreateAccelerator()	Creates a data object used to represent an accelerator (such as a GPU).
mccsAddAccelerator()	Binds an accelerator data object to a node data object.
mccsSetNodeSchedulingP()	Sets the node-level scheduling policy
mccsSetNodeSharingP()	Sets the node-level GPU sharing policy
mccsSetNodeVGPUConf()	Sets the virtual GPU configuration at a node
mccsQueryNodeQueues()	Queries the size of the job queues at a node
mccsConfigureNodes()	Prepares all nodes that will participate in the computation of a job.
Job and Queue Management	
mccsAddJob()	Add a job to a job queue. The job queue can be either the *unassigned-job-queue* or the *assigned-job-queue*, which store the new and running jobs, respectively.
mccsRemoveJob()	Retrieves and removes a job from a queue.
mccsRetrieveJob()	Retrieves a job from a queue (without removing it).
mccsSetProfilingPeriod()	Sets the time period for a node to report back profiling information on jobs.
mccsAddNodeToJob()	Assigns a given number of processes belonging to a specified job to a node
mccsRemoveAllNodesFromJob()	Remove all nodes that were previously assigned to a job
mccsSetDeviceLimit()	Sets resource limit in term of #of GPUs. Setting the limit to zero is equivalent to suspending the job on the node. This can be called to dynamically set the resource limit for a job while running.
mccsStartJob()	Starts a job on the specified nodes.
mccsDiscardJob()	Discards a job data object and releases the resources assigned to it.

Table 4-1 – Scheduling API

API Functions: The API functions (see Table 4-1) allow performing a variety of tasks, such us querying the status and capacity of the nodes, allocating nodes to jobs, starting the execution of jobs, gathering profiling information, and others. In the current implementation, the API functions fall into three categories: *General API*, *Node-Level Runtime Management API*, and *Job & Queue Management API*. General API functions allow the scheduling library to perform house-keeping tasks such as registering callback functions, printing log messages, and releasing resources. The Node-level Management API functions control the interaction with the node-level runtime components. In particular, this API includes node-level query and management functions, and allows administrators to configure the GPU sharing and scheduling policies used at the node-level, as well as other aspects of the node-level runtime (e.g., virtual GPUs – see Section IV). Finally, the Job & Queue Management API functions implement tasks required for scheduling purposes, such as assigning jobs to queues and nodes, limiting the resources available to specific jobs, collecting profiling information on running jobs, and scheduling jobs to execute on specific nodes. Note that using these API functions to access data structures within the cluster-level scheduler (such as job queues and node lists) avoids race conditions and improper data accesses.

```
1:  Scheduler_Main() {
2:    //Register callback functions
3:    mccsSetCallback(Termination, Term_CB());
4:    mccsSetCallback(NewJobArrival, NewJob_CB());
5:    mccsSetCallback(JobComplete, JobComp_CB());
6:
7:    //Retrieve list of compute nodes
8:    NodeList nodeList;
9:    mccsGetNodeList(&nodeList);
10:
11:   while (SchedulerActive) {
12:     if (UnassignedJobQueue.empty)
13:         wait for signal
14:     //Fetch a job from unassigned-job queue
15:     Job job;
16:     mccsRemoveJob(&job, UnassignedJobQueue);
17:     //Perform round-robin node assignment
18:     do {
19:       Node node;
20:       foreach node in the nodeList
21:         mccsAddNodeToJob(job, node, 1);
22:     } while (remainingProcesses(job) > 0);
23:     mccsAddJob(job, AssignedJobQueue);
24:     //Configure nodes and start the job
25:     mccsConfigureNodes(job);
26:     mccsStartJob(job);
27: } }
28:
29: //Callback functions
30: Term_CB(void *arg) { SchedulerActive = false }[1, 2]
31:
32: NewJob_CB(void *arg) {
33:   Enqueue the job in UnassignedJobQueue
34:   and send a notification signal }
35:
36: JobComp_CB(void *arg) {
37:   Notify user about job completion }
```

Listing 4-1 – Simplified implementation of a round-robin scheduling library

Callback Events and Functions: The scheduler can interact with the scheduling library by notifying the occurrence of events. These events can be originated both at the cluster- and at the node-level, and include: job submissions, job priority change requests from users, job completion notifications from compute nodes, profile information receipt from compute nodes, and termination requests. User-defined callback functions allow the scheduling library to react to these events. These callback functions must be registered and associated to events by invoking the *mcssSetCallback* primitive.

Listing 4-1 illustrates the use of the described API on a round-robin scheduling library. The library first registers the required callback functions (lines 2-5), that are implemented at lines 29-37. Then, it retrieves the list of nodes available for scheduling (lines 7-9). The core of the scheduling process is implemented at lines 11-27. Jobs to be scheduled are extracted from the UnassignedJobsQueue (line 12) and moved to the *AssignedJobQueue* (line 23) after consideration. The *mccsAddNodeToJob* function assigns jobs to nodes one process at a time (3rd parameter of the function call) in a round-robin fashion (lines 19-21). The *mccsConfigureNodes* and *mccsStartJob* functions (lines 25-26) trigger the execution of the scheduled processes on the identified nodes.

4.3 Scheduling Policies

In this section we present two scheduling policies for heterogeneous CPU-GPU clusters, namely: *co-location-based* and *latency-reduction-based* scheduling. The former aims to maximize the overall throughput by minimizing the communication latency and enabling GPU sharing among processes; the latter seeks to reduce the application latency observed by users. These two policies are implemented in separate scheduling libraries using the API described above.

4.3.1 Co-locating Scheduler

Modern high-performance computing clusters tend to be equipped with multiple GPUs per node. In the presence of multi-core nodes with multiple GPUs, communication costs can be reduced by scheduling multiple processes from the same application onto the same node. Especially in the presence of communication-intensive applications, co-locating application processes on a node can significantly increase the performance by reducing the communication costs (since intra-node communication is significantly faster than inter-node communication). By allocating one GPU to each process requiring it, traditional cluster resource managers such as TORQUE can co-locate only as many processes as GPUs available on a node. By leveraging the GPU sharing mechanisms provided by our node-level runtime (see Section IV), our cluster-level scheduler can overcome this limitation, and co-locate a larger number of processes from the same application. Although GPU sharing may degrade the performance of individual processes, it has the potential for increasing the node utilization and the overall performance, and decreasing the jobs queuing time.

To provide an idea of the effect of process co-location, we run three applications, each with 8 processes, on a 3-node cluster. We tested several process-to-node mappings (using from 1 to 3 nodes per experiments). Each node contains 2-4 GPUs: the details about the applications and the cluster setup are provided in Section 4.4. The applications were run separately from one another. Figure 4-2 shows the execution time with different mapping strategies. The number of processes mapped onto each node is indicated in square brackets. MM and NW are communication intensive, whereas ADFA is a compute intensive application. As can be seen, the communication-intensive applications fail to

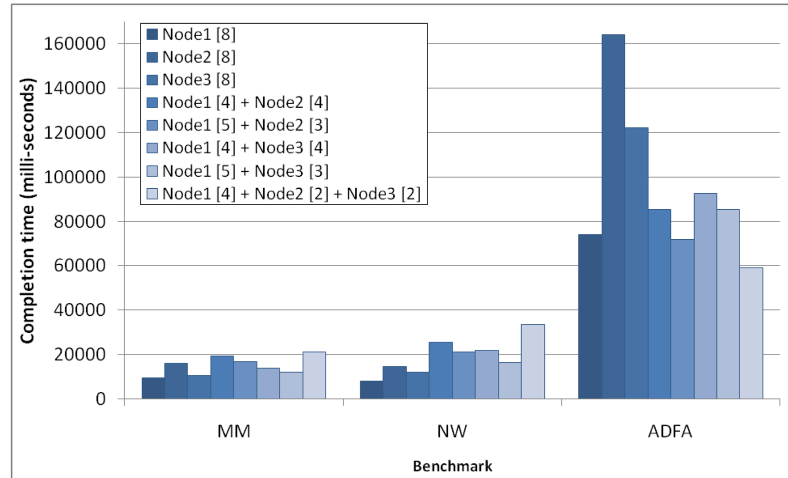


Figure 4-2 – Performance of three benchmark applications running with 8 processes on different node configurations (from 1 to 3 nodes). $\text{Node}_i[j]$ indicates that *j* processes are run on Node_i.

scale to multiple nodes. On the other hand, being compute intensive, ADFA performance scales when the application is assigned multiple nodes (and, as a consequence, more GPUs). For configurations using the same number of nodes, the performance depends on the process-to-node assignment, since the nodes differ both in the number and the compute capability of the GPUs (see Table 4-3 for details). In general, more powerful nodes are expected to handle more processes. To summarize, the optimal scheduling policy depends both on the nature of the application (communication vs. compute intensive) and on the compute capability of the nodes.

We make our co-location-based scheduling policy both *application-* and *heterogeneity-aware*. The communication vs. computation characteristic of the application can be provided either by static or by dynamic profiling (in the latter case, profiling data are collected at runtime on a few iterations of the application). In our experiments, we assume that this profiling information is available a priori, and does not

need to be measured at runtime. In order to handle heterogeneity, we assign different weights to the compute nodes according to their compute capability (e.g., the overall number of available GPUs or GPU cores) and sort the nodes based on these scores. During scheduling, we prioritize more powerful nodes.

Listing 4-2 shows a simplified pseudo-code of the co-location-based scheduling library, where co-location decisions depend on the application name, and co-located processes of an application are all scheduled onto a single node. In our implementation, we allow a less strict notion of co-location, whereby the number of nodes assigned to an application is bounded, but not necessarily equal to 1 (see Section 4.4).

```
1:  Scheduler_Main() {
2:   //Initialization
3:   Setup node weights and sort nodes
4:   mccsSetCallback(…);
5:
6:   while(SchedulerActive) {
7:     if (UnassignedJobQueue.empty)
8:       wait for signal
9:     Node node; Job job;
10:
11:    //Retrieve a job from unassigned job queue
12:    mccsRemoveJob(&job, UnassignedJobQueue);
13:
14:    //Determine whether to co-locate all
15:    bool colocate = (job.name = "ADFA")? F:T;
16:
17:    int totalGpu = 0;
18:    if (colocate) {
19:      Node selected;
20:      //Find the node with maximum score
21:      foreach node in nodeList {
22:        int nAvailGpu = NodeWeight(node) –
23:                 NumAssignedProcesses(node);
24:        if (nAvailGpu > totalGpu){
25:          selected=node; totalGpu=nAvailGpu;
26:      } }
27:      //If GPUs are available, assign job
28:      if (totalGpu > 0)
29:        mccsAddNodeToJob(job, selected,
30:                         job.numProcesses);
31:    } else {
32:      //Find available GPUs on all nodes
33:      foreach node in nodeList {
34:        int nAvailGpu = NodeWeight(node) -
35:                 NumAssignedProcesses(node);
36:        totalGpu += nAvailGpu;
37:        if (nAvailGpu <= 0) continue;
38:        mccsAddNodeToJob(job,node, nAvailGpu);
39:      }
40:      //let remaining processes share GPUs
41:      if (remainingProcess(job) > 0) {
42:        Assign processes to nodes round-robin
43:    } }
44:    //if cannot allocate resources for the
45:    //job, cancel the previous assignment.
46:    if (totalGpu == 0) {
47:       mccsRemoveAllNodesFromJob(job);
48:       mccsAddJob(job, UnassignedQueue)
49:    } else {
50:        mccsAddNode(job, AssignedQueue);
51: }} }
```

Listing 4-2 – Simplified implementation of the co-location-based scheduling library

4.3.2 Latency-reducing Scheduler

The described co-location-based scheduling policy, by leveraging GPU sharing, improves the overall throughput of the cluster at the cost of increased execution time of individual applications. As can be observed in Figure 4-2, the considered applications have different running times: specifically, MM and NW are short-running jobs, while ADFA is long-running. In the experiment in Figure 4-2, these applications are run in isolation. However, when different jobs compete for resources, long-running jobs can severely lower the performance observed by short-running jobs. Specifically, treating short- and long-running jobs the same way may create unfairness in resource utilization and unacceptable Quality-of-Service to the users submitting short-running applications. Here, we define Quality-of-Service as the ratio between the execution time on a shared environment and the execution time on a dedicated cluster. Based on this definition, users submitting short-running applications expect their jobs to complete in a short time.

We extend the co-locating scheduler to also reduce the overall latency observed by the users, and we refer to the new policy as *latency-reducing scheduler*. We assume the presence of profiling information that allows distinguishing between short- and long-running jobs depending on the nature of the application and the size of the input dataset. The latency-reducing scheduler prioritizes short-running jobs in two ways: (i) by appropriately sorting the queue of pending jobs, and (ii) by temporarily suspending long-running jobs when all GPUs are occupied and short-running jobs need to be scheduled.

Listing 4-3 shows the modifications performed to the co-locating scheduler to implement latency-reduction. In particular, the modified portion of the code starts at line

```
44:     //if cannot allocate resources for the
45:     //job, cancel the previous assignment,
46:     if (totalGpu == 0) {
47:       mccsRemoveAllNodesFromJob(job);
48:       //prioritize short-running jobs
49:       if (job.short_running) {
50:         mccsAddJobToHead(job,UnassignedQueue);
51:
52:         //Search long-running job to suspend
53:         foreach job in AssignedQueue {
54:           if (job.long_running)
55:             mccsSetDeviceLimit(job, 0);
56:         }
57:       // hold long-running jobs
58:       } else
59:         mccsAddJobToTail(job,UnassignedQueue);
60:     } else {
61:       mccsAddNode(job, AssignedQueue);
62: }} }
```

Listing 4-3 – Extensions to the co-location-based scheduling library to implement the latency-reduction policy

44. The newly added lines, which prioritize short-running jobs and suspend long-running ones when all GPUs are occupied, are shaded in grey. Note that fairness could be improved also by limiting the resources assigned to long running jobs (rather than completely suspending them). This can be achieved by modifying the second parameter of the *mccsSetDeviceLimit* call at line 55. Finally, note that the temporary suspension (or reassignment to fewer GPUs) of long-running jobs is made possible by the preemption mechanism implemented within our node-level runtime component (see Section 4.4).

	MM	NW	ADFA
Number of Processes	Configurable	Configurable	Configurable
Communication Type	MPI_Scatter, MPI_Gather,	MPI_Scatter, MPI_Gather	MPI_Gather,
Sensitivity	Communication sensitive	Communication sensitive	Computation sensitive
Size of MPI Transfer per iteration	800 MB	100-200 MB	8 MB
Number of iterations	2-8	15-30	1-2

Table 4-2 – Summary of benchmark characteristics

4.4 Experimental Results

4.4.1 Benchmark Applications

We designed a hybrid MPI-CUDA benchmark that consists of three applications: Matrix Multiplication *(MM)*, Needleman-Wunsch (*NW*), and the *ADFA* compression algorithm [66]. Our benchmark applications use MPI to distribute work among nodes and CUDA to offload computation to GPUs. These applications alternate inter-node communication and CPU-GPU computation phases (including data transfers between CPU and GPU). They mainly use scatter and gather as communication primitives. Below, we provide more detail on the three applications. Table 4-2 shows a summary of their main characteristics. In all cases, the duration of the GPU computation phases depends on the input dataset.

Matrix Multiplication (MM): Matrix multiplication is an iterative application that multiplies $N \times I$ pairs of matrices using P processes over I iterations. At the beginning of each iteration, N matrices are equally distributed among the P available

processes using the `MPI_Scatter` primitive. Each process is executed on a single GPU. For every iteration, each process transfers *N*/*P* pairs of matrices from CPU to GPU, performs the matrix multiplications on GPU and copies the product matrices from GPU to CPU. At the end of each iteration, the computed matrices are transferred to the root process using the `MPI_Gather` primitive, which also provides an implicit barrier synchronization among processes. The size of the matrices can be configured by the user. In our experiments, we run MM with 48-196 pairs of matrices, each of size 1,600 x 1,600. This results in about 800 MB of MPI data communication per iteration. The overall execution time is dominated by this inter-process communication, making the MM application communication sensitive. We run a number of iterations varying from 2 to 8 depending on the experiment.

Needleman-Wunsch (NW): This application performs the all-to-all pairwise alignments of large datasets of biological sequences using the Needleman Wunsch algorithm [67, 68]. We use MPI to distribute the input sequences to all available processes and CUDA to compute each alignment matrix. Like matrix multiplication, Needleman-Wunsch is iterative and computes $N \times I$ pairwise comparisons of sequences of length L over I iterations. At the beginning of each iteration, N sequence pairs are equally distributed among P available processes using the `MPI_Scatter` primitive. For every iteration, each process transfers *N*/*P* pairs of sequences from CPU to GPU, performs sequence alignment on the GPU, and copies the alignment matrices from GPU to CPU. At the end of each iteration, the alignment matrices are transferred to the root process using the `MPI_Gather` primitive. The length of the sequences and number of

iterations can be configured by the user. In our experiments, we use sequences varying from 500-1,000 bases in length, and a number of sequences varying from 2,400 to 7,200. This results in about 100-300 MB of MPI data communication per iteration. Similar to matrix multiplication, the inter-process communication dominates the computation time making NW communication sensitive.

Amortized time−bandwidth DFAs (ADFA): ADFA is a technique to compress Deterministic Finite Automata that accept large sets of regular expressions [66]. In our hybrid MPI-CUDA ADFA implementation, we maintain a work queue containing *N* input DFA files. At each iteration, *P* DFAs are removed from the work queue and each of them is assigned to a different MPI process for ADFA compression. At the end of each iteration, an `MPI_Gather` primitive transfers the result of ADFA compression into a single array to the root process. The application terminates once all DFAs from the work queue have been processed. There is an implicit work imbalance since the execution time of each process varies depending on the type of GPU on which it is executed. Some processes might lie idle at the gather stage until other processes reach that stage. The inter-process communication is limited: differently from MM and NW, ADFA is compute intensive. In our experiments, we use DFAs with 30,000 states and a number of such inputs varying from 8 to 16.

Attributes	Type-I	Type-II
CPU		
Type	Intel Core 2 Quad Q9400 @ 2.66 GHz	Quad-core Intel ® Xeon E5 @ 2.80 GHz
Memory	4 GB	4 GB
GPU		
Model	Nvidia Quadro 2000	Nvidia Geforce GTX 460
# SMs / # cores	4/192 @ 1.2 GHz	7/336 @ 1.4 GHz
Memory	1 GB @ 1.3 GHz	1 GB @ 1.8 GHz
# nodes	**5**	**3**

Table 4-4 – Commodity cluster setup

Node	Attributes	Values
Node$_1$	Type	8-core Intel Xeon E5 @ 2.4 GHz, 12 MB Cache
	Memory	48 GB
	# GPUs	4
	GPUs	Nvidia GeForce GTX 480 (Fermi) 15 SMs x 32 cores 1 GB Global Memory
Node$_2$	Type	12-core Intel Xeon E5 @ 2.00 GHz, 15 MB Cache
	Memory	64 GB
	# GPUs	3
	GPUs	Nvidia Tesla K40c (Kepler) 15 SMs x 192 cores @ 876 MHz 12 GB Global Memory @ 3004 MHz
		Nvidia Tesla C2075 (Fermi) 14 SMs x 32 cores @ 1147 MHz 5.6 GB Global Memory @ 1566 MHz
		Nvidia Tesla C2070 (Fermi) 14 SMs x 32 cores @ 1147 MHz 5.6 GB Global Memory @ 1494 MHz
Node$_3$	Type	12-core Intel Xeon E5 @ 2.10 GHz, 15 MB Cache
	Memory	64 GB
	# GPUs	2
	GPUs	Nvidia Tesla K20c (Kepler) 13 SMs x 192 cores @ 706 MHz 5 GB Global Memory @ 2600 MHz
		Tesla C2050 (Fermi) 14 SMs x 32 cores @ 1147 MHz 2.8 GB Global Memory @ 1500 MHz

Table 4-3 – High-end cluster setup

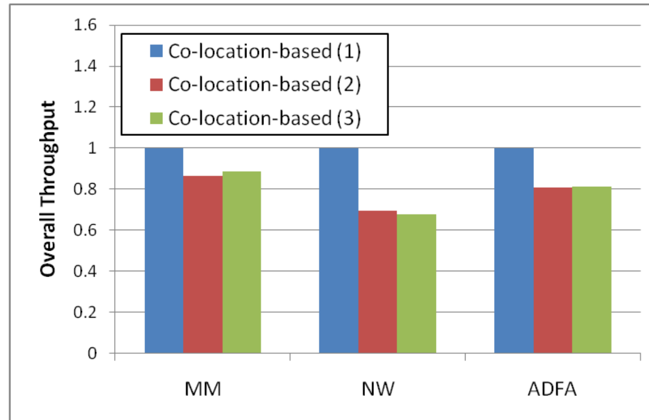
4.4.2 Experimental Setup

We conducted experiments on two clusters: a commodity cluster consisting of eight workstations, each equipped with a quad-core CPU and a low-end Fermi GPU, and a high-end cluster comprising three HPC servers, each equipped with a hyper-threaded 8- or 12-core CPU and 2-to-4 Nvidia GPUs. Table 4-4 andTable 4-3 detail the setup of these clusters. As can be seen, the two clusters are heterogeneous both in terms of CPUs and GPUs. The nodes in the clusters are interconnected through Gigabit Ethernet. The operating system used is Ubuntu 12.04 on the commodity cluster, and CentOS 5/6 on the high-end cluster. CUDA 5.5 is installed on all machines.

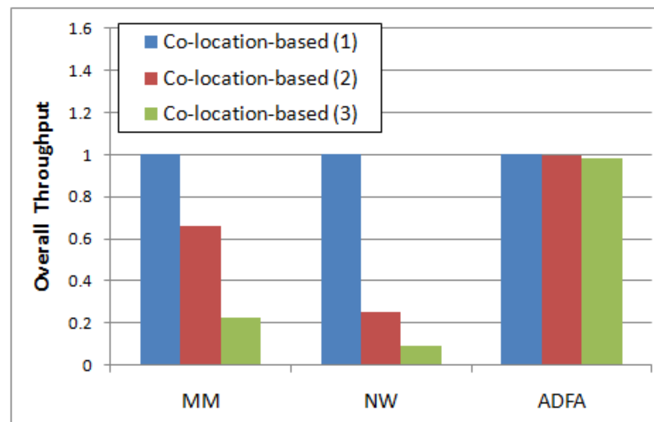
4.4.3 Analysis of Co-location

Figure 4-2 motivates the advantages of co-locating processes belonging to the same job on nodes equipped with multiple GPUs. We now study the effect of co-location on both high-end and commodity clusters. In particular, we measured overall throughput of different co-location-based policies as normalized overall execution time. In Figure 4-3, we show results reported by bounding the number of nodes used by each job to 1, 2 and 3. Each job spawns 4-to-8 processes. The results are normalized with the 1-node co-location policy. Note that, in 1- and 2-node co-location, nodes are selected in round-robin fashion based on GPU availability.

The results show that, independent of the number of GPUs installed on each node, it is always better to co-locate the processes of communication-sensitive applications (*MM* and *NW*) on a single node. On the other hand, the performance of the compute intensive ADFA application is not significantly affected by the degree of co-location,



(a) High-end cluster



(b) Commodity cluster

Figure 4-3 – Co-location-based scheduler with relaxed constraints (1, 2 and 3 nodes/job allowed)

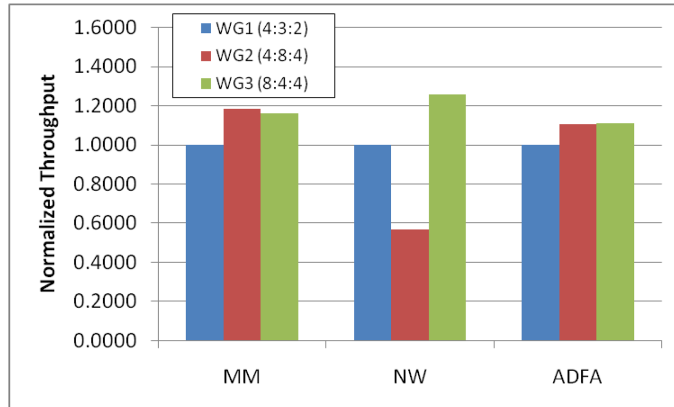
especially on the commodity cluster. However, we observed that the communication cost of ADFA dominates the computation cost in the high-end cluster whereat ADFA turns to be communication-intensive application. In all cases, GPU sharing helps the performance by increasing the GPU utilization.

4.4.4 Determining the Weights for the Nodes

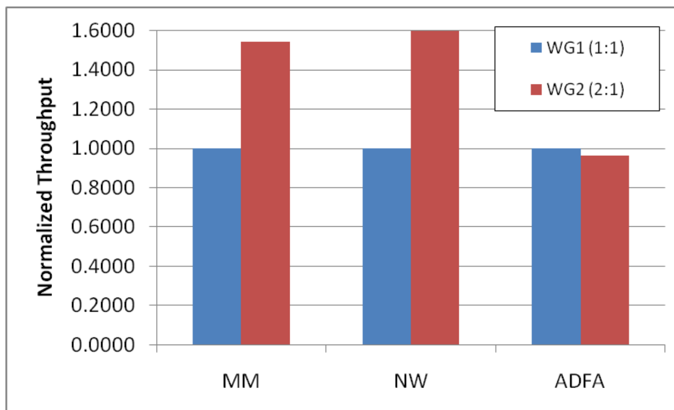
The different compute capabilities of the GPUs within each cluster create performance heterogeneity. As explained in Section 4.3, our scheduling policies use weights

associated to each node to capture this heterogeneity. We perform an experiment (Figure 4-4) to determine the best weight assignment for the nodes in the considered clusters. We will then use the determined weight assignment in the remaining experiments. This experiment is run on 6 instances of each application, each running 4-8 processes.

In the first weight assignment, WG_1, the weights associated to the nodes corresponds to the number of GPUs on them. On the high-end cluster, for example, WG_1 = ([$Node_1$:$Node_2$:$Node_3$] = [4:3:2]). We expect this scoring scheme to be sub-optimal since most of the time faster GPUs will be underutilized. We then perform experiments where we adjust the weights depending on the compute capabilities of the GPUs. The second set of weights WG_2, ([$Node_1$:$Node_2$:$Node_3$] = [4:8:4] on the high-end cluster), is based on the number of streaming multiprocessor and cores of the GPUs installed on each node. This set of weights yields to better results on all applications except *NW*. The third set of weights WG_3 ([$Node_1$:$Node_2$:$Node_3$] = [8:4:4] on the high-end cluster), which considers both the number of GPUs and their cores, yields to better results on all applications. The performance degradation for NW comes from the mismatch assignments of jobs to nodes resulting in slower nodes ($Node_2$ and $Node_3$) handling more work.



(a) High-end cluster

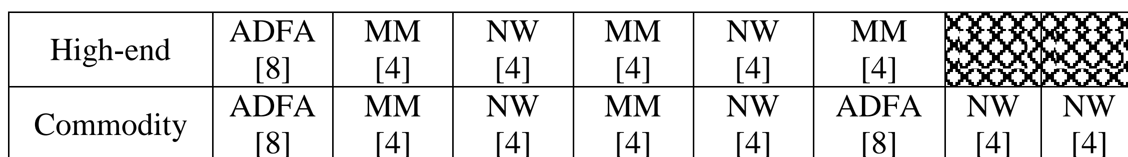


(b) Commodity cluster

Figure 4-4 – Throughput for different sets of weight assignments

4.4.5 Experiments on Heterogeneous Workloads

We now want to evaluate the performance of our scheduling schemes on mixed workloads. In particular, we run two sets of applications: 6 jobs on the high-end and 8 jobs on commodity cluster. The sequence of the job submission is shown in Table 4-5. As a baseline, we construct a batch-scheduling library that emulates the behavior of TORQUE: it assigns the processes of each job to GPUs in a round-robin fashion, without performing GPU sharing at the node level.

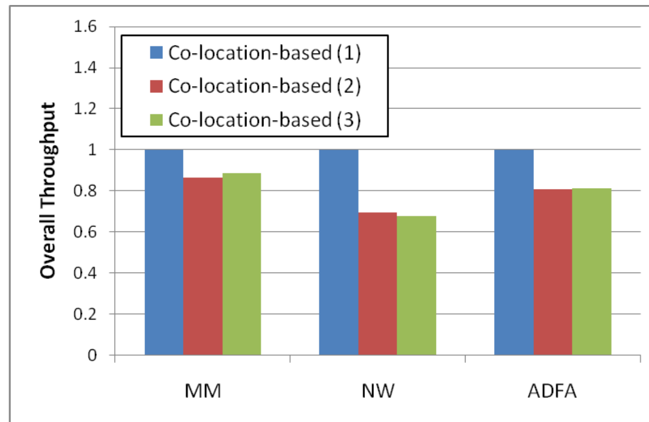


High-end	ADFA [8]	MM [4]	NW [4]	MM [4]	NW [4]	MM [4]		
Commodity	ADFA [8]	MM [4]	NW [4]	MM [4]	NW [4]	ADFA [8]	NW [4]	NW [4]

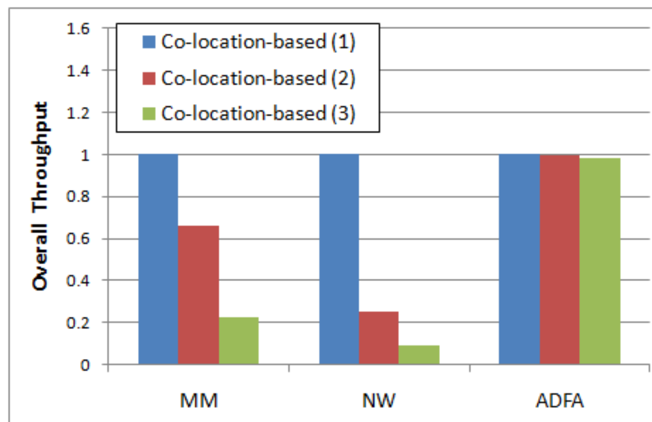
Table 4-5 – Sequence of jobs submitted (the number of processes spawned is indicated in square brackets)

Figure 4-5 shows the results reported on high-end and commodity clusters, respectively. The two charts report the normalized throughput for the whole workload, as well as the average Quality-of-Service (measured in terms of application execution time) experienced by long- and short-running jobs. Throughput results are normalized to the throughput of the batch scheduler whereas Quality-of-Service results are normalized to one when the applications run in dedicated environment. Recall that, according to the QoS definition in Section 4.3.2 the optimal QoS value is 1 (which indicates no performance degradation with respect to a dedicated environment). The results show that, when compared to batch scheduling, co-location- and latency-reduction-based scheduling improve the overall throughput by a factor 1.5x and 1.2x, respectively, on the high-end cluster, and by a factor 2.5x and 2.45x, respectively, on the commodity cluster. This is because that batch scheduler fails to consider the heterogeneity present in the clusters, distributes the work uniformly to all the GPUs, and underutilizes some of the devices. In addition, the batch scheduler is unfair: it provides good QoS to long-running jobs while penalizing short-running ones. By allowing GPU sharing between long- and short-running jobs and thus letting short-running jobs move ahead, the co-location-based scheduler improves the average QoS of short-running jobs, thus providing more fairness.

The latency-reduction-based scheduler further improves fairness by reducing the difference in average QoS experienced by long- and short-running applications.



(a) High-end cluster



(b) Commodity cluster

Figure 4-5 – Overall throughput and QoS for the heterogeneous workload summarized in Table 4-5.

CHAPTER 5

INTER-NODE VIRTUAL MEMORY PROGRAMMING MODEL

5.1 Objectives

As we have explained in the introduction, the programming of CPU-GPU cluster involves several issues.

First, hybrid high-performance computing clusters may present hardware heterogeneity at different granularities: besides consisting of nodes comprising both CPUs and GPUs, they can include nodes with different hardware configurations and GPUs with different compute capabilities even on a single node. The above programming models offer static mechanisms for mapping work to compute elements and require substantial effort from the programmer to implement adaptive scheduling mechanisms within the application. This limits the performance portability of GPU-accelerated applications distributed across clusters with different hardware configurations.

Second, the above programming models do not provide easy-to-use dynamic load balancing mechanisms. While load balancing schemes can be embedded in the application, they come at the cost of significant code complexity. In addition, inter-application load balancing requires the use of cluster resource managers such as TORQUE and SLURM. Unfortunately, these scheduling frameworks offer only basic

support for heterogeneity. In order to provide load balancing in heterogeneous settings, these tools require the integration with recently proposed GPU virtualization frameworks [1, 27, 30, 34, 69-71]; however, these frameworks are still at a prototype phase.

Finally, the above programming models expose disjoint memory addressing spaces to programmers. Since CPUs and GPUs have their own memory spaces, programmers usually need to explicitly manage data objects in different memory spaces. In addition, MPI assumes a private memory space on each process. Furthermore, these programming models do not support the concept of shared objects: data objects belong entirely to a task (or process) in a distributed application and cannot be shared among tasks. Not only does this assumption complicate programmability, but it can also lead to inefficiencies in terms of memory utilization. For example, multiple tasks executing on the same GPU will need to duplicate even read-only data objects.

In this work, we aim to overcome these limitations. To this end, we propose Inter-node Virtual Memory (IVM) - a programming framework for distributed applications that provides a shared memory abstraction across compute resources and offers support for dynamic load balancing within applications. Our framework presents the following distinctive features. First, it reconciles memory spaces and provides the view of a *single virtual memory space* across nodes and compute resources. Second, it provides a *unified view of compute resources* and treats CPUs and GPUs similarly. Third, it supports *one-sided communication primitives*, which simplify the design of memory consistency policies. Fourth, it supports *dynamic process creation* (DPC), a mechanism to facilitate dynamic load balancing. Finally, our framework is designed for integration with cluster-level scheduling systems that allow inter-application scheduling and load balancing.

We summarize our contributions as follows:

- We propose IVM - a novel programming framework which consists of a shared memory-based programming model and a runtime system. IVM increases the programmability of heterogeneous distributed applications and offers easy-to-use support for dynamic load balancing.
- We propose two self-tuning load balancing schemes based on our programming model and demonstrate their use.
- We evaluate our framework on four distributed applications with different characteristics. Besides increasing programmability, our framework improves the performance up to 1.74x over using MPI over OpenMPI.

5.2 Background and Motivations

5.2.1 Traditional Programming Models

Message Passing Interface (MPI) and Partitioned Global Address Space (PGAS) are the two most popular programming models for distributed applications. In this work, we use OpenMPI [72] and OpenSHMEM [73] as reference programming frameworks based on the MPI and PGAS standards, respectively. The execution of distributed applications under these frameworks is broken into multiple parallel processes, which alternate computation and communication phases. The main difference between MPI and PGAS lies in the type of communication primitives provided to allow information exchange. Specifically, MPI employs a two-sided communication paradigm: processes exchange information through primitives such as *send* and *receive,* which require intervention from both sending and receiving parties. PGAS adopts one-sided communication primitives

such as *put* and *get*. These primitives allow a process to write and read information from the address space of a target process without its intervention.

5.2.2 Load Balancing

Distributed applications running on heterogeneous clusters may experience two kinds of load imbalance: *intra-* and *inter-application* imbalance. The former may occur within an application when its processes experience performance heterogeneity or when the amount of work handled by different processes is not uniform. The latter is due to interference among concurrent applications competing for shared compute resources [70]. In both cases, load imbalance can cause resource underutilization and degraded application performance. In this work, we focus on addressing intra-application load imbalance.

On heterogeneous clusters, intra-application imbalance is more likely to occur in the presence of applications with frequent synchronization and inter-process communication. When synchronization or communication is required, processes executing on more powerful resources will need to wait for processes executing on less

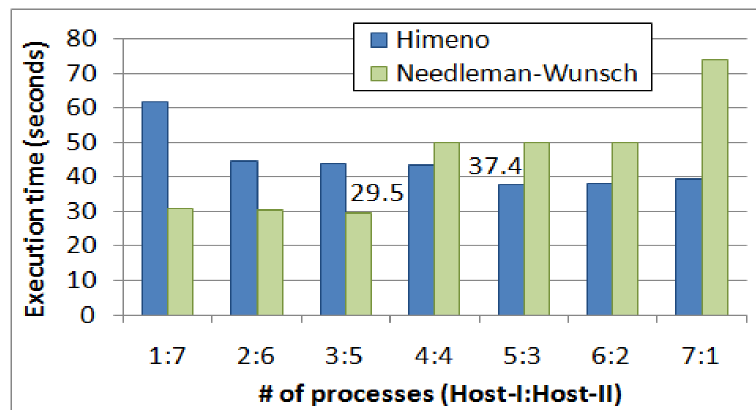


Figure 5-1 – Overall execution time of MPI-CUDA implementations of the Himeno and Needleman-Wunsch benchmarks under different process-to-node assignments. In all cases, eight processes are run on two heterogeneous nodes (Host-I and Host-II)

powerful ones. This load imbalance caused by performance heterogeneity arises from static assignment of tasks (or processes) to compute resources. In general, load balancing can be achieved either by dynamically assigning workloads to compute resources, or by statically assigning different amounts of work according to the performance of the compute resources.

MPI and PGAS do not provide a straightforward mechanism to dynamically assign workload to compute resources. The programmer must explicitly embed load balancing schemes within the application, which often comes at the cost of increased code complexity [74]. Alternatively, load balancing can be achieved by running basic MPI and PGAS applications on top of scheduling and virtualization frameworks for heterogeneous clusters. Recently, a mechanism to dynamically spawn processes has been introduced in the MPI-2 standard [75], thus providing a natural way to embed load balancing in the application. However, the use of this feature requires non-trivial coding effort: because parent and child processes belong to different MPI groups, the user is left with the burden of using MPI intra- and inter-communicators for inter-process communication. In addition to increasing the code complexity, the instantiation and handling of inter-communicators can add execution overhead to the application.

Statically assigning different amounts of work to distinct compute resources requires collecting profiling information on the underlying hardware. Figure 5-1 shows the overall execution time of an MPI-CUDA version of the Himeno and Needleman-Wunsch applications on two nodes with different GPU configurations (as detailed in Table 5-2). In all cases, the applications are configured to use eight processes. As can be seen, not only does the execution time depend on the mapping of the workload onto the

nodes, but the best assignment is also program-dependent: the optimal number of processes assigned to nodes Host-I/Host-II is 5/3 for Himeno and 3/5 for Needleman-Wunsch. Collecting profiling information can be an expensive operation, and must either be embedded in the application/scheduling framework (online profiling), or be repeated on each available hardware configuration (offline profiling). The lack of profiling information on some hardware configurations may cause performance portability issues.

5.2.3 The IVM Programming Framework

The overall goal of IVM is to provide a framework that simplifies the programming and the effective deployment of distributed applications on CPU-GPU clusters. To this end, our framework aims to provide *a uniform view of computing resources and memory*, and a simple *mechanism to embed load balancing within the application.* In addition, to allow inter-application load balancing, our framework must enable seamless integration with cluster-level schedulers (such as TORQUE and SLURM) and with GPU virtualization frameworks [1, 27, 30, 34, 69, 70, 74] for distributed CPU-GPU applications. IVM consists of a programming API and a runtime system.

IVM provides the programmer with a unified view of a shared virtual memory space across all available compute resources, and an API to access CPUs and GPUs uniformly. The uniform view of compute and memory resources is also crucial for supporting dynamic intra-application load balancing. Specifically, IVM supports dynamic process creation, which can be used to easily embed load balancing within the application. The dynamic spawning of processes allows runtime redistribution of the load. The shared virtual memory space simplifies communication, synchronization and

sharing of data objects between parent and child processes, and can reduce the overall memory footprint of the application.

5.3 IVM Framework Design

In this section, we describe our IVM framework in more detail. We start by discussing IVM's execution and memory models. Next, we present our system design, including IVM's software components. Finally, we illustrate IVM's programming interface.

5.3.1 Execution Model

We recall that the main goal of the IVM framework is to increase the programmability of applications running on distributed memory systems that include CPUs and GPUs. This is done by providing a uniform view of various memory spaces and compute resources. Specifically, IVM adopts a shared memory model with relaxed memory consistency; the programmer sees a single virtual memory space across all compute resources – CPUs and GPUs. Since all compute resources have access to data objects allocated in this virtual memory space, the programmer can access CPUs and GPUs in a unified way; in the IVM programming interface, all compute resources are represented by *device* instances.

Under the IVM framework, an application consists of multiple Processing Elements (PEs). In essence, a PE is a process that is bound to a specific compute resource. Figure 5-2 shows the physical view of a compute cluster consisting of heterogeneous nodes and including different physical memory spaces (to the bottom), and its abstract view (to the top). As can be seen, IVM provides the view of a virtual memory space containing shared data objects and accessed by all PEs.

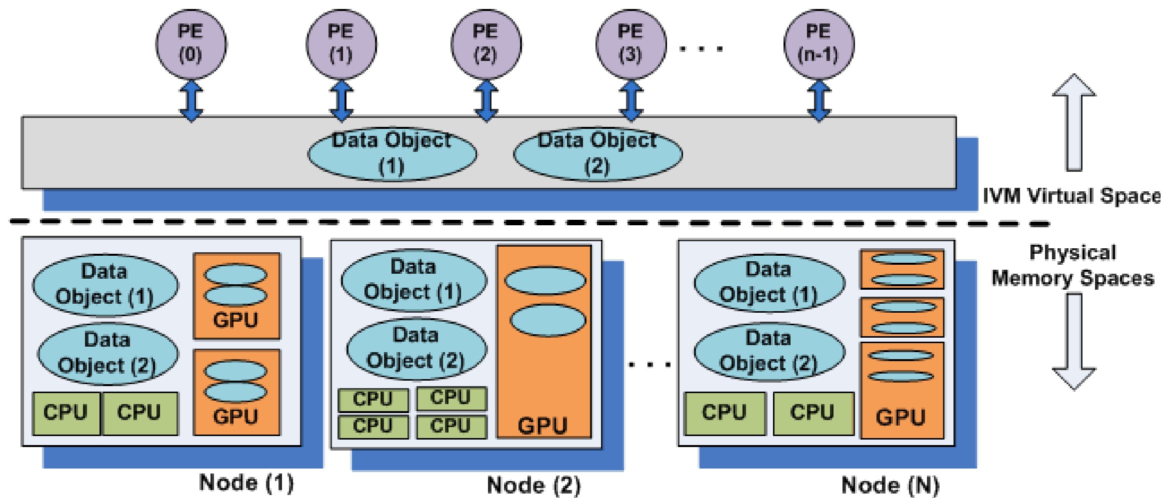


Figure 5-2 – IVM's Execution and Memory Models

Similarly to MPI, a distributed IVM-application is started by executing its binaries. Upon instantiation the IVM framework will create a single root-PE, which is the ancestor of all PEs belonging to the application. The root-PE can create its immediate child-PEs and distribute the work to them. It is possible for the child-PEs to further spawn additional PEs. In general, the framework allows a PE to spawn one or more PEs, which can bind to the same or different devices. The IVM framework also provides a mechanism to synchronize PEs. This mechanism consists of *wait* and *signal* primitives and is similar to POSIX-threads' conditional variables. For example, a PE can wait for its children by invoking the wait primitive and specifying the set of PEs to wait on; the child-PEs can then invoke the signal primitive to notify the waiting PE to continue. Programmers can also implement their own synchronization mechanisms using IVM's shared memory.

5.3.2 Memory Model

Figure 5-2 shows shared data objects accessed by PEs through the virtual memory space. A data object can be allocated by one and only one PE. Other PEs can gain access to the

data object through a mapping operation. Upon allocation, each data object is associated with an identifier that can be used by the IVM framework to reference that object. Different PEs can access a data object by providing its identifier to the IVM runtime through the programming interface. Note that data objects allocated in the virtual space do not belong to any specific PEs. All PEs will have the illusion that the data objects were floating singletons residing in the virtual space.

On the top part of Figure 5-2 the virtual memory space contains two data objects which can be accessed by the PEs. However, the framework needs to manage the physical data corresponding to each virtual data object. The IVM framework manages the data objects by transparently allocating memory for them on different physical spaces. As can be seen in the bottom part of Figure 5-2, there may be multiple physical copies of each (virtual) data object, and those may reside on different devices If a data object is accessed by multiple GPUs, each of these GPU will require a copy of the data object in its physical memory. Physical copies of data objects can be of two kinds: *master-copy* and *mirror-copy*. A master-copy is the copy of the data object that is initially allocated by the instantiating PE, whereas a mirror-copy is a copy that is created upon a mapping operation. Mirror-copies are created only if the allocating PE and the mapping PE are bound to different devices or physical memory spaces. If all PEs are associated to the same device, only the master-copy will be allocated. These physical copies are not exposed to the programmer, who sees only the data objects residing in the virtual space.

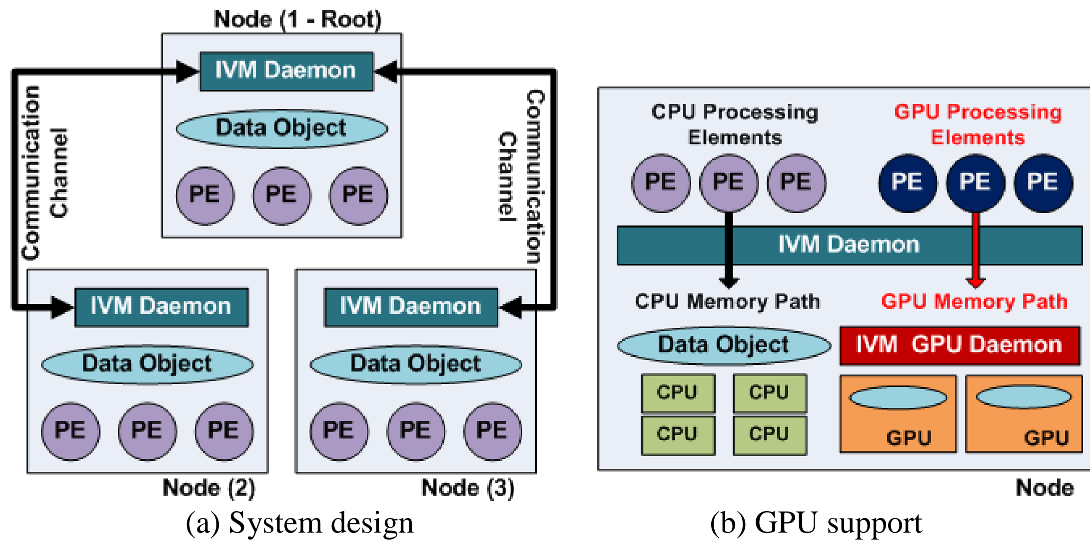
The IVM framework synchronizes the master- and mirror-copies of data objects through the *put* and *get* primitives. When a PE invokes a put/get operation on a data object, the entire or part of the master- and mirror-copies of the data object are

synchronized. The put/get primitives will have no effect for PEs that share the same physical copy. Beside simple put/get primitives, *gather* and *scatter* operations allow for a master-copy to synchronize with different mirror-copies and for a master-copy to distribute contents to different mirror-copies, respectively.

On a node, the framework allocates master- and mirror-copies within Linux's shared memory. Therefore, data consistency among PEs sharing the same copy is strict. This eliminates the need for duplicating data objects for PEs working on the same device or physical memory space and may reduce the memory footprint of the application. The allocation method on GPUs is different and will be described in 5.3.4.

5.3.3 System Design

The IVM framework consists of two main components: the runtime library and the runtime daemon. The IVM runtime library implements the programming API functions described in Section 0 and listed in Table 5-1. The runtime daemon runs on each compute node and serves API requests issued by applications. These requests include: PE registration/deregistration, memory allocation/de-allocation, memory put/get, PE creation and destruction.



(a) System design (b) GPU support

Figure 5-3 – System design and GPU support

Figure 5-3(a) shows the overall design of the IVM framework. As can be seen, a single instance of the runtime daemon runs on each compute node and serves requests from the PEs physically residing on that node. Daemons running on different compute nodes communicate through inter-node communication channels which support Ethernet networks and Infiniband fabrics. Programmers can specifically choose preferred physical links to be used for data objects synchronization purposes. For simplicity, we implement inter-node communication using a master/slave model whereby a single root-daemon services requests issued by slave-daemons. This simple design adopted in our initial implementation avoids coherency problems on internal data structures residing on each daemon; further optimizations will be applied in the future.

As mentioned above, a user starts an IVM application by executing the application's binaries. Once the operating system has created a process for the application, the process will register itself to the daemon running on the local node. The process and local daemon will be considered the root-PE and root-daemon for the

application. The root-PE can request memory allocation, memory mapping, PE-creation, and PE-destruction. Communication between PEs and their associated daemon are performed through Linux Inter-Process-Communication (IPC) mechanisms. PEs sends requests to the daemon by invoking the API functions described in Section 0. If the servicing daemon is a slave-daemon, it will determine from the type of request whether it can be serviced locally without the assistance of the root-daemon. If the request needs assistance from the root-daemon, the slave-daemon will forward the request to it and wait for a response. Examples of requests that cannot be serviced locally by the slave-daemon include: PE registration, PE synchronization, setting global attributes, and memory allocation. Other memory operations, such as memory mapping, physical copy synchronization, and PE creation can be accomplished without the assistance of the root-daemon. The local daemon notifies the PEs upon completion of their requests.

When the root-daemon receives allocation requests, it allocates the master-copy of the data object using Linux shared memory, and it returns the object's reference to the requesting PE. At this point, the object is immediately available to all PEs that reside on the same physical node. However, the object is not allocated on other compute nodes until one of the remote PEs issues a mapping request for the object. Once the first memory-map request for a data object is received by a remote daemon, this daemon allocates a mirror-copy of the data object, synchronizes the content of the mirror-copy with the master-copy, and returns the reference of the mirror-copy to the requesting PE. Subsequent mapping requests of the same data object will not incur memory allocation or synchronization. Further synchronization between data copies requires the use of *put* and *get* API primitives.

5.3.4 GPU Support

In the IVM framework, PEs are allowed to share physical data objects (master-/mirror-copies). However, commonly used GPU software stacks (e.g., CUDA and OpenCL runtimes) offer limited support for sharing GPU memory across processes. In particular, the CUDA runtime associates a different memory address space to each process that uses a GPU. Therefore, a naïve design would allocate on GPU multiple data copies of each shared data objects – one per PE. This solution would be inefficient in terms of memory usage, especially when the data objects are read-only. In addition, it would lead to multiple and unnecessary GPU memory allocations and CPU-GPU data transfers. To address this issue, we include in our IVM framework one additional component: the *IVM GPU-daemon*. This daemon, shown in Figure 5-3(b), follows the design of the GPU virtualization runtime system proposed in our previous work [1]. Specifically, the IVM GPU-daemon consists of a frontend library and a backend daemon. The frontend library intercepts CUDA calls and redirects them to the backend daemon, which decides which requests should be issued to the CUDA runtime to be executed on GPU. In case of IVM, CUDA calls are generated only by the IVM daemon. In fact, applications have a uniform interface to CPUs and GPUs, and access their memories only through IVM API primitives.

As shown in Figure 5-3(b), the IVM runtime includes two memory paths: one for PEs associated to CPUs, and the other for PEs associated to GPUs. The PEs executing on CPU perform memory allocation and mapping through the sole IVM daemon. The memory-related operations originating from PEs executing on the GPU go first through the IVM daemon and then through the IVM GPU daemon. When a PE performs a

memory operation, the IVM daemon determines the compute resource used by the PE. If the resource is a GPU, then the IVM daemon generates the required CUDA calls to complete the operations. These CUDA calls are intercepted by the frontend library and redirect to the backend daemon of the GPU-daemon. By controlling all memory operations issued to GPUs, the GPU-daemon can avoid multiple physical copies of data objects shared by PEs mapped to the same GPU. This can be done by keeping track of the identifiers of the data objects allocated on each GPU, and selectively ignoring any subsequent `cudaMalloc` associated to the same data object. In summary, this design bypasses the restrictions of the CUDA runtime and allows multiple PEs using the same GPUs to truly share data objects.

API Functions	**Description**
Initialization & Finalization	
ivmEnter()	Registers a PE - This function should be the first IVM call.
ivmExit()	Unregisters a PE - This function should be the last IVM call.
Device Management	
ivmCreateDev()	Creates a device instance to represent a CPU/GPU.
ivmDestroyDev()	Releases resources allocated for a device instance.
Processing Element Management	
ivmCreatePEs()	Spawns one or more PEs on a specified device.
ivmKillPEs()	Terminates one or more PEs on a specified device.
ivmGetMyId()	Retrieves identification number of the calling PE.
ivmGetMyGroupId()	Retrieves group identification number of the calling PE.
ivmGetMyDevType()	Retrieves type of the device to which the calling PE is bound.
ivmSetMaxPesId()	Sets the maximum identification number of PEs that can be spawned. Subsequent calls to *ivmCreatePes()* after the maximum identification number is reached have no effect.
ivmWait()	Waits for a specified PE.
ivmSignal()	Signals the waiting PE to continue.
Memory Management	
ivmMalloc()	Allocates memory in the virtual memory space.
ivmFree()	Releases memory in the virtual memory space.
ivmMap()	Maps to a memory object in the virtual memory space.
ivmMapSubset()	Maps a specific region of a memory object.
ivmUnmap()	Un-maps from a memory object in the virtual memory space.
ivmSyncPut()	Writes content from the specified reference in a mirror-copy to the specified reference in the master-copy.
ivmSyncGet()	Reads content from the specified reference in the master-copy to the specified reference in a mirror-copy.
ivmSyncPutGroup()	Writes content from a list of references in the master-copy to a list of references in mirror-copies (Scatter/Broadcast).
ivmSyncGetGroup()	Reads content from a list of references in mirror-copies to a list of references in the master-copy (Gather).

Table 5-1 – IVM Programming API

5.3.5 IVM Programming Interface

The IVM API, shown in Table 5-1, consists of four categories of primitives: *Initialization & Finalization, Device Management*, *PE Management* and *Memory Management*.

The *ivmEnter* and *ivmExit* primitives belonging to the first category allow the programmer to register and deregister PEs with the IVM daemon, and their use is similar to that of MPI's *MPI_Init* and *MPI_Finalize*. These functions do not cause inter-PE communication or synchronization. PEs belonging to an application can invoke these functions asynchronously even after Dynamic Process Creation.

The device management primitives are used to create and destroy instances of *devices* – either CPUs or GPUs – and allow uniform access to them.

The PE management primitives allow the dynamic creation and release of PEs. Upon creation, each PE is associated to a specific device and assigned an identification number and a group identification number by the IVM runtime. The group identifier is meant for use by scheduling frameworks implemented on top of the IVM layer. The *ivmWait* and *ivmSignal* primitives facilitate synchronization among PEs and offer a mechanism similar to POSIX threads' conditional variables.

The memory management functions allow creating data objects within the shared virtual memory space and managing the consistency between copies of these objects residing on different physical memory spaces. Each data object is created by a PE through the *ivmMalloc* primitive and can be accessed by other PEs by invoking the *ivmMap* and *ivmMapSubset* memory mapping functions. The former allows a PE to access an object in its entirety whereas the latter allows a PE to access only a contiguous portion of an object and is intended to reduce data transfers between compute nodes and

devices. We recall that each PE is spawned on a specific device; the IVM runtime uses the association between PEs and devices to determine the physical memory where the master-copy of each data object should reside. The *put* and *get* primitives allow managing the content of the data objects. Specifically, *ivmSyncGet* and *ivmSyncGetGroup* read and gather data from mirror-copies of a data object to its master-copy, while *ivmSyncPut* and *ivmSyncPutGroup* write and scatter data from the master-copy to mirror-copies. All GPU-related memory operations generate corresponding CUDA function calls to the GPU-daemon. References to objects returned by the memory-related primitives can be used directly in GPU kernel calls. Because the IVM framework supports a one-sided communication paradigm, peer-to-peer communication must be explicitly coded by the programmer inside the application by properly using the provided *put* and *get* primitives on shared data objects.

```
1:  #define VECSIZE 5000
2:  int   id,     gid;
3:  int * vecA, * vecB, * vecC;
4:  int   vec_size = VECSIZE * sizeof(int);
5:  int main(int argc, char ** argv) {
6:    ivmEnter();
7:    ivmGetMyId(&id); ivmGetMyGroupId(&gid);
8:    if (id == 0) {
9:      ivm_device d[2]; ivm_pe pe[2];
10:     ivmMalloc("VECA", &vecA, vec_size);
11:     ivmMalloc("VECB", &vecB, vec_size);
12:     ivmMalloc("VECC", &vecC, vec_size);
13:     InitializeVectors();
14:     ivmCreateDev(host_0, IVM_CPU, &d[0]);
15:     ivmCreateDev(host_1, IVM_GPU_1, &d[1]);
16:     ivmCreatePEs(d[0], 1, &pe[0]);
17:     ivmCreatePEs(d[1], 1, &pe[1]);
18:     ivmWait(pe[0]);
19:     ivmWait(pe[1]);
20:   } else {
21:     int start_pos = (id - 1) * 2500;
22:     int end_pos = start_pos + 2500;
23:     ivmMap("VECA", &vecA, 2500);
24:     ivmMap("VECB", &vecB, 2500);
25:     ivmMap("VECC", &vecC, 2500);
26:     vectorAdd(start_pos, end_pos);
27:     ivmSyncPut(&vecC[start_pos], vec_size);
28:     ivmSignal();
29:   } ivmExit(); }
```

Listing 5-1 – Example of use of IVM API: vector addition

Listing 5-1 illustrates the use of the IVM API to implement vector addition. The code assumes that the application is initially invoked using a single process (the root-PE) and dynamically spawns two additional processes: one on CPU and one on GPU. The application invokes *ivmEnter* and *ivmExit* to register and unregister PEs (lines 6 and 29, respectively). Similar to MPI processes, after registration the PEs retrieve the identification numbers associated to them by the IVM runtime (line 7). Before starting the computation, the root-PE allocates the required arrays using *ivmMalloc* (lines 10-12). These arrays are visible to all compute nodes and devices (CPUs and GPUs), and their

identifiers *VECA*, *VECB* and *VECC* can be used by the other PEs to get access to them. At lines 14-15 the root-PE creates instances of devices to represent the CPU and GPU on which the computation will be performed (*IVM_CPU* and *IVM_GPU_1*, respectively). It then offloads the computation to the child-PEs (lines 16-17) and waits for them to complete (lines 18-19). The execution of the child-PEs also starts from line 5. Each child-PE uses its identification number (obtained at line 7) to identify the portion of the arrays on which it needs to operate (lines 21-22). The mapping operations at lines 23-25 allow accessing the vectors allocated by the root-PE; if the child-PE does not execute on the same compute node or device as the root-PE, the mapping operations cause the automatic creation of a mirror-copy of the arrays. Note that this uniform interface frees the programmer from the need to account for the physical location of the variables and the nature of the devices performing the computation. After the computation (line 26), the child-PEs write the results back to the master-copy via *ivmSyncPut* (line 27) and notify the root-PE to continue (line 28). The code assumes the availability of two versions of the *vectorAdd* function at line 26: one for CPU and one for GPU. The IVM runtime uses the PE-to-device association to automatically invoke the proper implementation.

5.3.6 Integration to Higher-level Scheduler

The IVM system and its API are designed to separate the programming of applications from their scheduling. Resource management frameworks providing inter-application scheduling and more advanced resource virtualization can be implemented on top the IVM layer. We are in the process of designing an inter-application scheduling layer – called *Dynamic Global Address Space (DGAS)* – on top of the IVM framework. The design of DGAS framework will be described in chapter 6.

5.4 Benchmark Applications and Load Balancing

In this section we describe our proposed load balancing schemes and four benchmark applications that we have implemented with the IVM framework.

5.4.1 Load Balancing Schemes

The IVM framework allows programmers to create PEs dynamically and PEs to communicate in an intuitive manner. In this section, we propose two load balancing schemes based on the DPC mechanism and show how they can be easily implemented using IVM API.

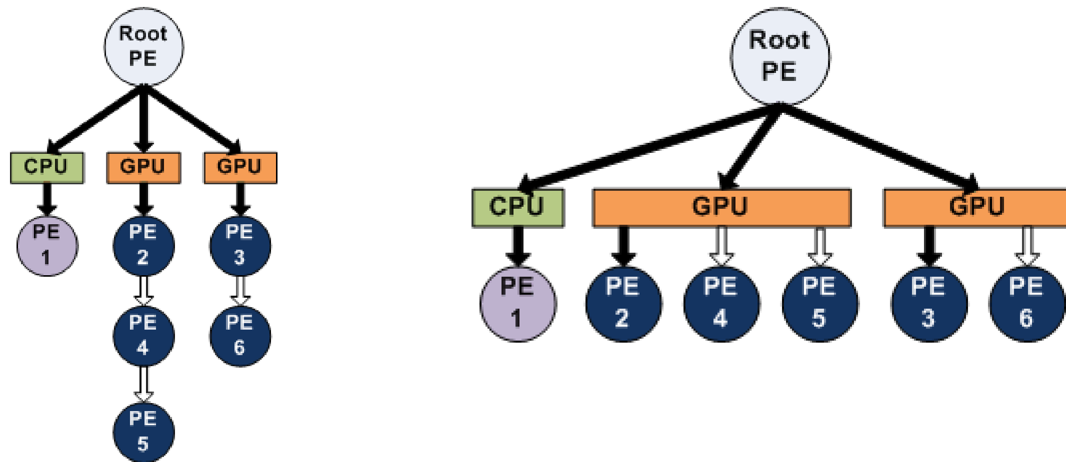
(a) Dynamic Spawning (DS-LB)

```
1  int main(int argc, char ** argv) {
2   ivmEnter();
3   Get my ID and my group ID
4   int *sync;
5   if (id == 0) { /* Root-PE */
6    ivmMalloc(&sync, … );
7    Create data objects
8    Create devices
9    for (i=0;i<ITERATIONS;i++) {
10    Set all slots of sync to 0
11    ivmSetMaxPesId(MAX_PEs * (i+1));
12    Create one PE on each device
13    Wait until all slots of sync are 1
14   }
15  } else {        /* Child-PEs */
16   ivmMap(&sync, … );
17   Map data objects
18   int work_id = GetWork(my_id);
19   Perform computation
20   if (unassigned work items exist){
21    ivmCreateDev(&d,THIS_HOST,THIS_DEV);
22    ivmCreatePEs(d, 1);
23   }
24   ivmSyncPut(my results);
25   sync[my_id-1] = 1;
26   ivmSyncPut(&sync[my_id-1], ...);
27  }
28  ivmExit();
```

(b) Online Monitoring (OM-LB)

```
1  int main(int argc, char ** argv) {
2   ivmEnter();
3   Get my ID and my group ID
4   double *time; bool *done;
5   if (id == 0) { /* Root-PE */
6    ivmMalloc(&time, …);
7    ivmMalloc(done, …); *done = 0;
8    Create data objects
9    Create devices
10   Create one PE on each device
11   for (i=0;i<ITERATIONS;i++) {
12    Wait all PEs to fill time
13    foreach (time[j])
14     if(time[j]/min(time) < threshold)
15      Create PE on device[j]
16   }
17   *done=1; ivmSyncPutGroup(done, …);
18  } else {        /* Child-PEs */
19   ivmMap(&time, …);
20   ivmMap(&done, …);
21   Map data objects
22   int work_id = GetWork(my_id);
23   do {
24    if (# of total PEs has changed)
25      work_id = GetWork(my_id);
26    Perform and time computation
27    time[my_id-1] = reported duration;
28    ivmSyncPut(my results);
29    ivmSyncPut(&time[my_id-1], …);
30   } while(*done == 0);
31  } ivmExit(); }
```

Listing 5-2 – Pseudo-code for load balancing schemes



(a) Dynamic Spawning (DS-LB) (b) Online Monitoring (OM-LB)

Figure 5-4 – Graphical representation of load balancing schemes

Dynamic Spawning Load Balancing (DS-LB) – As mentioned in Section 5.2.2, the static assignment of work to compute resources can cause load imbalance in the presence of performance heterogeneity. The DS-LB scheme enables work to be dynamically assigned to resources. Figure 5-4(a) presents a graphical representation of DS-LB. At a high level, this mechanism works as follows. The overall computation is broken into "work-portions", each executed by a PE. Each device will run a single PE at a time. When a PE finishes executing the work portion assigned to it, it spawns another PE on the same device. Because PEs associated to faster devices will spawn more PEs, more powerful devices will be assigned more work. The size of the work-portions affects the performance of this scheme. A small number of large work-portions will limit the degree of load balancing. Conversely, a large number of small work-portions will create more load balancing opportunities.

Listing 5-2(a) shows the pseudo-code for DS-LB. IVM API calls are bolded. Lines 6-13 are executed by the root-PE, while lines 16-26 are executed by the child-PEs (either spawned by the root-PE or by other child-PEs). Variable *sync* allows synchronization between root- and child-PEs. This variable, allocated and initialized by the root-PE (lines 6 and 10), contains a flag for each child-PE. The computation is broken in iterations (line 9), and up to MAX_PEs are spawned in each iteration (this limit is set at line 11). In each iteration, the root-PE first spawns a child-PE on each device (line 12) and then waits for all PEs to complete (line 13). The child-PEs use their identification number to retrieve the work-portions that they must execute, and then perform the computation (lines 18-19). Upon completion, if more work is still pending, each child-PE spawns a new PE on the same device where it resides (lines 20-23). Finally, every child-PE synchronizes its result with the master-copy (line 24), notifies the root-PE through the *sync* variable (lines 25-26), and terminates (line 28).

Online Monitoring Load Balancing (OM-LB) – At the high level, the OM-LB scheme mitigates the effect of load imbalance by monitoring the performance of different resources and dynamically assigning more work to more powerful devices. OM-LB limits the overhead of DPC by spawning fewer PEs and is suitable for applications with small running times. Figure 5-4(b) shows the graphical representation of the OM-LB scheme. The root-PE first distributes the work-portions equally to all resources as indicated by the solid arrows. During execution, the root-PE observes the performances of all PEs and spawns more PEs on the resources that can handle more work-portions, as indicated by pitted arrows. Once more PEs are created, all the PEs will involve in the procedure of re-dividing the work-portions to allow the new PEs to participate in the computation.

Differently from DS-LB, in this case all spawned PEs remain active for the entire computation, and each device can be time-shared by the executing PEs.

Listing 5-2(b) shows the pseudo-code of OM-LB. Lines 6-17 are executed by the root-PE, while lines 19-30 are executed by the child-PEs. The root-PE first allocates memory for variables *time* and *done* along with required data objects (lines 6-8). Variable *time* is an array used for synchronization purposes and holds the computation times reported back by the child-PEs. Variable *done* is used to notify the child-PEs when the entire computation is completed. The root-PE then creates devices, spawns PEs on all devices and waits for all PEs to report the time (lines 9-10 and 12). Once all PEs have reported time, it determines the performance differences of all PEs and spawns more PEs on the resources which cause performance differences larger than a threshold (line 13-16). The child-PEs map to data objects including variables *time* and *done* and perform work division (lines 19-22). Child-PEs that have already worked in previous iterations determine whether the number of total PEs is changed and re-divide the work-portions as necessary (line 24-25). Finally, all the PEs write back the results and report back the time (line 27-29) The PEs will terminate when the root-PE indicates that the entire computation is completed through the *done* variable (lines 17 and 30).

5.4.2 Benchmark Applications

N-body simulation (NBODY) – NBODY is a simulation of a dynamical system of particles. The computation is performed in time-steps. In each time-step, attributes of all particles, i.e. position and velocity, are updated.

Our IVM implementation of NBODY uses the DS-LB scheme. At each time-step, the calculations of particle attributes are divided into a number of work-portions, each

containing a subset of particles. The child-PEs retrieve work-portions based on their identifiers and spawn child-PEs dynamically until the overall computation completes. The MPI-CUDA is very similar to the IVM version except that subset of particles are statically assigned to processes.

Dense Matrix Multiplication (DMM) – DMM computes multiplication of two square matrices.

Our IVM implementation of DMM uses the DS-LB scheme. We divide the result matrix into *MxM* work-portions along both dimensions. The root-PE initializes the matrices and spawns the child-PEs; the child-PEs can then spawn more PEs as necessary. Because DMM involves large data transfers, we use *ivmMapSubset* for the child-PEs to map only the required parts of the multiplicand and multiplier matrices. This allows breaking a single large transfer into multiple smaller transfers, overlapping computations and data transfers, and avoiding broadcasting the whole matrices to all PEs. The tiles that have already been mapped can be reused by other child-PEs on the same physical node. Our MPI-CUDA implementation is similar to the IVM version except that processes are statically assigned portions of the result matrix column-wise and progress down the matrix.

Needleman-Wunsch (NW) – NW is a dynamic programming algorithm widely used in bioinformatics for comparing biological sequences. More detail can be found in our previous work [74].

Our IVM implementation of NW uses DS-LB. The root-PE and the child-PEs perform allocation and mapping of the entire dataset, respectively. Sequence-pairs are

divided into multiple work-portions which can be dynamically retrieved by the child-PEs. In the MPI-CUDA version, the sequence-pairs are distributed to processes equally.

Himeno (HIMENO) – HIMENO is a well-known benchmark application which implements a computational kernel found in the simulation of incompressible fluids. This kernel performs stencil computations on a 3-D grid of pressure values. We divide the computation in work-portions along the Z-axis of the grid. The MPI-CUDA version assigns to all processes the same amount of work along the Z-axis.

Because the kernel spends a small amount of time in the computation, we use OM-LB to minimize the overhead of DPC. We enable point-to-point communication for exchanging XY-planes by allocating a shared buffer. The sending PEs write into the corresponding slots of the buffer and synchronize the contents with the master-copy. Then, the contents in the master-copy are distributed to mirror-copies to allow the receiving PEs to access the corresponding slots. As the computation proceeds through iterations, the OM-LB scheme keeps performance records of all PEs and spawns more PEs on faster resources as necessary.

5.5 Experimental Evaluation

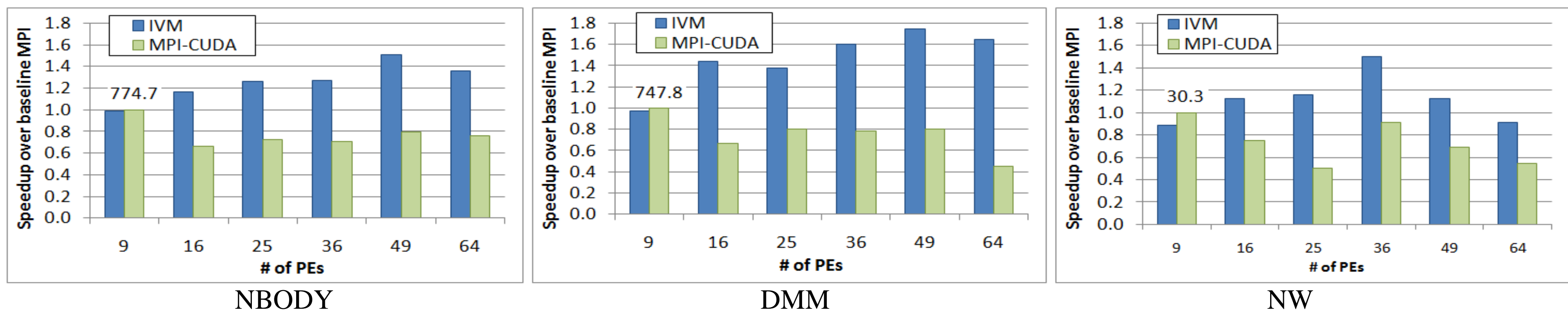
In this section we evaluate the performance and effectiveness of the two proposed load balancing schemes using the benchmark applications described in Section 5.4.2. Our experiments cover two aspects: (i) the comparison between the IVM and MPI-CUDA implementations of the applications described above; and (ii) the analysis of the effect of different parameter settings on load distribution.

5.5.1 Experimental Setup

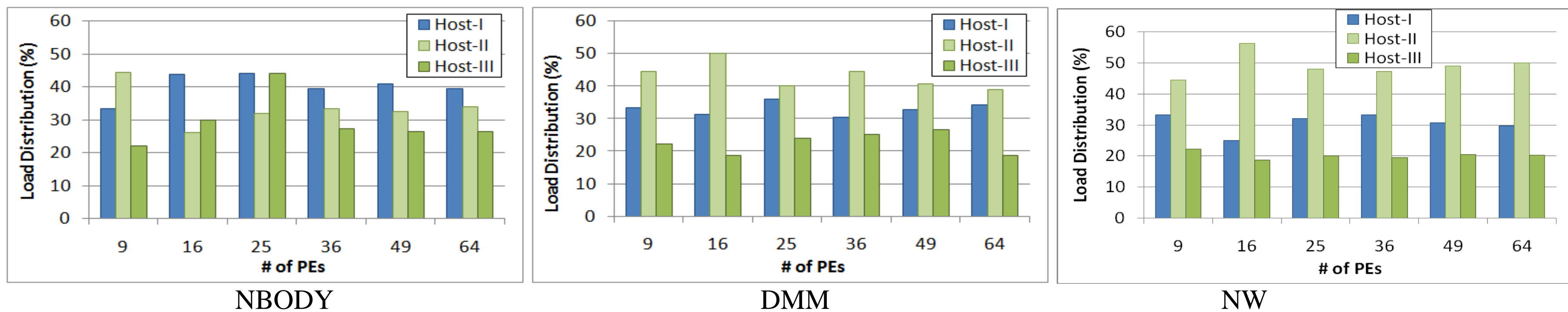
We conducted our experiments on a three-node cluster that includes nine Nvidia GPUs. Table 5-2 shows the hardware configuration of the compute nodes. As can be seen, the nodes differ in both the number of GPUs (from two to four) and their compute capability (e.g. number of cores, memory-speed and core-speed). This hardware heterogeneity can cause load imbalance within the application. The three nodes are interconnected through an Infiniband fabric, and they are each equipped with a Mellanox DDR HCA. CentOS 5.1, OpenMPI 1.8.1 and CUDA 6.5 are installed on every node.

Node	Attr.	Values
Host-I	Type	12-core Intel Xeon E5@2.00GHz, 15MB Cache, 64GB
	GPUs	1x Nvidia Tesla K40c (Kepler) 15 SMs x 192 cores @ 876 MHz 12 GB Global Memory @ 3004 MHz
		1x Nvidia Tesla C2075 (Fermi) 14 SMs x 32 cores @ 1147 MHz 5.6 GB Global Memory @ 1566 MHz
		1x Nvidia Tesla C2070 (Fermi) 14 SMs x 32 cores @ 1147 MHz 5.6 GB Global Memory @ 1494 MHz
Host-II	Type	8-core Intel Xeon E5@2.4 GHz, 12 MB Cache, 48GB
	GPUs	4x Nvidia GeForce GTX 480 (Fermi) 15 SMs x 32 cores @ 1401 MHz 1 GB Global Memory @ 1848 MHz
Host-III	Type	12-core Intel Xeon E5@2.10GHz, 15MB Cache, 64GB
	GPUs	1x Nvidia Tesla K20c (Kepler) 13 SMs x 192 cores @ 706 MHz 5 GB Global Memory @ 2600 MHz
		1x Tesla C2050 (Fermi) 14 SMs x 32 cores @ 1147 MHz 2.8 GB Global Memory @ 1500 MHz

Table 5-2 –Hardware setup



(a) Speedup over baseline MPI-CUDA. The numbers on the first bars indicate the execution time of the baseline MPI-CUDA code.



(b) Percentage load distribution across hosts

Figure 5-5 – Speedup and load distribution in case of DS-LB

5.5.2 Dynamic Spawning Load Balancing (DS-LB)

Performance Comparison – Figure 5-5(a) shows a performance comparison between the IVM and the MPI-CUDA versions of the three considered applications. The datasets used in these experiments have the following sizes: 1.2 million particles for NBODY, matrices of size 18,200×18,200 for DMM, and 4.8 thousand sequence pairs for NW. For both the IVM and the MPI-CUDA implementations, we vary the number of PEs (or processes) from 9 (the number of GPUs in the cluster) to 64. In case of MPI-CUDA, the processes are statically assigned to GPUs in a round-robin fashion. For IVM, we use the DS-LB scheme. Specifically, we divide the computation into a number of work-portions equal to the maximum number of PEs that we want to spawn over the execution of the application. Each PE will handle a work-portion. Initially the root-PE spawns 9 PEs (one per GPU); child-PEs are then spawned dynamically as needed until the number of PEs equals that of work-portions. Because the MPI-CUDA applications report the best performance when running with as many processes as GPUs, we take as baseline the performance of the 9-process configuration of the MPI-CUDA versions of the code.

Figure 5-5(a) shows the speedup/slowdown of IVM and MPI-CUDA over the baseline as the number of PEs (processes) varies from 9 to 64. When the number of PEs equals that of GPUs, IVM does not perform any load balancing. As can be seen, in this situation the IVM implementation of NBODY and DMM achieves the same performance as the baseline, while NW achieves slightly lower performance. The baseline execution time of NW (30.3 seconds) is significantly lower than that of NBODY and DMM (greater than 700 seconds): the slight performance penalty of the IVM version of NBODY is due

to the initialization overhead of the IVM runtime. As shown, while improving programmability by offering homogeneous access to compute resources and shared virtual memory, the IVM framework shows performance comparable to OpenMPI even in the absence of load balancing.

When the number of PEs exceeds that of GPUs, the GPUs are oversubscribed. MPI-CUDA statically assigns the same amount of work to all GPUs. IVM performs load balancing by allowing PEs that terminate earlier to spawn more PEs and assign them work. As the number of PEs increases, the size of the work-portions decreases, leading to finer-grained load distributions and thus higher degrees of load balancing. However, excessively increasing the number of PEs leads to high DPC overhead and consequent performance degradation. As shown in Figure 5-5(a), the use of more PEs than GPUs degrades performance for MPI-CUDA and improves performance for IVM. Thanks to the increased level of load balancing, the speedup of IVM over MPI-CUDA increases with the number of PEs, and reaches an optimal point at 49, 49, and 36 PEs for NBODY, DMM and NW, respectively. This corresponds to a speedup of 1.50, 1.74 and 1.50, respectively Due to the DPC overhead, a further increase in the number of PEs degrades performance. We also note that the use of the *ivmMapSubset* primitive in the DMM implementation allows caching and reusing read-only matrices. We observed that this mechanism reduces the inter-node communication of DMM by 33% to 65% as the number of PEs increases from 4×4 to 8×8.

Effects on Load Distribution – Figure 5-5(b) shows the percentage load distribution across the compute nodes for the IVM experiments of Figure 5-5(a). Specifically, the bars indicate the percentage of the load assigned to each host during

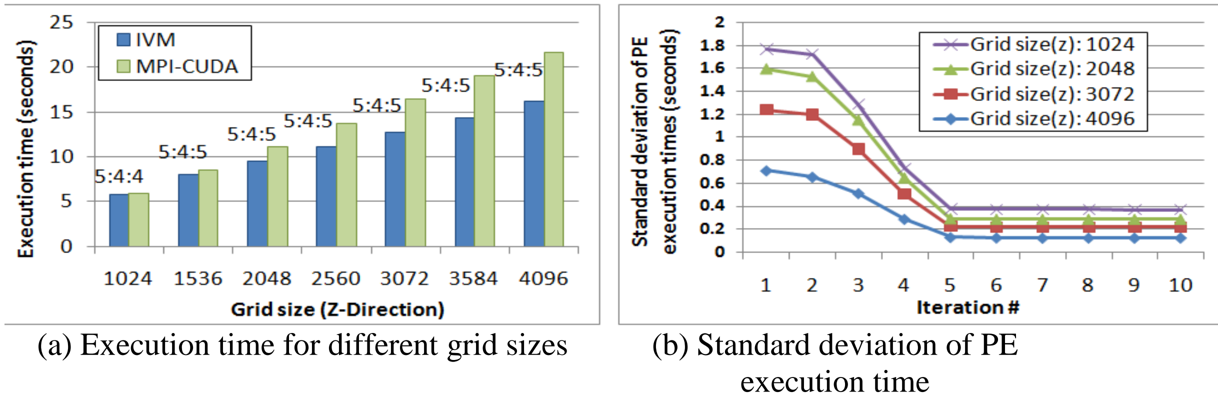
execution. As can be seen, the load distribution across nodes varies across applications. The majority of the load is assigned to *Host-I* in case of NBODY, and to *Host-II* in case of DMM and NW. The presence of heterogeneity among GPUs causes applications with different characteristics (i.e. compute- vs. memory-bound) to spawn PEs at different rates, thus affecting the load distribution. These results show that the DS-LB scheme provides applications with the ability to self-adapt to the available compute resources and thus achieve performance portability. Finally, we observe that the load distribution fluctuates when the number of PEs is low and becomes steady when this number increases. This is due to the finer-granularity work distribution occurring for larger numbers of PEs. In all cases, the application performance starts to gradually improve when the load distribution becomes steady. For NBODY, for example, this happens when the number of PEs is greater than 25; a similar effect can be observed on DMM and NW.

5.5.3 Online Monitoring Load Balancing (OM-LB)

The IVM implementation of HIMENO uses the OM-LB scheme. Figure 5-6(a) shows the execution time of the MPI-CUDA and IVM versions of HIMENO for different dataset sizes. Specifically, we fixed the size of the XY-plane to 513×513 and varied the size along the Z-dimension from 1,024 to 4,096. The numbers above the bars indicate the PEs spawned on the nodes (*Host-I:Host-II:Host-III*). Recall that the OM-LB scheme monitors the execution of all PEs and periodically spawns more PEs on faster compute resources and redistributes the work among the PEs mapped to these resources. In all experiments, the root-PE initially spawns 9 PEs (one per GPU), leading to a 3:4:2 (*Host-I,Host-II:Host-III)* load distribution. The threshold described in Section IV.A was set to 2 in all experiments. Because this scheme does not directly control the total number of PEs, in all

cases we compare the performance of IVM with the baseline MPI-CUDA code running with 9 processes (one per GPU). As reported in Figure 5-6(a), the speedup of IVM over the baseline ranges from 1.00x to 1.34x as the grid size along the Z-dimension varies from 1,024 to 4,096. The speedup is lower for small datasets, where the overhead of dynamic process creation, workload redistribution and inter-node communication weighs more on the execution time. The load distribution becomes more balanced during the application's execution. OM-LB spawns more PEs on *Host-I* and *Host-III,* which are equipped with more powerful GPUs.

We expect the PEs to take the same amount of time once the load distribution reaches a steady state. Powerful GPUs are expected to be time-shared by more PEs. We confirm this intuition by measuring the standard deviation of the execution time of the PEs, as reported in Figure 5-6(b). As can be seen, this metric is high during the first few iterations of the application and decreases during its execution, as the OM-LB scheme increasingly submits PEs to the powerful GPUs on *Host-I* and *Host-III*.



(a) Execution time for different grid sizes

(b) Standard deviation of PE execution time

Figure 5-6 – Execution time of HIMENO with OM-LB

5.5.4 Discussion

A limiting factor for the performance of the IVM runtime is the dynamic process creation overhead, which can reduce the positive effect of load balancing. To see how this overhead affects the application performance, we performed a set of experiments on applications that use the DS-LB scheme. In these experiments, we scaled down the size of the input datasets as follows: 300 thousand particles for NBODY, matrices of size 2,275×2,275 for DMM, and 1.2 thousand sequence pairs for NW. In all cases, we used an optimal number of PEs. The speedups of IVM over the baseline MPI-CUDA (with 9 processes) are 0.78, 0.34, and 0.55 for NBODY, DMM, and NW respectively. In all cases, the DPC overhead dominates the computation time and each DPC operation is estimated to take 0.11 to 0.2 seconds depending on the application (the overhead also includes users' code for initializing PEs). This DPC overhead can especially penalize applications with short running times (less than 30 seconds). However, these short running applications do not motivate the use of a large number of high-performance computing nodes.

In the future, we plan to evaluate our framework on larger settings and optimize it for scalability to large clusters. While applications that require deployment in these environments are typically long running and therefore less affected by the DPC overhead, a scalable implementation of our framework requires the optimization of the communication layer, which affects the overhead for shared virtual memory handling and dynamic process creation. The implementation of our framework on-top of MPI and the reuse of communication libraries such as GASNet [76] require significant design and engineering work, as our shared virtual memory design diverges from existing models.

CHAPTER 6

CONCLUSION & FUTURE WORK

GPUs have become the primary computing resource for a large number of scientific and engineering applications and are increasingly part of HPC clusters. However, the current GPU software stacks including CUDA and OpenCL view GPU devices as dedicated accelerators causing performance and underutilization issues in heterogeneous clusters. Since the existing cluster resource managers rely directly on these software stacks and, as such, they inherit their view of GPUs and their limitations. The resource managers, therefore, offer limited and basic support for GPUs.

In this chapter, we first describe a detailed analysis of the limitation of the existing cluster management and programming frameworks for CPU-GPU clusters. Then, we describe how we leveraged the lessons learned from this study to propose a hierarchical scheduling framework and a programming framework that overcome such limitations. The remainder of this chapter first describes the lessons learned from the analysis and, then, describes our contributions.

Lessons Learned

Resource Abstraction–Resources in a heterogeneous cluster are structured hierarchically. In particular, a heterogeneous cluster includes multiple compute nodes wherein each node consists of multi-core CPUs and a number of GPUs, possibly with different performance

capabilities. The management of resources can, therefore, be performed at a coarse- and fine-grain level. Because of the lack of resource abstraction, existing resource managers and software stacks cause inefficiencies in the distribution of the load to resources in the following ways:

- At a fine-grain, the current software stacks including CUDA and OpenCL view GPU as dedicated accelerator and require users to select and manage explicitly GPUs for their computations. Since the users have full control on the GPUs, it is not possible for the system dynamically to perform scheduling and load balancing across GPU devices. Hence, users may experience performance degradation due to interference between applications.
- To assign load to compute nodes at a coarse-grain, users need to retain knowledge on the underlying configuration and be aware of the current load condition of the cluster. Such information is necessary to make a request to the resource manager in order to allow the application to execute with minimal performance penalty. Submitting a non-optimal resource request can cause the user and the service provider to suffer from low Quality-of-Service and resource underutilization, respectively. In addition, since the resource managers rely directly on CUDA and OpenCL, their ability to perform scheduling and load balancing is limited.

Fine-grained Sharing of GPUs – Due to the nature of single- and multi-process GPU applications, the coarse-grain space sharing mechanism provided by existing resource managers, which maps only a single process to a GPU, is not adequate to improve GPU utilization in heterogeneous clusters.

Not only does the alternation between phases of applications underutilize compute resources, but the presence of multi-threaded and multi-process applications that include communication and synchronization also leads to underutilization in the presence of intra- and inter-application imbalance. These aspects are not considered by the existing resource managers and GPU software stacks. Without fine-grained resource sharing, an application running on a set of dedicated resources (including CPUs and GPUs) will certainly underutilize both CPUs and GPUs because of these aspects.

Fine-grained sharing, such as time-sharing of a GPU, allows multiple processes to share the GPU at a finer-grain, which allows the followings to be achieved: (i) better GPUs utilization, (ii) increased overall throughput, and (iii) improved Quality-of-Service in some cases. However, time-sharing is limited with CUDA and OpenCL and is not allowed by the existing resource managers. Although, time-sharing can be used to address the inefficiencies, it needs to be performed judiciously to avoid the negative effects on application performance.

Scheduling & Load Balancing – The presence of heterogeneity in a cluster causes performance bottleneck for both single- and multi-process applications and GPU underutilization. At the coarse- and fine-grain, a good load balancing is necessary to appropriately distribute load to nodes and GPUs, respectively. The evaluation of effectiveness of scheduling and load balancing policies depends on the requirements posed by the users and the service provider, and these requirements may change over time. Different scheduling algorithms can be employed in the policies to optimize specific performance metrics. Flexibility is the key to accommodating the changing needs of the users and the service providers. Specifically, the effective resource schedulers

should provide the ability for administrators to apply specific scheduling schemes, both at the coarse- and fine-grain, which are particularly optimized for the cluster. In addition, these scheduling policies should not rely on the input or profiling information from the users since the information provided by the users may be inaccurate, leading to inefficient load placement.

Programmability – Distributed GPU applications running in heterogeneous clusters mainly rely on traditionally programming models, such as MPI and SHMEM, which do not handle heterogeneity well and expose a complex view of memory spaces. Because of these complexities, programmers cannot focus on the main algorithms of the applications and have to spend significant effort on house-keeping tasks such as distributing load to heterogeneous resources and addressing issues of disjoint memory spaces. The execution and memory models for distributed applications need to be revisited to achieve a programming model that facilitates load balancing and provides an intuitive memory model.

Contributions

At a high-level, the limitations of existing resource management, scheduling, and programming frameworks are linked to the view of GPU as dedicated accelerators, which implies the following: First, there is no resource virtualization. Second, resource sharing is very limited. And, finally, GPU memory is viewed as dedicated GPU resource. These are the factors that cause inefficiencies and issues mentioned above. To address the issues of heterogeneous clusters, we view GPUs as primary computing resources by proposing the following frameworks: (1) a hierarchical scheduling framework for CPU-GPU

clusters to address the performance and underutilization issues both at the coarse- and fine-grained level, and (2) a programming framework (IVM - Inter-node Virtual Memory) for CPU-GPU clusters that target the programmability/usability issues. The former can be used both with unmodified GPU binaries or in part as scheduling framework for the programming framework.

- To resolve issues at the fine-grained level (i.e. load distribution of GPU resources), we have proposed a runtime system that provides abstraction and sharing of GPUs while allowing isolation of concurrent applications. Two fundamental features of our runtime are: (i) dynamic application-to-GPU binding and (ii) virtual *memory for GPUs*. In particular, dynamic binding maximizes device utilization and improves performances in the presence of concurrent applications with multiple GPU phases and of GPUs with different compute capabilities. Besides dynamic binding, the virtual memory abstraction enables the following features: (i) load balancing in case of GPU addition and removal, (ii) resilience to GPU failures, and (iii) checkpoint-restart capabilities.
- Next, we have proposed GPU preemption as a mechanism to address the inefficiencies of scheduling multi-threaded and multi-process applications that include synchronization. We have implemented GPU preemption in our node-level runtime system to schedule efficiently multi-process applications on CPU-GPU nodes and clusters.

- To resolve coarse-grained scheduling problems (i.e. assignment of application processes on compute nodes and CPU resources) we have proposed a hierarchical scheduling framework for heterogeneous CPU-GPU clusters. Our system includes a scheduling API that allows administrators to define custom cluster-level scheduling policies and to configure the sharing and scheduling schemes used at the node level. We have proposed two cluster-level scheduling schemes that leverage GPU sharing at the node-level (co-location and latency-reduction-based scheduling).
- To increase programmability of CPU-GPU clusters, we have proposed Inter-node Virtual Memory (IVM): a parallel programming model for distributed applications, which offers a uniform view of compute resources, and a simplified shared memory abstraction. We also described the design of a runtime framework that supports IVM. IVM, which is meant to increase programmability and reduce the complexity of application codes, offers Dynamic Process Creation (DPC) as a mechanism to facilitate dynamic load balancing within applications. We designed two self-tuning load balancing schemes based on DPC and tested them on four benchmark applications.

Future Work

Our hierarchical scheduling framework provides a supporting platform to manage heterogeneous workloads and resources efficiently. As we mentioned, different scheduling and load balancing policies can be deployed in our framework. To better accommodate needs and requirements from users and service providers, some aspects

need to be explored to derive scheduling policies for our hierarchical scheduling platform. For example, analysis of codes at run-time can aid the scheduler in making scheduling decisions. Since performance of a GPU application varies across GPU devices, the code analysis can assist the scheduler to select the suitable device for the application. In addition, the characteristic of application can be estimated by the code analysis. This allows the scheduler to estimate the amount of resource used on each GPU (e.g. register, shared-memory, and global-memory) and may allow co-location of applications on a GPU so as to avoid interference with another applications. Several aspects such as overall power consumption, data locality, and communication patterns have effects on application performance and hardware utilization. These aspects can be further considered to yield scheduling policies at both the coarse- and fine-grain.

Since the design of our IVM programming framework is based on our hierarchical scheduling framework, our primary goal is to use the programming framework as a substrate for constructing high-level programming model. We plan to derive a programming model based on IVM that will further increase the programmability of heterogeneous clusters and will provide the flexibility to manage resources for concurrent applications. In addition, it is possible to use the IVM programming framework in a computation at a larger scale, such as a grid computation. In a grid computation, heterogeneity is commonly presented among clusters and data transfers between clusters have to be minimized. We plan to extend our framework to investigate data placement on clusters to minimize communication and to distribute judiciously the load among clusters.

BIBLIOGRAPHY

1) Becchi, M., et al., *A virtual memory based runtime to support multi-tenancy in clusters with GPUs*, in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. 2012, ACM: Delft, The Netherlands. p. 97-108.

2) Sajjapongse, K., X. Wang, and M. Becchi, *A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with GPUs*, in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. 2013, ACM: New York, New York, USA. p. 179-190.

3) *Top 500 Supercomputing Sites*. Available from: http://www.top500.org/.

4) Danalis, A., et al., *The Scalable Heterogeneous Computing (SHOC) benchmark suite*, in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. 2010, ACM: Pittsburgh, Pennsylvania, USA. p. 63-74.

5) Owens, J.D., et al., *GPU Computing.* Proceedings of the IEEE, 2008. **96**(5): p. 879-899.

6) Shuai, C., et al. *Rodinia: A benchmark suite for heterogeneous computing*. in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. 2009.

7) *GPU Accelerated Applications (Nvidia Catalog)*. Available from: http://www.nvidia.com/docs/IO/123576/nv-applications-catalog-lowres.pdf.

8) *Torque Resource Manager*. Available from: http://www.adaptivecomputing.com/products/open-source/torque/.

9) *SLURM: A Highly Scalable Resource Manager*. Available from: https://computing.llnl.gov/linux/slurm/.

10) Sajjapongse, K., T. Agarwal, and M. Becchi, *A Flexible Scheduling Framework for Heterogeneous CPU-GPU Clusters*, in *Proceedings of the 21st Annual International Conference on High Performance Computing*. 2014: Goa, India.

11) Juan, W. and G. Wenming. *The Application of Backfilling in Cluster Systems*. in *Communications and Mobile Computing, 2009. CMC '09. WRI International Conference on*. 2009.

12) Lawson, B.G., E. Smirni, and D. Puiu. *Self-adapting backfilling scheduling for parallel systems*. in *Parallel Processing, 2002. Proceedings. International Conference on*. 2002.

13) Zotkin, D. and P.J. Keleher. *Job-length estimation and performance in backfilling schedulers*. in *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*. 1999.

14) *CUDA C Programming Guide*. 2015, NVIDIA Corporation. p. 240.

15) Da, L., et al. *A distributed CPU-GPU framework for pairwise alignments on large-scale sequence datasets*. in *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*. 2013.

16) *Compute Unified Device Architecture (CUDA)*. Available from: https://developer.nvidia.com/cuda-zone.

17) *The open standard for parallel programming of heterogeneous systems*. Available from: https://www.khronos.org/opencl/.

18) *IEEE 802.3 Ethernet Working Group*. Available from: http://www.ieee802.org/3/.

19) *OpenFabrics Alliance*. Available from: https://www.openfabrics.org/.

20) *Open MPI: Open Source High Performance Computing*. Available from: http://www.open-mpi.org/.

21) *MPICH: High-Performance Portable MPI*. Available from: https://www.mpich.org/.

22) *MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE*. Available from: http://mvapich.cse.ohio-state.edu/.

23) *Unified Parallel C*. Available from: https://upc-lang.org/.

24) *Coarray Fortran*. Available from: http://en.wikipedia.org/wiki/Coarray_Fortran#References.

25) Feind, K., *Shared Memory Access (SHMEM) Routines*, in *Proceedings of the 1995 Cray User Group*. 1995. p. 303 - 308.

26) *OpenSHMEM*. Available from: http://www.openshmem.org/.

27) Gupta, V., et al., *GViM: GPU-accelerated virtual machines*, in *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*. 2009, ACM: Nuremburg, Germany. p. 17-24.

28) Lin, S., C. Hao, and S. Jianhua. *vCUDA: GPU accelerated high performance computing in virtual machines*. in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. 2009.

29) Duato, J., et al. *rCUDA: Reducing the number of GPU-based accelerators in high performance clusters*. in *High Performance Computing and Simulation (HPCS), 2010 International Conference on*. 2010.

30) Giunta, G., et al., *A GPGPU transparent virtualization component for high performance computing clouds*, in *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*. 2010, Springer-Verlag: Ischia, Italy. p. 379-391.

31) Gelado, I., et al., *An asymmetric distributed shared memory model for heterogeneous parallel systems*, in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. 2010, ACM: Pittsburgh, Pennsylvania, USA. p. 347-358.

32) Becchi, M., et al., *Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory*, in *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. 2010, ACM: Thira, Santorini, Greece. p. 82-91.

33) Guevara, M., et al., *Enabling Task Parallelism in the CUDA Scheduler*, in *Programming Models and Emerging Architectures Workshop (Parallel Architectures and Compilation Techniques Conference)*. 2009.

34) Ravi, V.T., et al., *Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework*, in *Proceedings of the 20th international symposium on High performance distributed computing*. 2011, ACM: San Jose, California, USA. p. 217-228.

35) Feitelson, D.G., L. Rudolph, and U. Schwiegelshohn, *Parallel job scheduling — a status report*, in *Proceedings of the 10th international conference on Job Scheduling Strategies for Parallel Processing*. 2005, Springer-Verlag: New York, NY. p. 1-16.

36) Hori, A., H. Tezuka, and Y. Ishikawa, *Highly efficient gang scheduling implementation*, in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. 1998, IEEE Computer Society: San Jose, CA. p. 1-14.

37) Agarwal, S., et al. *Co-ordinated coscheduling in time-sharing clusters through a generic framework*. in *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*. 2003.

38) Anglano, C., *A Performance Comparison of Coscheduling Strategies for Workstation Clusters.* Cluster Computing, 2001. **4**(2): p. 121-131.

39) Arpaci-Dusseau, A.C., *Implicit coscheduling: coordinated scheduling with implicit information in distributed systems.* ACM Trans. Comput. Syst., 2001. **19**(3): p. 283-331.

40) Choi, G.S., et al., *Performance Comparison of Coscheduling Algorithms for Non-Dedicated Clusters Through a Generic Framework.* Int. J. High Perform. Comput. Appl., 2007. **21**(1): p. 91-105.

41) Dusseau, A.C., R.H. Arpaci, and D.E. Culler, *Effective distributed scheduling of parallel workloads*, in *Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 1996, ACM: Philadelphia, Pennsylvania, USA. p. 25-36.

42) Feitelson, D.G. and L. Rudolph, *Coscheduling based on runtime identification of activity working sets.* Int. J. Parallel Program., 1995. **23**(2): p. 135-160.

43) Gyu Sang, C., et al. *Coscheduling in Clusters: Is It a Viable Alternative?* in *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*. 2004.

44) Sobalvarro, P., et al., *Dynamic Coscheduling on Workstation Clusters*, in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. 1998, Springer-Verlag. p. 231-256.

45) Yoo, A. and M. Jette, *An Efficient and Scalable Coscheduling Technique for Large Symmetric Multiprocessor Clusters*, in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson and L. Rudolph, Editors. 2001, Springer Berlin Heidelberg. p. 21-40.

46) Mars, J., et al., *Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations*, in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 2011, ACM: Porto Alegre, Brazil. p. 248-259.

47) Mars, J., et al., *Contention aware execution: online contention detection and response*, in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 2010, ACM: Toronto, Ontario, Canada. p. 257-265.

48) *Texas Advanced Computing Center*. Available from: https://www.tacc.utexas.edu/.

49) Phull, R., et al., *Interference-driven resource management for GPU-based heterogeneous clusters*, in *Proceedings of the 21st international symposium on*

High-Performance Parallel and Distributed Computing. 2012, ACM: Delft, The Netherlands. p. 109-120.

50) Ravi, V.T., et al., *ValuePack: value-based scheduling framework for CPU-GPU clusters*, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, IEEE Computer Society Press: Salt Lake City, Utah. p. 1-12.

51) Ravi, V.T., et al., *Scheduling Concurrent Applications on a Cluster of CPU-GPU Nodes*, in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. 2012, IEEE Computer Society. p. 140-147.

52) Irwin, D.E., L.E. Grit, and J.S. Chase, *Balancing Risk and Reward in a Market-Based Task Service*, in *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*. 2004, IEEE Computer Society. p. 160-169.

53) *Unified Memory in CUDA 6*. Available from: http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/.

54) Potluri, S., et al. *Extending OpenSHMEM for GPU Computing*. in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. 2013.

55) Dinan, J., et al. *Dynamic Load Balancing of Unbalanced Computations Using Message Passing*. in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. 2007.

56) Olivier, S., et al., *UTS: An Unbalanced Tree Search Benchmark*, in *Languages and Compilers for Parallel Computing*, G. Almási, C. Caşcaval, and P. Wu, Editors. 2007, Springer Berlin Heidelberg. p. 235-250.

57) *MPI-2 Extensions to the Message-Passing Interface*. Available from: http://www.mpi-forum.org.

58) Kale, L.V. and S. Krishnan, *Charm++: Parallel Programming with Message-Driven Objects*, in *Parallel Programming using C++*, G.V. Wilson and P. Lu, Editors. 1996, MIT Press. p. 175-213.

59) Acun, B., et al., *Parallel programming with migratable objects: charm++ in practice*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, IEEE Press: New Orleans, Louisana. p. 647-658.

60) Kurt, M.C. and G. Agrawal, *DISC: a domain-interaction based programming model with support for heterogeneous execution*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, IEEE Press: New Orleans, Louisana. p. 869-880.

61) *HP Helion Eucalyptus*. Available from: https://www.eucalyptus.com/.

62) Becchi, M. and P. Crowley, *Dynamic thread assignment on heterogeneous multiprocessor architectures*, in *Proceedings of the 3rd conference on Computing frontiers*. 2006, ACM: Ischia, Italy. p. 29-40.

63) *gVirtuS*. Available from: http://osl.uniparthenope.it/projects/gvirtus.

64) *BLCR*. Available from: http://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/.

65) *Rodinia:Accelerating Compute-Intensive Applications with Accelerators*. Available from: https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Main_Page.

66) Becchi, M. and P. Crowley, *A-DFA: A Time- and Space-Efficient DFA Compression Algorithm for Fast Regular Expression Evaluation.* ACM Trans. Archit. Code Optim., 2013. **10**(1): p. 1-26.

67) Needleman, S.B. and C.D. Wunsch, *A general method applicable to the search for similarities in the amino acid sequence of two proteins.* Journal of Molecular Biology, 1970. **48**(3): p. 443-453.

68) Needleman, S.B. and C.D. Wunsch, *A general method applicable to the search for similarities in the amino acid sequence of two proteins.* Journal of Molecular Biology, 1970. **48**(3): p. 443-453.

69) Duato, J., et al., *Enabling CUDA acceleration within virtual machines using rCUDA*, in *Proceedings of the 2011 18th International Conference on High Performance Computing*. 2011, IEEE Computer Society. p. 1-10.

70) Sajjapongse, K., T. Agarwal, and M. Becchi, *A Flexible Scheduling Framework for Heterogeneous CPU-GPU Clusters*, in *Proceedings of the 21st IEEE International Conference on High Performance Computing*. 2014, IEEE: Goa, India.

71) Sengupta, D., et al., *Scheduling multi-tenant cloud workloads on accelerator-based systems*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, IEEE Press: New Orleans, Louisana. p. 513-524.

72) Gabriel, E., et al., *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*, in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Editors. 2004, Springer Berlin Heidelberg. p. 97-104.

73) Curtis, T., et al., *OpenSHMEM Specification 1.1 Final*. 2014.

74) Li, D., et al., *A Distributed CPU-GPU Framework for Pairwise Alignments on Large-Scale Sequence Datasets*, in *Proceedings of the 24th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2013: Ashburn, VA.

75) *MPI-2: Extensions to the Message-Passing Interface*. 2003; Available from: http://www.mpi-forum.org/.

76) Bonachea, D., *GASNet Specification, v1.1*, in *U.C. Berkeley Tech Report (UCB/CSD-02-1207)*. 2002.

VITA

Kittisak Sajjapongse obtained his Bachelor of Engineering from the Department of Electrical engineering, Faculty of Engineering, Mahidol University in 2005. After graduation, he worked as a Design Engineer and a Research and Development Engineer for 3 years. He then attended graduate school to receive his Master of Science in Computer Engineering at the Department of Electrical and Computer Engineering, University of Missouri – Columbia, from January 2008 to May 2010 under Dr. Tina Smilkstein's guidance. He continued his Doctoral of Philosophy in Electrical and Computer Engineering at the same department commencing in May 2010 and graduated in August 2015 under Dr. Michela Becchi's guidance.