Runtime Systems and Scheduling Support for High-End CPU-GPU Architectures

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Vignesh Trichy Ravi, B.E.

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2012

Dissertation Committee:

Gagan Agrawal, Advisor
P Sadayappan
Feng Qin
Abstract

In recent years, multi-core CPUs and many-core GPUs have emerged as mainstream and cost-effective means for scaling. Consequently, a trend that is receiving wide attention is of heterogeneous computing platforms consisting of both CPU and GPU. Such heterogeneous architectures are pervasive across notebooks, desktops, clusters, supercomputers and cloud environments. While they expose huge potential for computing, the state-of-the-art software support lacks much of the desired features to improve the performance and utilization of such systems. Particularly, we focus on three important problems: (i) While machines consisting of both multi-core CPU and GPU are available, there is no standard software support that enables application to harness aggregate compute power of both CPU and GPU, (ii) Although GPUs offer very high peak performance, often, its utilization is low, which is an important concern in heavily shared cloud environments. While resource sharing is a classic way to improve utilization, there is no software support to truly share the GPUs, and (iii) In shared supercomputers and cloud environments, a critical software component is a job scheduler, which aims at improving the resource utilization and maximizing the aggregate throughput. Thus, we formulate and revisit scheduling problems for CPU-GPU clusters.

For the first problem, we have developed a runtime system that will enable an application to simultaneously benefit from the aggregate computing power of available CPU and GPU. Starting from a high-level API support, the runtime system transparently handles the
concurrency control, and efficiently distributes the work automatically between CPU and GPU. This work has been extended and optimized to consider structured grid computation pattern. Our evaluation shows that significant performance benefits can be achieved while also improving the productivity of the user.

For the second problem, we have developed a framework with runtime support for enabling one or more applications to transparently share one or more GPUs. We use consolidation as a mechanism to share a GPU and provide solutions to the conceptual problem of consolidation through affinity score and molding. Particularly, affinity score between two or more kernels provides an indication of potential performance improvement upon kernel consolidation. In addition, we explore molding as a means to achieve efficient GPU sharing in the case of kernels with conflicting resource requirements. As such, we demonstrate significant performance improvements from the use of our GPU sharing mechanisms.

For the third problem, our scheduling formulations actively exploit the portability offered by programming models like OpenCL to automatically map jobs to CPU and GPU resources in the cluster. Based on this assumption, we have developed a number of scheduling schemes with two different goals. One is based on system-wide metrics like global throughput (make span) and latency, while the other is based on market-based metrics (known as value or yield) as defined or agreed between user and the service provider. Particularly, our scheduling schemes improve the utilization (thus, global throughput) by minimizing resource idle time, and also by efficiently handling the trade-off between queuing delay and non-optimal resource penalty. When the goal is to improve yield, we also factor various parameters of value functions (in addition to afore-mentioned trade-offs) into scheduling decisions. Our experimental results show that our schemes can significantly outperform the state-of-the-art solutions in practice.
To my family, friends, and mentors.
Acknowledgments

This work is far from complete without proper acknowledgment of those who continued to support me over the years.

My family. My parents have been extremely supportive and encouraging throughout my graduate studies, especially during tough times. They did a lot of sacrifices for me and have always kept my education as their top priority. They also brought me up with great values in life, including hard work, humility and patience. I also thank my cousin, Satish, who has played a notable part during this transformational phase with a lot of technical discussions.

My wife, Radha, for her infinite patience and unwavering love and support during even the most testing times. I respect her deeply, and has been a great inspiration to me.

My mentors and Collaborators. I am very grateful to have worked with my advisor, Gagan Agrawal, who supported me throughout my doctoral program. I initially started out with him as an MS student and he suggested me to continue for Ph.D. at the end of my first year. I deeply thank him for that recommendation. He always gave me sufficient freedom to suggest new problems and work with others (inside or outside his research group). I am always amazed by his availability for technical discussions. I am sure that my association with him has transformed me both technically and personally. I also thank Srimat Chakradhar and Michela Becchi with whom I have collaborated for the past two years. They have always been very supportive and their guidance towards my dissertation
has been very useful. A special thanks for their advice during my job search. I also thank Sadayappan, Feng Qin, and Chirstopher Stewart for their constructive feedback.

My friends and extended family, with whom I have been able to freely share both my happiness and disappointments without any hesitation. My special thanks to my former roommates: Balaji, Hari and MRK for the immense patience they have shown towards me and the innumerable pranks we have played together. I am happy to have made a number of close friends at OSU: Tekin, David, Naveen, Wenjing, Wei, Xin, Bin, Leo, Qian, Sai, Muthu, Shantanu, Shirish, Mai, Mehmat and Tantan.
Vita

2001-2005 ....................... B.E., Computer Science and Engineering, P.S.G College of Technology, India
2005-2007 ....................... Analyst, Verizon Data Services, India
2007-Present ...................... Ph.D., Computer Science and Engineering, The Ohio State University, U.S.A
2009-Present ...................... Graduate Research Associate, The Ohio State University
2011 ............................... Summer Research Intern, NEC Laboratories America
2010 ............................... Summer Research Intern, NEC Laboratories America
2007-2009 ......................... Graduate Teaching Associate, The Ohio State University

Publications


Fields of Study

Major Field: Computer Science and Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiv</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Aggregate CPU-GPU Power and Automatic Work Distribution</td>
<td>3</td>
</tr>
<tr>
<td>1.1.2 GPU Sharing in the Cloud Environments</td>
<td>5</td>
</tr>
<tr>
<td>1.1.3 Scheduling to Improve Global Throughput on CPU-GPU Clusters</td>
<td>8</td>
</tr>
<tr>
<td>1.1.4 Market-based Job Scheduling on CPU-GPU Clusters</td>
<td>9</td>
</tr>
<tr>
<td>1.2 Contributions</td>
<td>11</td>
</tr>
<tr>
<td>2. Related Work</td>
<td>14</td>
</tr>
<tr>
<td>2.1 Programming Models and Runtime Frameworks</td>
<td>14</td>
</tr>
<tr>
<td>2.2 Dynamic Scheduling between CPU and GPU</td>
<td>15</td>
</tr>
<tr>
<td>2.3 GPU Virtualization and Workload Consolidation on GPU</td>
<td>16</td>
</tr>
<tr>
<td>2.4 Job Scheduling to Improve Global Throughput</td>
<td>17</td>
</tr>
<tr>
<td>2.5 Value-based Job Scheduling</td>
<td>18</td>
</tr>
</tbody>
</table>
3. Runtime Support for Generalized Reduction Computations on CPU-GPU Systems 20

3.1 Approach and System Design ........................................ 20
3.1.1 Generalized Reduction Structure ................................ 21
3.1.2 Parallelization Approach for Multi-cores .................... 22
3.1.3 GPU Computing for Generalized Reductions .................. 23
3.1.4 System Design for a Heterogeneous Platform .................. 23
3.1.5 Dynamic Work Distribution ........................................ 25

3.2 Experimental Results .............................................. 28
3.2.1 Scalability of CPU-only and GPU-only versions ............. 30
3.2.2 Performance of Heterogeneous Configurations ............... 32
3.2.3 Work Distribution ................................................. 36

3.3 Summary ............................................................. 38

4. A Dynamic Scheduling Framework for Emerging Heterogeneous Systems . . 39

4.1 Dynamic Scheduling Approach and Goals .......................... 40
4.1.1 Representative Processing Structures ......................... 41
4.1.2 Initial Scheme and Optimization Goals ....................... 44

4.2 Cost Model and Optimized Work Distribution Schemes ............. 45
4.2.1 Cost Model for Choosing Number of Chunks ................. 45
4.2.2 Optimized Work Distribution Schemes ......................... 48

4.3 Experimental Results ............................................... 50
4.3.1 Validation of Accuracy of Cost Model ......................... 53
4.3.2 Performance Gains from Heterogeneous Configurations ....... 55
4.3.3 Overhead of Dynamic Distribution Schemes .................. 57
4.3.4 Analysis for Benefits from Optimized Dynamic Schemes .... 59

4.4 Summary ............................................................. 61

5. Supporting GPU Sharing in Cloud Environments .................... 66

5.1 Motivation .......................................................... 67
5.1.1 Background on GPU and CUDA Mapping and Scheduling Mechanisms 67
5.1.2 GPU Sharing by Workload Consolidation ...................... 68

5.2 Software Framework Design ........................................ 70
5.2.1 Design Challenges ............................................... 71
5.2.2 gVirtuS Current Design ......................................... 72
5.2.3 Our Runtime Consolidation Framework ....................... 73
5.2.4 Design Issues and Limitations ................................. 76

5.3 Consolidation Policies and Algorithm ............................. 77
5.3.1 GPU Sharing Mechanisms ....................................... 77
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.2</td>
<td>Causes of Resource Contention</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>5.3.3</td>
<td>Advanced Policies with Molding</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>5.3.4</td>
<td>Runtime Consolidation Algorithm</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Experimental Results</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>5.4.1</td>
<td>Evaluation Methodology</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>5.4.2</td>
<td>Evaluation</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>Summary</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>Scheduling Concurrent Applications on a Cluster of CPU-GPU Nodes</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>System Model and Challenges</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>6.1.1</td>
<td>System Model</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>6.1.2</td>
<td>Problem Formulation and Challenges</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>Scheduling Policies</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>6.2.1</td>
<td>Scheduling of Single Node Jobs</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>6.2.2</td>
<td>Scheduling of Multi-Node Jobs</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>6.3</td>
<td>Experimental Results</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>6.3.1</td>
<td>Experimental Setup</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>6.3.2</td>
<td>Benchmarks</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>6.3.3</td>
<td>Experiments on Single-Node Applications</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>6.3.4</td>
<td>Experimental Results on Multi-node Applications</td>
<td>117</td>
<td></td>
</tr>
<tr>
<td>6.3.5</td>
<td>Additional Considerations</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>Conclusions</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>Value-Based Scheduling Framework for Modern Heterogeneous Clusters</td>
<td>122</td>
<td></td>
</tr>
<tr>
<td>7.1</td>
<td>Problem Definition</td>
<td>124</td>
<td></td>
</tr>
<tr>
<td>7.1.1</td>
<td>Value Function</td>
<td>124</td>
<td></td>
</tr>
<tr>
<td>7.1.2</td>
<td>Problem Formulation</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>Scheduling Heuristics</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>7.2.1</td>
<td>Overall Approach</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>7.2.2</td>
<td>Initial Placement and Ordering</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>7.2.3</td>
<td>Scheduling Schemes</td>
<td>131</td>
<td></td>
</tr>
<tr>
<td>7.2.4</td>
<td>Heuristics for Sharing</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Software Framework Design</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>7.3.1</td>
<td>High-Level Scheduler Framework Design</td>
<td>139</td>
<td></td>
</tr>
<tr>
<td>7.3.2</td>
<td>GPU Sharing Framework</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>7.4</td>
<td>Experimental Results</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>7.4.1</td>
<td>Evaluation Methodology</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>7.4.2</td>
<td>Comparison with TORQUE and MCT</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>7.4.3</td>
<td>Benefit from Enabling Sharing</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>7.4.4</td>
<td>Overhead of Sharing a CPU Core</td>
<td>151</td>
<td></td>
</tr>
</tbody>
</table>
7.4.5 Summary of Results ........................................... 151
7.5 Conclusions ...................................................... 152

8. Conclusions and Future Work .................................. 154

Bibliography ......................................................... 158
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 % Idle with Uniform Chunk Size</td>
<td>34</td>
</tr>
<tr>
<td>4.1 Sequential Execution Time on CPU (ENV1)</td>
<td>51</td>
</tr>
<tr>
<td>4.2 Sequential Execution Time on CPU (ENV2)</td>
<td>51</td>
</tr>
<tr>
<td>4.3 Benefit of NUCS over UCS</td>
<td>59</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A Popular Heterogeneous Computing Platform</td>
<td>2</td>
</tr>
<tr>
<td>3.1</td>
<td>Generalized Reduction Processing Structure</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>High-level Architecture of the System</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Uniform Chunk-Size Distribution Scheme</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Non-Uniform Chunk-Size Distribution Scheme</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Scalability of K-Means with CPU-only and GPU-only</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Scalability of PCA with CPU-only and GPU-only</td>
<td>28</td>
</tr>
<tr>
<td>3.7</td>
<td>K-Means Using Heterogeneous Version with Uniform Chunk Size</td>
<td>31</td>
</tr>
<tr>
<td>3.8</td>
<td>K-Means Using Heterogeneous Version with Non-Uniform Chunk Size</td>
<td>31</td>
</tr>
<tr>
<td>3.9</td>
<td>PCA Using Heterogeneous Version with Uniform Chunk Size</td>
<td>32</td>
</tr>
<tr>
<td>3.10</td>
<td>PCA Using Heterogeneous Version with Non-Uniform Chunk Size</td>
<td>32</td>
</tr>
<tr>
<td>3.11</td>
<td>Work Distribution (K-Means)</td>
<td>35</td>
</tr>
<tr>
<td>3.12</td>
<td>Work Distribution (PCA)</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>Generalized Reduction Processing for a Chunklet</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Structured Grid Pattern</td>
<td>42</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>4.3</td>
<td>Structured Grid Computation Performing Local and Global Updates for a Chunklet</td>
<td>43</td>
</tr>
<tr>
<td>4.4</td>
<td>Idle Time and Device Invocation Overheads with Varying Chunk-Sizes</td>
<td>46</td>
</tr>
<tr>
<td>4.5</td>
<td>Validation of Cost Model</td>
<td>52</td>
</tr>
<tr>
<td>4.6</td>
<td>Performance of CPU-only, GPU-only, and Heterogeneous Configuration</td>
<td>54</td>
</tr>
<tr>
<td>4.7</td>
<td>Overhead of Non-Uniform-Chunk Distribution Scheme Against Static Schemes</td>
<td>63</td>
</tr>
<tr>
<td>4.8</td>
<td>Overhead of Two-Level Hybrid Distribution Scheme Against Static Schemes</td>
<td>64</td>
</tr>
<tr>
<td>4.9</td>
<td>Jacobi - Benefits from Optimized Distribution Scheme</td>
<td>65</td>
</tr>
<tr>
<td>5.1</td>
<td>Scalability of Applications with Increasing Thread Blocks</td>
<td>69</td>
</tr>
<tr>
<td>5.2</td>
<td>Throughput of Consolidated Applications</td>
<td>70</td>
</tr>
<tr>
<td>5.3</td>
<td>gVirtuS System Architecture</td>
<td>72</td>
</tr>
<tr>
<td>5.4</td>
<td>High-Level Consolidation Framework Design</td>
<td>74</td>
</tr>
<tr>
<td>5.5</td>
<td>Benchmarks and its Memory Characteristics</td>
<td>86</td>
</tr>
<tr>
<td>5.6</td>
<td>Dataset Description for Benchmarks</td>
<td>88</td>
</tr>
<tr>
<td>5.7</td>
<td>Throughput Benefits from Space Sharing</td>
<td>89</td>
</tr>
<tr>
<td>5.8</td>
<td>Throughput Benefits from Time Sharing</td>
<td>89</td>
</tr>
<tr>
<td>5.9</td>
<td>Drawbacks of Basic Policies</td>
<td>90</td>
</tr>
<tr>
<td>5.10</td>
<td>Impact of Large Threads on Throughput</td>
<td>91</td>
</tr>
<tr>
<td>5.11</td>
<td>Effect of Affinity Scores on Throughput</td>
<td>92</td>
</tr>
<tr>
<td>5.12</td>
<td>Impact of Molding on Large Threads - Time Sharing with Reduced Threads</td>
<td>93</td>
</tr>
<tr>
<td>5.13</td>
<td>Impact of Molding on Large Threads - Time Sharing with Reduced Threads and Forced Space Sharing</td>
<td>93</td>
</tr>
<tr>
<td>5.14</td>
<td>Impact of Molding on Shared Memory Contention using Forced Space Sharing</td>
<td>93</td>
</tr>
</tbody>
</table>
5.15 Impact of Molding on Per Application Slowdown ........................................ 95
5.16 Choice of Molding Type for Workloads ....................................................... 95
5.17 Impact of High Contention on Throughput .................................................... 96
5.18 Framework Overhead - No Consolidation .................................................... 98
5.19 Framework Overhead - Due to Consolidation ............................................... 98
6.1 Single-node Benchmark Applications Characteristics - Large Datasets ........ 113
6.2 Multi-node Benchmark Applications Characteristics ...................................... 113
6.3 System Throughput and Latency for Uniform CPU/GPU Workload ............. 115
6.4 System Throughput and Latency for CPU Biased Workload ....................... 115
6.5 System Throughput and Latency for GPU Biased Workload ....................... 115
6.6 System Throughput for Multi-node Jobs with Varying Execution Lengths .... 118
6.7 System Throughput for Multi-node Jobs with Varying Resource Request Size 119
6.8 Penalty from Simultaneous Execution of CPU- and GPU-only jobs on the Same Set of Nodes 119
7.1 An Example Value Function (Linear Decay Slope) ....................................... 124
7.2 Scheduler Framework Design and its Components ......................................... 138
7.3 Benchmarks Characteristics .......................................................................... 144
7.4 Comparison of Scheduling Schemes Based on Completion Time and Latency (Lower is Better) 144
7.5 Impact of Varying CPU-GPU Job Mix on Yield (Higher is Better) ............ 147
7.6 Impact of Different Value Decay Functions on Yield (Higher is Better) .... 148
7.7 Impact of Increasing Load on Yield (Higher is better) ................................ 148
7.8 Impact of Varying K-Factor on Sharing (Higher is Better) . . . . . . . . . . . 150

7.9 Benefit from Sharing with Increasing Load (Higher is Better) . . . . . . . . 151

7.10 Overhead of Sharing One CPU Core Between a CPU and a GPU Job . . . . 152
Chapter 1: Introduction

1.1 Motivation

The traditional method to improve processor performance, i.e., by increasing clock frequencies, has become physically infeasible. To help offset this limitation, multi-core CPU and many-core GPU architectures have emerged as a cost-effective means for scaling performance. On the one hand, a large body of research has been focused on the effective utilization of multi-core CPUs. For instance, library support and programming models are currently being developed for efficient programming on multi-core platforms [27, 74]. On the other hand, scientists have also been seeking ways to unleash the power of the GPU for general-purpose computing [13, 79, 46, 89].

An emerging trend that is now beginning to receive attention [67, 96] is of increasingly heterogeneous computing platforms. It has become common for today’s desktops and notebooks to have both multi-core CPUs and GPU(s). Figure 1.1 shows such an architecture. Increasingly, there is also evidence that such a trend is likely to continue. Examples of heterogeneous systems are many, and include Cray supercomputers (with their vector processing units), IBM Cell Processor (use of SPEs), Cray XD1 (use of FPGAs), and others.

Today it is extremely common for desktops, laptops, clusters, supercomputers, and cloud environments to include both multi-core CPUs and GPUs. For instance, 3 out of
the top 5 supercomputers (as of November 2011) [8] use both CPUs and Nvidia GPUs. Continuing this trend, cloud environments have also started to embrace the presence of heterogeneous resources. For example, Amazon, a successful cloud provider, and Nim-bix \(^1\) introduced GPU instances in 2010 [1] to attract more HPC users. While CPU-GPU clusters are the most popular heterogeneous systems today, other processor architectures are also emerging. Intel has recently announced its Many Integrated Core (MIC) architecture \(^2\), which is likely to be released in late 2012. Accelerated Processing Units (APUs) that integrate CPU and GPU cores into a single die represent yet another flavor of heterogeneous processors. The AMD Fusion and the Intel Sandy Bridge are two popular instances of APUs. As an example of how heterogeneity will feature in future high-end systems, consider the recently announced Stampede system [7], which will have peak performance of more than 10 petaflops. As per the current plan, the Stampede system will include Intel Sandy Bridge processors, with Intel MIC co-processors, and a number of NVIDIA GPUs. Overall, it is clear that notebooks, desktops, clusters, supercomputers, as well as the cloud

\(^1\)Please see:http://www.nimbix.net/
environments targeting HPC workloads are increasingly involving more heterogeneous resources.

While the complexity of such architectures have been steadily increasing, the lack of desired software support (or mechanisms) to harness such hardware impacts both the productivity of the developer and the efficiency of the applications, negatively. In this thesis, we have developed a number of system softwares aimed at both improving the productivity as well as the performance of the applications. We first briefly describe the list of problems that we address in this thesis.

• First, we address the problem of accelerating the application performance by simultaneously utilizing the aggregate power of CPU and GPU within a node. To achieve this, we provide a high-level API and runtime support to handle the concurrency control as well as dynamic work distribution between CPU and GPU.

• Next, we address the problem of efficient GPU utilization in shared cloud environments due to the lack of support for GPU sharing across different applications. Towards this end, we develop a framework with software mechanisms to enable GPU sharing.

• Finally, we revisit the problem of independent job scheduling on modern heterogeneous clusters comprising multi-core CPUs and GPUs. We have proposed a number of novel scheduling schemes with both system and market-based metrics.

1.1.1 Aggregate CPU-GPU Power and Automatic Work Distribution

In most studies scaling applications on modern multi-core CPUs, the possibility of using the GPU to further accelerate the performance has not been considered. Similarly,
in studies involving GPGPUs, the multi-core with a very high peak performance remains idle. Effectively exploiting the aggregate computing power of a heterogeneous architecture comprising the multi-core CPU and a GPU is one of the most critical challenges facing high-performance computing. Thus, programming heterogeneous machines poses many challenges for application development [47].

On many systems, the challenge is identifying the processing element suitable for each task within the computation. For example, operations that can be vectorized must be mapped to a vector processing unit for better performance [63]. Recently, researchers have studied how tasks from within a certain application can be chosen for execution on either the multi-core CPU or the GPU [90]. Many recent studies have shown that GPUs are well suited for accelerating a variety of general purpose computations [13, 79, 46, 89], with up to 50 or 100 times performance improvements over a single threaded CPU code reported in many cases. At the same time, other studies show that highly tuned multi-threaded CPU implementations can also be quite competitive [10].

It is highly desirable to accelerate the same computation using the aggregate processing power of a multi-core CPU and a GPU [67, 96], flexibly benefiting from both. This is a critical problem in speeding up applications using modern desktops, as well as HPC clusters. Recently, there is increasing interest in developing applications/systems for heterogeneous configurations. For example, a recent study implemented stencil kernels over a hybrid CPU/GPU system [96]. Also, Luk et al. have proposed Qilin [67], an adaptable workload mapping scheme based on offline training.

Yet, dividing the work among heterogeneous multiprocessors is a critical challenge that has to be addressed to obtain high efficiency. Today’s CPU cores and GPU cores differ
considerably in their processing capabilities, memory availability, and communication latencies. Thus, the relative performance of a CPU and GPU varies for every application. Hence, it is important to distribute the work that is proportionate to the capabilities of CPU and GPU. It is infeasible to have a static partitioning method as it will be based on exhaustive search. Therefore, dynamic work distribution methods that can divide the work proportional to the processing abilities of CPU and GPU is desirable. However, dynamic work distribution in heterogeneous setting may involve non-trivial overheads. Thus, it is challenging to design a work distribution scheme that can efficiently distribute the work between CPU and GPU. Moreover, the relative performance of a CPU core and a GPU is not only dependent on the application characteristics, but also on the problem size and heterogeneous hardware configuration. Extensive offline training [67] or autotuning [98] could determine the appropriate distribution of workload, but this requires substantial offline work for a specific application, problem size, and hardware configuration (i.e., each combination of GPU and CPU). A dynamic distribution scheme that can assign workloads according to the given application’s characteristics (i.e., relative performance on a CPU core and the GPU) could help exploit heterogeneous cores more effectively. Such a scheme would also reduce the programmer’s effort in explicitly partitioning the data, and invoking kernel functions on GPUs.

1.1.2 GPU Sharing in the Cloud Environments

Next, the emergence of GPUs as a major player in high performance computing [13, 14, 67, 78, 89, 91], and the rapidly growing popularity of cloud environments, including their use for high performance applications [28, 60, 66, 95] is very apparent in the recent years. The peak single-precision performance of a Nvidia Fermi card today is more
than 1 Teraflop, giving a price to performance ratio of $2-4 per Giga-flop. High performance computing has traditionally been performed using resources on supercomputing centers and/or various local clusters maintained by organizations. However, rapid changes in HPC architectures and the need for specialized installation, maintenance, and support skills are becoming formidable challenges in acquiring and operating these resources. The “pay-as-you-go” model introduced by cloud computing has gained immense popularity recently. Multiple cloud providers are now specifically targeting HPC users and applications. Though initial configurations offered by the cloud providers were not very suited for traditional tightly-coupled HPC applications (typically because they did not use high performance interconnects), this has been changing recently. In November 2010, Mellanox and Beijing Computing Center have announced a public cloud that will be based on 40 Gb/s Infiniband. Amazon, probably the single largest cloud service provider today, announced Cluster Compute Instances for HPC in July 2010. This allows up to a factor of 10 better network performance as compared to a regular collection of EC2 instances, and an overall application speedup of 8.5 on a “comprehensive benchmark suite”\(^2\). It was argued that GPU acceleration on cloud will be a natural evolution of HPC in the cloud\(^3\), and indeed, in November 2010, Amazon announced the availability of GPU instances as part of the available cloud services. At least one other vendor, Nimbix\(^4\), is also offering GPUs as part of the cloud.

\(^2\)Please see www.hpcinthecloud.com/offthewire/Amazon-Introduces-Cluster-Compute-Instances-for-HPC-on-EC2-98321019.html

\(^3\)Please see www.hpcinthecloud.com/features/GPU-Acceleration-for-the-On-Demand-Era-103974239.html

\(^4\)http://www.nimbix.net
However, the goals in a cloud environment can be very different. Here, the important factor is to improve the overall throughput as opposed to just accelerating a single application. This is because, the cloud providers strive to increase the resource utilization and a common mechanism used to improve the resource utilization is by allowing the workloads to share the resource. One way of achieving this is through consolidation [84], [100]. However, on a cloud environment with heterogeneous resources like multi-core CPU and GPU, such sharing is non-trivial. The problem of how to share a CPU among different workloads is a well-studied and generally, memory-intensive and compute-intensive applications can share the CPU to get improve the utilization [84], [100]. Cloud environments are based on virtualization and sharing of resources. Till recently, a GPU could not be accessed from an application executing on top of a virtual machine. Several projects have addressed this [41, 34, 43, 83]. However, the challenge of making GPUs a shared resource in the cloud has not yet been addressed. The motivation for sharing GPUs in a cloud environment is multi-fold. First, on a multi-core node with a single GPU, several VMs (4-8) may be sharing a single GPU. Second, despite a very favorable price/performance ratio, a high-end GPU is an expensive resource. For example, a Fermi GPU with 6 GB memory costs nearly $4,000, possibly even higher than a multi-core CPU. Sharing of resources and obtaining high throughput is the basis for cost-effectiveness of clouds, and this applies to GPUs as well. Finally, a single high-end GPU today involves a high degree of parallelism, and many CUDA programs cannot individually fully utilize the card. Thus, it is desirable to understand and device systematic method for sharing GPUs in the cloud environments.
1.1.3 Scheduling to Improve Global Throughput on CPU-GPU Clusters

Heterogeneous systems are receiving much attention from the parallel software community, although mostly from the programmability and application portability view-point. Despite much interest in CPU-GPU clusters, some key scheduling challenges associated with them have not yet received much attention. Even though supercomputers are often rated by the peak performance achieved on a single application, in practice they are concurrently used by a number of independent jobs. In such cases, a critical component of the software stack is a job scheduler, which aims at improving the resource utilization and maximizing the aggregate throughput. In such shared clusters or cloud environments, better utilization of resources and overall system throughput are the important considerations, as opposed to the need for scaling a single application. We consider a cluster where each node has a multi-core CPU and a GPU. Our goal is to accelerate a set of applications using the aggregate set of resources in the cluster. Particularly, we formulate two distinct scheduling problems. Our first scheduling formulation (single-node jobs), views a cluster as having two sets of independent resources (CPU and GPU), and each job can be mapped to any one resource. Such mapping needs to be performed with the goal of maximizing overall throughput, while keeping the latency of each application at a reasonable level. In the second formulation (multi-node jobs), an application can either use the GPU, the multi-core CPU, or both, on any number of nodes in the cluster. Thus, the same job can be executed on a cluster of GPUs, or a cluster of CPUs, or both. While there is very limited amount of work on developing applications with these properties today, it is clearly desirable to have such flexibility in the future.
The two scheduling problems we have defined above are challenging for a number of reasons. The first factor is that CPUs and GPUs have different processing capabilities, and as a result, each workload performs differently on multi-core CPUs and GPUs [58]. Moreover, the relative performance of a workload on CPU and GPU can vary with the problem size. Similarly, different applications may scale differently with increasing number of nodes. Thus, there are very interesting trade-offs with respect to scheduling an application on the resource where it will perform better, as compared to scheduling it on the resource that becomes available sooner. Additionally, while overall system throughput is important, one cannot ignore the latency of each application from the point of submission to its completion. Earlier research in scheduling has addressed concurrent parallel job scheduling on supercomputers and clusters [36, 35, 71, 81, 80], and on chip multi-processors with different computing cores [92, 15]. However, these works were proposed either for homogeneous distributed systems or did not involve heterogeneity introduced by the presence of CPUs and GPUs within a machine.

1.1.4 Market-based Job Scheduling on CPU-GPU Clusters

Most large supercomputing centers allocate a given number of units to each user or research group and these units are consumed as the users’ jobs are executed. Cloud environments, on the other hand, have a notion of pay-as-you-go model, where resources are viewed as services. Apart from global throughput, user satisfaction and/or the priority across users can be important considerations for both supercomputing centers and cloud environments. Job scheduling in supercomputers has been extensively studied over the years [37, 69, 80, 81, 88, 93]. However, we believe that it is compelling to revisit scheduling problems in the context of today’s modern heterogeneous clusters (including multi-core
CPUs, many-core GPUs, and others), while factoring user satisfaction, profitability of the service providers, and priority across different user types.

TORQUE [9] is an open-source batch scheduler for today’s heterogeneous (CPU-GPU) clusters, and is widely used in hundreds of supercomputer centers. We claim that it lacks three important features that are highly desirable today:

(i) First, scheduling schemes within TORQUE are designed with the goal to improve system-wide metrics like the global throughput. In an emerging market-based service world, the users and jobs are becoming more classified. For instance, one user could be more valuable than the other, and one job could be more urgent than the other. We believe that scheduling schemes that can maximize the service providers’ profit and/or improve user satisfaction are useful.

(ii) Next, the type of resource (CPU or GPU) to execute a job is currently decided by the user (as configured through the qsub command). While this was a reasonable model when there was no portability across CPUs and GPUs, emergence of OpenCL [59] and compilers like PGI CUDA-x86 [5] are now enabling portability across different devices. Such portability is not exploited by TORQUE. Particularly, when contention on one resource (say GPU) can be much higher than on the other (say multi-core CPU), the uneven resource demands can adversely affect utilization, and user response times. Scheduling schemes that can automatically assign either CPU or GPU to the jobs, exploiting the portability of applications, and taking system load into account, are clearly desirable.

(iii) Finally, today GPUs are massively parallel (about 512 cores), and running only one application at a time can sometimes under-utilize the GPU. Hence, it is often preferable to allow GPU sharing among multiple jobs. Currently, TORQUE neither has the mechanism nor the schemes to automatically allow GPU sharing.
1.2 Contributions

While we discuss in depth the contributions of this thesis in the next few chapters, a summary of it is given here.

- We first developed a framework and runtime support targeting a particular class of applications (characterized by generalized reductions) for a CPU-GPU heterogeneous configuration. Starting from a high-level API, our runtime system automatically maps the computations to the heterogeneous processing elements and can utilize the aggregate computation power of CPU and GPU. We also developed two dynamic work distribution schemes that can effectively realize the power of such platforms. The proposed work distribution scheme minimizes the overheads associated with dynamic scheduling in the heterogeneous setup. One key observation from this work is that, the choice of initial data chunk size is key in obtaining the best performance.

- Next, we describe a general dynamic scheduling framework for data parallel loops geared towards CPU-GPU heterogeneous machines. Our approach involves analysis of architectural trade-offs and communication patterns in the loop. Based on the analysis, we have developed a cost model for predicting the optimal chunk size, by balancing the performance critical factors. We verify that the cost model can predict the optimal initial chunk size for different applications and problem sizes. In addition to the general framework, we derive optimized instances of dynamic work distribution schemes for applications involving two distinct communication patterns, which are generalized reductions and structured grid computations. We have evaluated using four applications involving these two classes of communication patterns.
Further, we verify that our dynamic work distribution schemes incur low overhead and can adapt to different heterogeneous configurations.

- Next, we have developed a framework for transparently enabling one or more GPUs to be shared by applications executing over a number of VMs. We have both extended an existing open source software to provide mechanisms for enabling consolidation, as well as provided solutions to the conceptual problem of consolidation. Particularly, we have carefully studied the characteristics of GPU kernels that can and cannot benefit from consolidation, and based on this, have developed a method for computing the affinity score between two or more kernels, indicating the improvement in throughput when they are consolidated. We have also introduced two new molding methods, where a kernel’s execution configuration is changed to have a different number of thread blocks or a different number of threads per block. Based on the affinity score and considering the possibility of molding, we have developed an algorithm for mapping a set of given kernels on GPUs. Our evaluation demonstrates that, when there is high contention, our consolidation mechanisms achieve improved global throughput compared to an execution without consolidation.

- We have proposed a number of novel scheduling schemes to enable job scheduling on CPU-GPU clusters. For the single node jobs scheduling, we have designed three new policies. These policies vary with respect to whether they focus on using the resource where an application’s performance will be better, or on using the resource that becomes available sooner. These policies also vary with respect to the amount of information about the application that a user needs to provide to the scheduler. For the multi-node jobs, we have developed a Flexible Moldable Scheduling scheme
(FMS) that considers molding of jobs along two dimensions: resource type and the number of requested nodes.

- For value-based scheduling problem, we have proposed four novel dynamic mapping and scheduling schemes, broadly categorized as *Uncoordinated* (three schemes) and *Coordinated* (one scheme). These schemes use resource-specific queues and operate by minimizing the imbalance between them. In our coordinated scheme, we allow periodic monitoring of the state of each queue to dynamically readjust the mapping and scheduling at a finer granularity. Uncoordinated schemes lack this periodic monitoring; however, they can dynamically readjust the mapping and scheduling at a coarser-granularity. We also developed a novel scheduling heuristic to allow jobs to efficiently share multi-core CPUs and GPUs. In particular, we observe that some jobs are less urgent, and/or exhibit performances that do not scale well beyond a certain number of cores, and hence are preferred candidates for sharing without incurring significant penalties.

The remainder of this thesis is organized as follows. In chapter 2, we study the related efforts in the literature. In chapter 3, we describe a framework and runtime support for enabling generalized reductions on heterogeneous configurations. In chapter 4, we describe a scheduling framework for data parallel loops on emerging heterogeneous configurations. In chapter 5, we describe a transparent framework for sharing the GPUs in the cloud environments. In chapter 6, we elaborate on the scheduling schemes for concurrent job scheduling on a cluster of CPU-GPU nodes with global throughput as the primary goal. In chapter 7, we study the value-based scheduling schemes for CPU-GPU clusters, and conclude in chapter 8.
Chapter 2: Related Work

We now compare our work against related efforts in programming models and runtime support for application development on heterogeneous architectures. We also compare our work with efforts on GPU virtualization, workload consolidation on GPU, and job scheduling on homogeneous and heterogeneous distributed systems.

2.1 Programming Models and Runtime Frameworks

Open Computing Language (OpenCL) [59] is a C-based framework for heterogeneous programming. It enables data parallel programming on CPU, GPU, and other devices. OpenCL, while promising, is still in development. Exochi [97] and Merge [65] provide both a programming model and a runtime framework for transparently utilizing heterogeneous resources. However, neither of them account for the performance-critical factors that we consider. Harmony [29] involves an execution model and a runtime framework to schedule computations either on a CPU or on a GPU, based on the estimated kernel performance. Teodoro et al. [90] describe a runtime framework that selects either of a CPU core or the GPU for a particular task. In comparison to [29] and [90], we consider data parallel programs and utilize both CPU and GPU simultaneously.

In contrast, we presented a framework and runtime support for generalized reductions for utilizing the aggregate power of CPU and GPU in a machine. In the context of emerging
heterogeneous systems, our distinct contributions of this paper are, 1) a general framework that transparently supports various communication patterns, 2) a cost model that is effective in predicting the optimal chunk size, by balancing performance-critical trade-offs, and 3) an optimized work distribution schemes that is designed to achieve improved performance for distinct communication patterns.

2.2 Dynamic Scheduling between CPU and GPU

Recently, Qilin system [67] has been developed with an adaptable scheme for mapping the computation between CPU and GPU simultaneously. The adaptivity of the system is based on extensive offline training and does not provide mechanisms to minimize GPU device overheads. Our work, in comparison, is based on dynamic work distribution that does not require any offline training. We also consider several architectural trade-offs like device invocation and idle time to improve the performance. Becchi et al. [14] use a data-aware scheduling of kernels that reduces the data transfer overhead due to GPU. However, this system is based on profiling and does not use the heterogeneous resources simultaneously.

Prior to the recent development of multi-core CPUs and GPUs, many techniques were proposed for scheduling data parallel loops. Guided self-scheduling scheme [73] initially allocates larger chunk size to reduce scheduling overhead, and subsequently reduces the chunk size towards the end of computation. Several variations of the guided self-scheduling were developed [49, 94, 70], each with a different strategy for reducing chunk size. SASH [44] is a dynamic scheduling technique that adapts on-line based on the workload. Ciorba et al. [23] describe a dynamic scheduling technique for loops with dependence in their iterations. None of the above works is designed to handle the disparities in heterogeneous configuration involving today’s multi-core CPU and GPU. The scheduling
overheads for today’s heterogeneous systems are very different. To this end, we provide a scheduling framework with cost model and optimized work distribution schemes that carefully minimizes the overheads geared towards heterogeneity involved with multi-core CPU and GPU.

### 2.3 GPU Virtualization and Workload Consolidation on GPU

gVirtuS [41], GViM [43], vCUDA [83] and rCUDA [34] use API Remoting to provide GPU *visibility* from within virtual machines. GViM and vCuda leverage the multiplexing mechanism provided by the CUDA runtime in order to allow GPU sharing among different applications. In addition, GViM uses a Working Queue per GPU in order to evenly distribute the load across different GPUs. However, none of these frameworks consider GPU kernel consolidation as a mechanism to improve the overall system utilization and provide throughput benefits.

GViM focuses on minimizing 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 the data copy 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. Memory transfer optimization is orthogonal to the objectives of our work. Some of these memory optimizations have been adapted in gVirtuS 2.0, on top of which we have implemented our GPU kernel consolidation framework.

Another related work [42] has explored a way to enable task parallelism in the CUDA scheduler. Since the proposal was done before the advent of the Fermi architecture, it required source code level changes to GPU kernel functions. Kernel consolidation was
performed on a thread-block basis, which is conceptually analogous to what was done by
the CUDA runtime on the Fermi architecture. The authors of [42] pointed out that kernels
suitable for consolidation are those with complementary resource requirements (e.g.: a
compute intensive and a memory intensive kernel). Another work [64] uses task parallelism
to consolidate GPU workloads to achieve energy efficiency. However, this requires source
code level changes. In our work, we go a step further and distinguish space sharing and time
sharing. In addition, we introduce the concept of affinity score to allow a more fine grained
control of kernel consolidation. Finally, we propose two distinct ways to dynamically mold
the execution configuration of kernels, in order to allow efficient GPU sharing also in the
case of GPU kernels with high or conflicting resource requirements. To the best of our
knowledge, this dynamic and fine grained control is unique to our proposal.

2.4 Job Scheduling to Improve Global Throughput

Scheduling of parallel jobs in a cluster or a supercomputing center is a very widely
studied problem [36]. Other related works that enable scheduling for parallel jobs in a
distributed cluster are [35], and [71]. These efforts propose scheduling strategies that in-
volve both static and dynamic mechanisms. However, the above works focus only on the
homogeneous type of processors. Sabin *et al.* [81], and [80] proposed and evaluated job
scheduling strategies multiple sites, i.e. separate clusters with different types of processing
nodes. Our work consider a distinct type of heterogeneity, which is arising in the emerging
environments.

Torque [9], an open-source resource manager, is being widely used to manage het-
erogeneous clusters comprising multi-core CPUs and GPUs. However, the mapping and
scheduling strategy used by Torque (based on OpenPBS) is fairly simple. While waiting
for a resource, it does not consider the possibility of assigning the job to another resource type. Thus, Torque cannot exploit the flexibility that emerging frameworks like OpenCL will provide. Other related efforts [37, 69] do consider the possibility of scheduling a job onto a different resource type, but are limited in the moldability they consider. Parallel job scheduling with moldability has also been studied [24, 86]. Our work is specific to moldability in heterogeneous environments. Ahmad et al. [11] developed a scheduling scheme for jobs with fixed priority on heterogeneous computing systems, where they dynamically compute the precedence for jobs on different resources to make scheduling decisions. However, unlike our work, they consider scheduling for dependent set of tasks represented as directed acyclic graphs (DAG). Scheduling with consideration of resource heterogeneity has also been addressed in the context of grid computing [55, 56, 17, 40]. These efforts, however, differ both in targeting much longer running applications, and in the nature of systems they focus on.

2.5 Value-based Job Scheduling

Scheduling for parallel systems has been extensively studied over the past two decades [69, 80, 81, 88, 93]. Here, we restrict our discussion to the work in parallel job scheduling for heterogeneous resources, and to the work along the lines of value-based scheduling.

We first compare our work with the existing schedulers for heterogeneous resources. Torque [9], an open-source resource manager, is being widely used in hundreds of supercomputer centers to manage heterogeneous clusters comprising multi-core CPUs and GPUs. However, the mapping and scheduling strategy used by Torque has been fairly simple. Specifically, while waiting for a resource, it does not consider the possibility of assigning the job to another resource type. Thus, Torque cannot exploit the flexibility that popular
framework like OpenCL provides. Other related efforts [37, 69] do consider the possibility of scheduling a job onto a different resource type. However, these efforts [9, 37, 69, 77] consider only metrics like minimizing the makespan of the jobs being scheduled, and does not include notion of urgency or priority among the jobs.

Ahmad et al. [11] developed a scheduling scheme for jobs with fixed priority on heterogeneous computing systems, where they dynamically compute the precedence for jobs on different resources to make scheduling decisions. However, unlike our work, they consider scheduling for dependent set of tasks represented as directed acyclic graphs (DAG). Scheduling for jobs with hard deadlines [38] has also been considered in the context of heterogeneous multiprocessors. We believe that such a consideration is only realistic for real-time systems, and not for HPC jobs or systems. Our work differs from all the above works, in that, we use a value function which allows us to capture both important and urgency of the jobs. Moreover, our work considers effective ways for sharing a single resource, which has not been considered previously.

There is also an extensive literature on schedulers that consider market-based or economy-based or user-centric approach for scheduling. Millenium [22] is a user-centric scheduler based on the FirstPrice heuristic, which gives precedence for the job that gives the highest value per unit execution. Irwin et al [50] extended the work to consider FirstReward, which considers not only gains that can be obtained from scheduling the job, but also the cost involved in choosing one job over the other. Libra [82] is a budget-based proportional sharing scheduling scheme, where slack of the job and the amount of money a job is willing to pay is used. However, these efforts [22, 50, 82] consider only homogeneous resources. We have focused on heterogeneous resources, and our work has addressed several new challenges associated with heterogeneity for value-based scheduling.
A trend that has materialized, and has given rise to much attention, is of the increasingly heterogeneous computing platforms. Presently, it has become very common for a desktop or a notebook computer to come equipped with both a multi-core CPU and a GPU. Capitalizing on the maximum computational power of such architectures (i.e., by simultaneously exploiting both the multi-core CPU and the GPU) requires an efficient work distribution between CPU and GPU. However, it is a non-trivial problem due to significant overheads incurred by GPU arising from data copy and kernel invocations. This chapter describes a framework that can automatically map a class of applications, namely those characterized by generalized reductions, to a system with a multi-core CPU and GPU. The runtime system provides efficient schemes for dynamically partitioning the work between CPU cores and the GPU.

3.1 Approach and System Design

This section offers a description of the class of generalized reduction applications on which we focus. We explain an approach for mapping generalized reduction algorithms on a multi-core CPU and a GPU independently, followed by an approach for enabling hybrid execution.
3.1.1 Generalized Reduction Structure

```c
/* Outer Sequential Loop */
While (unfinished) {
    /* Reduction Loop */
    Foreach (element e) {
        (i,val) = process(e);
        Reduc(i) = Reduc(i) op val;
    }
}
```

Figure 3.1: Generalized Reduction Processing Structure

The processing structure on which we focus is summarized in Figure 4.1. The outer `while` loop is a sequential loop and controls the number of times the inner `foreach` loop is executed, typically until `unfinished` (end condition) is false. The success or failure of `unfinished` is dependent on the results obtained at the end of each iteration of the `foreach` loop. Due to this dependency, the outer `while` loop cannot be parallelized. However, we note that the `op` function is an associative and commutative function, which permits the iterations of the `foreach` loop to be performed in any order. Therefore the entire `foreach` loop is data parallel. Inside the `foreach` loop, each data instance, `e`, is processed to produce an `(i, val)` pair, where `i` is a key and `val` is the value that is to be accumulated. The data structure, `Reduc`, is referred to as the reduction object, and the `val` obtained after each processing iteration is accumulated into this reduction object based on the corresponding key. The reduction performed is, however, irregular, in the sense that which elements of the reduction object are updated depends upon the results of the processing of data instance.
In our earlier work, we had made an observation that parallel versions of many well-known data mining, OLAP, and scientific data processing algorithms share the aforementioned generalized reduction structure [53, 54]. This observation has some similarities with the motivation for the *map-reduce* paradigm that Google has developed [27].

### 3.1.2 Parallelization Approach for Multi-cores

Applications following such generalized reduction structure can be parallelized on shared memory platforms by dividing the data instances among the processing threads. A key challenge with such parallelization, however, is the potential data races with updates to the reduction object. The reduction object, shown as `Redustructure` in Figure 4.1, is shared among all the processing threads and is used to accumulate the results obtained after processing each data instance. Furthermore, in the application we target, it is not possible to statically partition the reduction object such that each processing thread updates disjoint parts of the reduction object due to processing dependencies. A simple strategy to avoid such data race is to *privatize* the reduction object. Specifically, a copy of reduction object is created and initialized for each processing thread. Each thread only updates its own copy of the reduction object. The resulting reduction objects are merged or combined at the end to obtain the final results.

In our previous work, we have developed a middleware system [53, 54] to support the parallelization of this class of applications. Operations including reading data instances from the disk, initial thread creation, replicating reduction object, and merging reduction objects are automatically performed by this middleware. This, in turn, simplifies our code generation task; our system needs only to generate code for the middleware API. Such code generation is quite straight-forward, and is not discussed in detail in this chapter.
3.1.3 GPU Computing for Generalized Reductions

GPUs support SIMD shared memory programming. Specifically, a host (driver) program is required to launch and communicate with a GPU process to execute the device function in parallel. Therefore, GPU computations involve the following three steps: (1) Copying the data block from the host (main) memory to GPU device memory, (2) launching the GPU kernel function from host and executing it, and (3) copying the computed results back from device memory to host memory.

Now, consider parallelizing generalized reduction applications on GPUs. The reduction objects are shared by all the GPU threads. Again there are potential data races, which can be avoided by replicating the reduction object for each thread. The data block is divided into small blocks such that each thread only processes one data block at a time. The results from each thread are then merged to form the final result. Finally, the merged reduction object is copied back to the host memory. Thus, three steps are required for implementing a reduction computation: (1) Read a data block, (2) compute the reduction object updates based on the data instance, and (3) write back the reduction object update.

3.1.4 System Design for a Heterogeneous Platform

Based on the approaches for parallelizing generalized reductions on a multi-core CPU and on a GPU, as we described above, we have developed a framework for mapping these applications to a heterogeneous platform. We only give a brief description of our approach in this subsection, while key components of our system are elaborated in the later sections. A high-level design of the system is shown in Figure 3.2.

Our system takes generalized reduction applications written in C with some supplementary metadata for annotating variable information (described later). A code generation
Figure 3.2: High-level Architecture of the System
module automatically generates the middleware API code for the multi-core CPU, and CUDA code for the GPU. The code generation module is a previous work [68] developed from our group. Our runtime system creates the worker threads, including one for each CPU core and one for managing the communication with the GPU. Consistent with our mechanism for parallelizing these applications on either of CPUs or a GPU, a separate copy of the reduction object is created for each of these threads to avoid race conditions while performing the reduction computations. A key component of the runtime system is our dynamic work distribution scheme which divides the work between different threads, including the thread that manages the GPU. This module is further elaborated on in the next subsection. After the processing is finished by all the threads, the results from each thread are merged together to form a final result.

3.1.5 Dynamic Work Distribution

In a heterogeneous setting with multi-core CPU and GPU, cores have disparate processing capabilities. While their relative peak performance can be known in advance, our experiences have shown that relative performance of CPU and GPU can vary significantly across applications. This renders work distribution to be more challenging. Particularly, a static distribution scheme is not likely to be effective. Thus, our work focuses on developing dynamic distribution schemes. Within the general class of dynamic schemes, there could be two possible approaches, which are work sharing and work stealing.

Work Sharing: This is one of the classical techniques, based on a centralized approach. In general, the scheduler enqueues the work in a globally shared work list, and an idle processor consumes work from this work list [99].
**Work Stealing:** In work stealing, a private work list is maintained with each processor [16, 18]. When a processor becomes idle by finishing the work in its private queue, it searches the other busy processors for work they could steal. Work stealing is shown to be a useful technique for balancing the workload with irregular applications [25, 32] especially on very large systems where a centralized scheme may be prohibitive.

In designing our system, we have taken a work sharing approach based on several reasons. In a heterogeneous computing environment, the potential load imbalance is due to the disparity among the processing cores. In our work, we focus on data parallel applications, where the amount of computation is proportional to the amount of input data assigned to each processor. Furthermore, communication with the GPU involves high latency, and the maximum workload that can be distributed to a GPU at a time is limited by the capacity of its device memory. Because of these factors, a work stealing approach would be impractical. Considering our class of target applications and the limitations of the GPU device memory, we have designed two dynamic work distribution schemes based on work sharing.

---

**Figure 3.3:** Uniform Chunk-Size Distribution Scheme

**Figure 3.4:** Non-Uniform Chunk-Size Distribution Scheme

26
Uniform-chunk Distribution Scheme: This scheme resembles a classical master-slave model, and is shown in Figure 3.3. The entire data set is divided into chunks of uniform size. A master thread acts as a job scheduler, and uses the First Come First Served (FCFS) scheduling policy. There exists a global work list shared by all processors, into which the master enqueues work whenever there is empty space. When a process becomes idle, it simply requests work from this list. The rationale behind the FCFS policy is that, a faster worker ends up requesting more work to process when compared to the slower worker. This approach ensures that a faster worker is rewarded with more work, while a reasonable amount of work is also completed by a slower worker. By keeping the policy simple, the overhead of work distribution is amortized.

Clearly, a critical factor with the above scheme is the choice of the chunk size. Intuitively, and also supported by our experiments, we can see the following trends. In the case of the GPU, because there is a high latency for invoking kernel function and data transfer and that the processing speed is faster, it would favor a larger chunk size. However, each CPU core is slower, but has much lower latency. Thus, a smaller chunk size is preferable for CPU cores. This leads us to the following scheme:

Non-uniform-chunk Distribution Scheme: This distribution scheme is shown in Figure 3.4. The initial partition of the entire data set is still of a uniform chunk size, and we typically choose a relatively smaller chunk size. When a CPU thread requests for data, the chunk with the original size is provided. When a GPU thread requests for data, a certain number of chunks of the initial size are merged together to form a larger chunk at runtime. This chunk is then forwarded to the GPU for processing. This scheme ensures that the GPU spends most of its time actually processing the data rather than spending on transferring data and the results between host and device memory. At the same time, it
ensures proper distribution of work among CPUs. To this end, when only a small fraction of data is left for processing, the scheme does not allow a single CPU core to slow down the entire application.

3.2 Experimental Results

In this section, we report results from a number of experiments that were performed. Our experiments were conducted on an AMD Opteron 8350 machine (running Redhat Linux) with 8 CPU cores and 16 GB of main memory. This machine is also equipped with a GeForce 9800 GTX graphics card, which contains 512MB of device memory. As such, we evaluate the scalability of two applications involving generalized reductions.

Figure 3.5: Scalability of K-Means with CPU-only and GPU-only

The first application we consider is k-means. Clustering is one of the key data mining problems and k-means [51] is one of the most ubiquitous approaches. The clustering problem is as follows: Given a set of points in high-dimensional space, we want to classify these
points into $K$ groups, i.e., clusters. The proximity within this space is used as the criterion for classifying the points into these clusters. In general, k-means consists of four steps: (1) Begin with $K$ random centroids, (2) for each data instance (point), find the centroid closest to it and assign this point to the corresponding cluster. (3) With these assignments, recalculate the $K$ centroids to be the average among all the points assigned to it. (4) Repeat this entire process until the centroids converge. In our experiments, the number of clusters, $K$, was fixed at 125, and the data used for k-means, a 6.4GB file, contains a set of nearly 100 million 3-Dimensional points.

The second application is Principal Component Analysis (PCA) [72], a widely used dimensionality reduction method. This algorithm performs three passes on the data set, which represents a matrix. (1) First, the mean value of the column vectors are determined, followed by (2) the calculation of the standard deviation of column vectors. (3) In the third pass, the co-variance matrix is computed. The experiments on PCA were conducted with a data set of size 8.5 GB. The number of columns of our resulting covariance matrix, $m$, was set to 64.

To exploit the parallelism that is afforded by the generalized reduction structure, the data used in our experiments are split into smaller, independently processed “chunks.” While uniform chunk sizes are common in practice due to the simplification of data distribution, we argue that distributing non-uniform chunk sizes to corresponding computing resources could be beneficial in a heterogeneous environment. With these two applications (k-means and PCA), we wish to demonstrate the main goals with our experiments: 1) To evaluate the performance from a multi-core CPU and the GPU independently, and to study how the chunk-size impacts this performance.
2) To study performance gain that can be achieved while simultaneously exploiting both multi-core CPU and GPU, (ie., the heterogeneous version). We also provide an elaborate evaluation with two dynamic distribution schemes that were discussed in earlier section.

### 3.2.1 Scalability of CPU-only and GPU-only versions

In our initial experiment, we focus on the CPU-only version and the GPU-only version separately. For all experiments with the GPU, we report results that are obtained only from the best *thread block configuration* (after having experimented with multiple configurations). The results for k-means and PCA are presented in Figure 3.5 and Figure 3.6 respectively. For the CPU-only execution (top x-axis), the chunk size has been fixed at 12.5MB for k-means and 16MB for PCA, and we report the execution times when varying the number of worker threads from 1 to 8. In the GPU-only execution (bottom x-axis), we vary the chunk sizes distributed to the GPU. Thus, both the figures use two x-axes, the bottom x-axis represents varying GPU chunk size, while the top x-axis represents the number of CPU threads. As our preliminary experiments suggested that the chunk size did not impact the performance of the CPU-only version, we did not show results varying them.

**Results from K-Means:** In the CPU-only version, the performance with 8 threads resulted in a speedup close to 9. This superlinear performance is due to low communication overheads required by k-means, and mostly from the benefits of the additional aggregate cache memory that is available with a larger number of threads. The GPU-only version performs better with increasing chunk size. This is due to the fact that, with larger chunk sizes, fewer costly device function calls are required. Indeed, the latency associated with invoking a device function and initiating a data transfer are quite significant in GPUs. The
best performance from a GPU version is observed when the data size of each chunk is 200 MB, which is about a factor of $20 \times$ speedup relative to the 1-thread CPU version. We could not experiment with a chunk size of more than 200 MB as the GPU device memory is only 512 MB, and significant space is required for storing program data structures. Our results also show that the GPU-only version is faster than the 8-core CPU-only version by a factor of 2.2.

Results from Principle Component Analysis (PCA): With 8 threads in the CPU-only version, we are able to achieve a speedup of around 6.35. Unlike k-means, this application has substantial parallelization overheads. For the GPU-only version, we increase the data chunk size from 16 MB to 256 MB, with the latter being the upper limit on the chunk size we can use, again due to the limited GPU memory. The best performance from the GPU-only execution is obtained when the data chunk size is 256 MB, which is about a factor of 4 faster relative to the 1-thread CPU version. Interestingly, contrary to what we
observed for k-means, here, the 8-thread CPU-only version is faster than the GPU-only version by a factor of 1.75.

One observation from our experiments is that the relative performance of the CPU-only and GPU-only versions varies significantly between the two applications we have considered. We believe that this is because k-means is highly compute-intensive relative to the number of memory accesses, while in PCA, the reverse is true. Moreover, in k-means, we can exploit the shared memory to achieve high speedups on the GPU.

3.2.2 Performance of Heterogeneous Configurations

We now focus on execution with heterogeneous configurations, and particularly, we study the impact of chunk size and our dynamic work distribution schemes.

Figure 3.9: PCA Using Heterogeneous Version with Uniform Chunk Size

Figure 3.10: PCA Using Heterogeneous Version with Non-Uniform Chunk Size

Recall that we had described two dynamic work allocation schemes earlier in this chapter. In the first scheme, which we refer as uniform chunk size (UCS), each data chunk processed by the CPU and GPU workers is of the same size. In the second case, the data
chunks processed by CPU and GPU workers vary in sizes, and this scheme is referred to as non-uniform chunk size (NUCS).

**Results from K-Means:** We first discuss UCS and NUCS results for k-means, which have been juxtaposed as Figure 3.7 and Figure 3.8. The x-axis denotes the heterogeneous configuration. For instance, $1+4$ indicates the simultaneous usage of 1-GPU and 4-CPU cores. Let us focus first on Figure 3.7. To illustrate the impact of chunk size, we have varied it from 12.5 MB to 200 MB. For the configuration of $1+8$, we observe that the best performance is attained when chunk size is 100 MB. As we increase the chunk size, the performance of heterogeneous version increases, similar to what we observed previously with the GPU-only case. However, as it can be seen in this heterogeneous version, the growth in performance halts at a certain point, and then actually degrades (as in the case when chunk size = 200 MB). This is in contrast to our previous results in Figure 3.5, which showed that a chunk size of 200 MB was optimal in that independent-GPU version. Another interesting observation is that little, if any, speedup can be observed between the $1+4$ and $1+8$ configurations for the best case, i.e., when chunk size = 100 MB. Overall, with UCS, the best speedup achieved with the heterogeneous version is nearly 25 times relative to the 1-thread CPU version. Also, compared to the faster of the CPU-only and GPU-only versions, we are able to achieve a performance improvement of around 23.9%. Also, dynamic scheme does not require any prior information about the computational power ratio of different processors.

Next, we consider the use of the NUCS strategy. In Figure 3.8, there are four versions, with each corresponding to a particular chunk size assigned to the CPU threads. For the GPU, we have fixed the chunk size as 200 MB (recall that Figure 3.5 showed us that a chunk size of 200 MB was optimal for the GPU). The larger GPU chunk size is created
<table>
<thead>
<tr>
<th>K-Means</th>
<th></th>
<th>PCA</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Chunk Size (MB)</td>
<td>Idle %</td>
<td>Chunk Size (MB)</td>
<td>Idle %</td>
</tr>
<tr>
<td>100</td>
<td>35.2</td>
<td>128</td>
<td>11.3</td>
</tr>
<tr>
<td>200</td>
<td>45</td>
<td>256</td>
<td>16.9</td>
</tr>
</tbody>
</table>

Table 3.1: % Idle with Uniform Chunk Size

dynamically by merging multiple smaller chunks, irrespective of the CPU chunk size. It can be observed from the results that the version with the CPU chunk size of 25 MB tends to perform the best. In fact, the best speedup that we achieved using \(1+8\) configuration with NUCS is nearly 32 times over the 1-thread CPU version. Also, when compared to the faster of the CPU-only and GPU-only versions, we have improved the performance by 60%. In comparing the best results from UCS, we have improved the performance by about 36% – a solid indication of the benefits afforded by the NUCS approach for this application. It should also be noted that this 36% performance gain is mostly obtained by almost eliminating the time spent by fast processors waiting for slower processors to complete (idle time), which we will elaborate later.

Results from PCA: The result from UCS in PCA is shown in the Figure 3.9. The best performance on the \(1+8\) configuration is achieved using both 64 MB and 128 MB chunk size. Again, tantamount to the effects noted previously for k-means, PCA’s performance increases with growth in chunk size up to a point and then decreases. Overall, a speedup of nearly 9 using the heterogeneous version is obtained when compared to the 1-thread CPU version, and when compared to the faster of CPU-only and GPU-only version, we are able to gain 45.2% in relative performance.
Figure 3.11: Work Distribution (K-Means)

Next, the results for PCA with NUCS are shown in the Figure 3.10. Similar to k-means, we again compare the four CPU chunk size variations (between 16 MB and 128 MB). Akin to the earlier experiment, a chunk of size 256 MB is used for GPU processing in all versions (since it was determined in Figure 3.6 to be optimal for GPU-only in PCA). The version with the initial chunk size of 64 MB offers us the best performance. The best speedup that we achieved using 1+8 configuration with non-uniform chunk size is about 10.4x relative to the 1-thread CPU version. Also, it should be recounted that the base version results for PCA suggested that 8-core CPU version was faster than the GPU. And when compared to this version, we have improved the performance by 63.8%. We have furthermore improved the performance of the best results from UCS by 18.6%.

We make the following key observations behind the results from the experiments comparing UCS and NUCS. First, GPUs are faster than a single CPU core, but have a much higher latency in terms of initiating data movement. Thus, a larger chunk size helps lower
the latency for GPUs. In comparison, a lower chunk size for CPUs helps improve load balance and reduce the likelihood of any CPUs becoming idle. To further validate this observation, we measured the idle time, by which we mean that either the CPU or the GPU is idle, for increasing chunk sizes. These results are shown in Table 3.1. We show only the results from the 1+8 thread configuration with larger chunk size, since contention between CPU threads is highest here. From the table we can see that for k-means, the faster resource spends about 35% and 45% of the entire execution time waiting for the slower resource to finish at the end of the execution. Similarly for PCA, the faster resource waits for about 11% and 17% of the execution time for the slower one to finish. It should be noted that, for both the applications, the performance improvement in NUCS version over UCS version is achieved by almost eliminating this idle time.

### 3.2.3 Work Distribution

Up till now, our experiments and results mainly focused on the performance that we could achieve using the heterogeneous configuration compared to the best of the CPU-only and GPU-only versions. It is also interesting, however, to assess the fraction of the work done by each of these resources. We show this analysis in Figure 3.11 and 3.12 for k-means and PCA respectively. In both these applications, we are showing results from the UCS and NUCS schemes, with focus on the 1+8 configuration.

**Results from K-Means:** First consider the version with UCS (on the left half of Figure 3.11). For smaller chunk sizes, the CPU does majority of the work compared to the GPU. But, on the contrary, we observed in the results from base version that the GPU is much faster than the 8-core CPU-only version. This is, however, expected via our previous argument which suggested that smaller chunk sizes cause GPUs to more frequently invoke
high-latency device function calls and data transfer between host and device memory. This argument is validated as we observe that an increase in chunk size leads to a corresponding increase in the GPU’s proportion of work. We note that the best performance with UCS was obtained when the distribution of work is 70.4% and 29.6% respectively for GPU and CPU threads.

Let us now consider the NUCS version (on the right half of Figure 3.11). Here we have five versions which indicate the corresponding CPU chunk size. All the five versions achieve a work proportion close to the optimally observed distribution of 70.4% (GPU) and 29.6% (CPU). The GPU transfers a larger chunk (200 MB), effectively reducing data movement overheads. As expected from previous results, the best performance was achieved from the 25 MB chunk size for the CPU. The work proportion corresponding to this configuration was 68.75% and 31.25% for GPU and CPU, respectively.

**Results from PCA:** We saw that the base version results from PCA showed that, contrary to k-means, 8-core CPU-only version is faster than the GPU-only version. Consider, first, the UCS results (on the left half of Figure 3.12). Varying the uniform chunk size for both CPU and GPU only causes the GPU to take on slightly more work. Again, this is due to the heavy data movement nature of PCA. The best performance was obtained in this case when the distribution of work is 18% and 82% for GPU and CPU respectively. Next, we consider NUCS (on the right half of Figure 3.12). Tantamount to our observation in k-means, all five versions share very similar distribution of work. The best performance is achieved from a configuration with an initial chunk size of 64 MB and the work proportion corresponding to this configuration was 22% and 78% for GPU and CPU respectively.
3.3 Summary

We have developed a framework and runtime support targeting a particular class of applications for such a heterogeneous configuration. Our runtime system automatically maps the computations to the heterogeneous processing elements. We also discuss two dynamic work distribution schemes that can effectively realize the power of such platforms. Our experimental results show that significant performance gain can be achieved on heterogeneous configurations using effective work distribution. Our key results are as follows. For the k-means application, the heterogeneous version with 8 CPU cores and 1 GPU, achieved a speedup of about 32.09x relative to 1-thread CPU. When compared to the faster of the CPU-only and GPU-only versions, a performance gain of about 60% can be observed. For PCA, the heterogeneous version managed a speedup of about 10.4 relative to 1-thread CPU, and when compared to the best of CPU-only and GPU-only versions, we achieved a performance gain of about 63.8%.

One key observation from this work is that, the choice of initial data chunk size is key in obtaining the best performance. One of the limitations of this work is that, the framework cannot point the user to optimal initial chunk size and currently, we experiment with different chunk sizes. Next, we have limited our work to one class of applications which is characterized by the generalized reduction structure.
Chapter 4: A Dynamic Scheduling Framework for Emerging Heterogeneous Systems

In this chapter, we first present a general framework for dynamic scheduling of data parallel loops on heterogeneous systems. We analyze tradeoffs in today’s heterogeneous architectures and computation pattern of the application. Consequently, we infer performance critical constraints and factor them in making scheduling decisions. We identify Chunk Size as a key factor that can significantly impact the performance in a heterogeneous setup. Consequently, we develop a general cost model to predict the optimal chunk size for different applications, problem sizes and heterogeneous configurations. Starting from above general framework and cost model, we derive optimized work distributions schemes specific to each communication pattern as part of our transparent runtime framework.

In the previous chapter, we presented a runtime framework for generalized reductions. In the context of emerging heterogeneous systems, our distinct contributions from this chapter are, 1) a general framework that transparently supports various communication patterns, 2) a cost model that is effective in predicting the optimal chunk size, by balancing performance-critical trade-offs, and 3) an optimized work distribution scheme that is designed to achieve improved performance for structured grid computations.
4.1 Dynamic Scheduling Approach and Goals

Our general approach for designing a dynamic distribution scheme on a heterogeneous configuration is based on two important factors, which are the architectural trade-offs between the heterogeneous processing elements and the underlying communication pattern in the application that is being scaled on the heterogeneous environment. Based on this analysis, we infer key constraints and factor them into scheduling decisions.

First, consider the architectural trade-offs in the heterogeneous environments we are considering. GPUs have high peak computational power, but may have a relatively smaller amount of device memory. The latency of data transfer between host memory and device memory can be very high, and every device call incurs a significant overhead. Thus, the number of device calls should be minimized, within the constraints of the device memory for holding the data associated with the particular invoked function. Multi-core CPUs, on the other hand, have lower peak performance when compared to GPUs, but have much larger memory. Moreover, invoking new functions or assigning a new data chunk for processing has a much lower latency. Moreover, the disparity in processing capabilities of CPU and GPU can cause idle time towards the end of processing.

The second factor we consider is the computation patterns. For this purpose, we consider the computation after dividing the input dataset into a large number of chunklets. Since, the computation patterns we discuss are data-parallel, each chunklet can be scheduled independently on any processing unit. However, every processing on a chunklet involves local updates i.e. updates on output elements that are updated only by this chunklet, and/or global updates i.e., elements that can be updated by iterations from other chunklets as well.
We only consider cases where the updates to global locations is performed using an associative and commutative operation. This is true for *foreach* or *doall* loops, which we target. Therefore, these loops can be parallelized by privatizing these global elements for each chunklet they are updated by. Through the information available at compile-time, some of these global elements can possibly be updated through the processing in any of the chunklets, whereas, some of the other global elements can be updated by only a small subset of chunklets. In the first case, the space for the global update needs to be allocated for processing associated with each chunklet. In the latter case, the space may only be allocated on the processes that may actually update this element. This is further illustrated through examples in the next subsection.

### 4.1.1 Representative Processing Structures

```plaintext
Foreach (element e in Chunklet) {
    (i,val) = process(e);
    Reduc(i) = Reduc(i) op val;
}
```

Figure 4.1: Generalized Reduction Processing for a Chunklet

To explain how we capture communication patterns, we use two representative examples. The processing structure of a generalized reduction loop is summarized in Figure 4.1. Note that the *op* function is an associative and commutative function, which permits the iterations of the *foreach* loop to be performed in any order. Therefore, the entire *foreach* loop is data parallel. Inside the *foreach* loop, each data instance, *e*, is processed to produce an *(i, val)* pair, where *i* is a key and *val* is the value that is to be accumulated. The shared
data structure, *Reduc*, is referred to as the *reduction object*, and the *val* obtained after each processing iteration is accumulated into this reduction object based on the corresponding *key*. The reduction performed is, however, *irregular*, in the sense that the elements updated into the reduction object depends upon the results of the processing from a chunklet.

```plaintext
for(int i = 1 to num_rows_chunklet) {
    for(int j = 1 to y-1) {
    }
}
```

**Figure 4.2: Structured Grid Pattern**

Now, we consider another computation pattern, that involves *Structured Grid computations*. Stencil kernels are instances that very well represent structured grid problems [26]. This class of applications is mostly implemented using an iterative finite-difference technique, over a spatially dense grid, and involves performing some pattern of *nearest neighbor* computations. The input problem and the output are represented in the form of matrices, where each point in the matrix is updated with weighted contributions from its neighbors. A simple example of a 2-D, 5-point stencil kernel with uniform weight contribution from its neighbors is shown in Figure 4.2.

As a simple mechanism for parallelizing this application, the input matrix can be partitioned along the rows. Now, the elements that reside on the border of two neighboring threads need to be communicated/shared among a pair of threads. In this case, thread *i* and *i* + 1, thread *i* + 1 and *i* + 2, and so on are the communicating/sharing pairs of threads, respectively.
for(int i = 1 to num_rows_chunklet ) {
    for(int j = 1 to y-1) {
        /* Local Updates */
        if( is_local_row(i) ) {
            B[i,j] += C0 * A[i,j];
            B[i+1,j] += C0 * A[i,j];
            B[i-1,j] += C0 * A[i,j];
            B[i,j+1] += C0 * A[i,j];
            B[i,j-1] += C0 * A[i,j];
        }
        else /* Global Updates */
            Reduc(offset) = Reduc(offset) op A[i,j];
    }
}

Figure 4.3: Structured Grid Computation Performing Local and Global Updates for a Chunklet

Moreover, we can rewrite such computations as a reduction, or in a data-driven processing structure. Consider the computation from the simple example shown in Figure 4.2. In this procedure, every point in the output matrix, \( B \), is computed using five points in the input matrix, \( A \). However, we can replace the computation shown in Figure 4.2 with the one shown in Figure 4.3 and still maintain correctness. Here, for each point in input matrix, \( A \), we determine the corresponding points to be updated in the output matrix, \( B \).

The computation, as stated here, involves both local (directly into the \( B \) matrix) as well as global (using \( \text{Reduc} \)) updates. As the size of final output is equal to the input data size, using a replicated reduction object to store entire result is not feasible. Therefore, local updates can be performed within the processor’s own chunklet, while the global updates can be accumulated into the Reduction Object, \( \text{Reduc} \). Also, the global updates are shared only by a subset of processing elements.
In comparison, in the generalized reduction structure, the only elements that are updated are the elements in \textit{Reduc}. Each of these updates is global, and can potentially be shared by all the processing elements. In summary, generalized reductions involve only global updates. Also, it should be noted that in generalized reduction, the desired output can be stored entirely inside the reduction object. However, in both the computation patterns, the global updates in \textit{Reduc} are combined to form the final result. We refer to this as the \textit{combination phase}.

4.1.2 Initial Scheme and Optimization Goals

Dividing the computation into chunklets and using reduction objects for storing shared updates can allow us to dynamically schedule computations on a heterogeneous setting. We can use a simple work sharing scheme. In this \textit{centralized} approach, a scheduler enqueues the work in a \textit{globally shared} work list, and an idle processor consumes work from this work list [99]. A master thread acts as a job scheduler, and uses the First Come First Served (FCFS) scheduling policy. A global work list is shared by all processors, into which the master enqueues work whenever there is empty space. When a process becomes idle, it simply requests work from this list. The rationale behind the FCFS policy is that a faster worker naturally requests more work to process when compared to the slower worker. This approach ensures that a faster worker is rewarded with more work, while a reasonable amount of work is also completed by a slower worker. We refer to this initial distribution scheme as \textit{Uniform Chunk Size} distribution scheme (UCS).

With this simple strategy, however, we may not achieve very good performance for most applications due to several reasons. A large number of chunklets, which form the unit of work, can lead to high scheduling overheads. Additionally, space allocation for global
elements and the coordination of updates to each element between different processing units can also contribute to lower performance.

Thus, we combine the chunklets into a larger unit of work, which we refer to as **chunks**.

In addition, we state a set of goals for optimization based on above analysis.

1. Ensure that there are a sufficient number of chunks to achieve a good load balance, i.e., none of the units should be idle for a substantial amount of time.

2. Ensure that the total overhead for invoking kernel function on the GPU is not large.

3. Minimize the number of global elements allocated for the entire computation.

4. Minimize the number of distinct processors that may share a global element.

### 4.2 Cost Model and Optimized Work Distribution Schemes

We now present details of our dynamic schemes, particularly, how we address the four goals we have listed above. Initially, we present a cost model to address the first two goals. Then, we present optimized distribution schemes that can further improve the performance of basic scheme. The cost model and the set of optimizations are unique to our proposal, specifically geared towards today’s heterogeneous systems.

#### 4.2.1 Cost Model for Choosing Number of Chunks

We present a cost model for choosing the number of chunks for dynamic allocation. This model focuses on achieving a balance between idle time for some of the processing units when the number of chunks is too small, and the invocation costs on GPUs when the number of chunks is too large. Figure 4.4 shows how these overheads vary as the number of chunks is varied. These results are obtained from k-means clustering with a data size.
of 6.4 GB, and on a machine with 8 AMD Opteron cores and a 9800 GTX card, however, they represent a general trend. In view of this trend, we need to determine the number of chunks for which the sum of these two overheads is minimized.

Let there be $p$ processing cores in the CPU, and we assume that there is a single GPU. Let $t$ be the time spent by a single CPU core on processing one chunk. Let the total number of chunks be denoted as $N$. $T$ is the total execution time, when the entire processing is performed by a single CPU core. Then, we have $N = T/t$. Let us assume that the GPU can process a chunk $m$ times faster than a single CPU core.

When a kernel function is invoked to process a chunk of data on a GPU, there are three types of overheads involved. The first is the latency of data transfer between the host and the GPU, a factor that is independent of the chunk size. The second is the bandwidth and chunk-size dependent data transfer cost. The third is the cost of invocation of a GPU function. Among these three factors, the second factor turns out to be a constant over the execution of the entire application, i.e., it is simply the latency-independent data transfer cost for the entire dataset. Let $d$ denote the sum of the first and third factors, for invocation and data transfer for a single chunk.
Suppose there are \( p + l \) chunks remaining in the end, where \( 1 \leq l \leq m - 1 \). \( p \) of these chunks are assigned to CPU processing cores, and the remaining \( l \) chunks are processed by the GPU. Then, the GPU will be idle for

\[
\frac{m - l}{m} \times t
\]  

(4.1)

Now, we restate \( N \) as a function of the number of iterations, \( n + 1 \), over which work was carried out by CPU cores and the GPU.

\[
N = (p + m) \times n + p + l
\]  

(4.2)

Note that there is no idle time during the first \( n \) iterations, but the last iteration has the idle time as stated through the Expression (4.1).

The Equation (4.2) can be restated as:

\[
n = \frac{N - p - l}{p + m}
\]  

(4.3)

Also, from the Equation (4.2), the total number of chunks processed by the GPU will be \( m \times n + l \). Thus, the total cost of invoking kernel functions will be

\[
d \times (m \times n + l)
\]  

(4.4)

Our goal is to minimize the sum of the idle time (Expression (4.1)) and the total cost of invoking kernel function (Expression (4.4)), i.e.

\[
\frac{m - l}{m} \times t + d \times (m \times n + l)
\]  

(4.5)

Recall that \( l \) can be vary between 1 and \( m - 1 \). Thus, as an average case, we consider \( l = m/2 \). Also, since \( t = T/N \), and further using the Expression 4.3, we have the total overhead as
\[
\frac{T}{2 \times N} + d \times m \times \frac{N}{p+m} \quad (4.6)
\]

This expression is minimized when

\[
N^2 = \frac{(p+m) \times T}{2 \times m \times d} \quad (4.7)
\]

This expression shows that the chunk-size is proportional to the square-root of the total processing time. In other words, if problem size is quadrupled for a given application, the chunk size leading to the optimal performance would increase by a factor of 2. Later, we validate our cost model.

### 4.2.2 Optimized Work Distribution Schemes

We now describe two optimized schemes that could further improve performance of dynamic scheduling.

**Non-uniform-chunk Distribution Scheme (NUCS):** Similar to UCS, the initial partition of the entire data set is still of a uniform chunk size. When a CPU thread requests for data, the chunk with the original size is provided. When a GPU thread requests for data, a certain number of chunks of the initial size are merged together to form a larger chunk at runtime. This chunk is then forwarded to the GPU for processing. This scheme ensures that the GPU spends most of its time actually processing the data rather than spending on transferring data and the results between host and device memory. At the same time, it ensures proper distribution of work among CPUs. To this end, when only a small fraction of data is left for processing, the scheme does not allow a single CPU core to slow down the entire application.

The benefit of this scheme, compared to Uniform Chunk size (UCS) scheme, is in further reducing the device invocation and transfer overhead. Thus, when \( C \) chunks are
merged before the GPU actually starts its processing, there is a potential for further reducing the device invocation overhead by a factor of $C$. Later, in Section 4.3.4, we experimentally show the advantage of this scheme compared to UCS.

**A Two-Level Hybrid Distribution Scheme:** The schemes we have presented so far have focused on only the first two factors, among the four we listed earlier in Section 4.1.2. For applications where all global updates are shared by all processing elements, only the first two goals are relevant. However, for other applications, the third and the fourth considerations can be important. Particularly, we will like to choose a strategy that maximizes local updates, i.e., minimizes the fraction of updates that are global, and furthermore, minimizes the number of distinct pairs of threads that update the same element.

To this end, we propose a two-level hybrid distribution scheme. In this scheme, we group the homogeneous CPU cores together, while GPU is considered as a separate group. Now, we dynamically partition the work between different groups (i.e., CPU and GPU), but statically partition the work within a group. This allows us to perform dynamic load balancing at first level, and at second level, significantly reduce the number of distinct pairs that update the same element. Thus, the data chunks are dynamically distributed between a group of CPU cores and a GPU. However, within a group of CPU cores or a GPU, the data chunk is equally partitioned statically. The rationale behind such a hybrid design choice is as follows.

The dynamic partitioning at the first level provides flexible and finer load balancing between disparate resources, while static partitioning at the second level (within the group) provides an efficient and tractable partitioning among homogeneous cores. The coarse-grained data decomposition minimizes global updates and hence, the volume of communication across different processing elements is minimized. Another key optimization is
provided by the choice of static partitioning within a group. For more clarity, consider the data partitioning inside the CPU data chunk. In the structured grid pattern, only a subset (two in most cases) of threads share the global updates. If the data partitioning within a group (say $p$ CPU cores) is dynamic, then, there can be $p^2$ such distinct subsets that can share the global space. This leads to at least two distinct problems. First, keeping track of which subsets share which shared row could be cumbersome. Second, this will introduce significant memory overhead due to privatization of global updates for every possible pair ($p^2$) of communicating threads. On the other hand, consider the case when the distribution within a group is static. The work slot of thread $k$ is fixed within a CPU chunk. Thus, the work slot for each thread has a total ordering, although the work slot can be allocated at arbitrary times. This reduces the number of distinct subsets to $p - 1$ and, in turn, minimizes the memory overhead due to privatization and the implementation becomes more tractable. Another observation with this scheme is that the size of data chunk processed by GPU is larger than the one processed by a CPU core (similar to NUCS scheme described previously), and hence, this implicitly lowers the device call costs.

4.3 Experimental Results

In this section, we report results from a number of experiments we performed to evaluate our framework and schemes, including the cost model we have developed. Our experiments were conducted with the following goals.

- To validate the effectiveness of our cost model in choosing the optimal number of chunks for the *Uniform Chunk Size* (UCS) scheme.
• To evaluate the performance gains that can be achieved by simultaneously exploiting both the multi-core CPU and the GPU, while using our optimized dynamic distribution schemes.

• To evaluate the performance of dynamic distribution schemes compared to a number of possible static distribution schemes.

• To further analyze the benefits of optimized dynamic schemes compared to basic scheme (UCS).

<table>
<thead>
<tr>
<th>Application</th>
<th>Execution Time (sec)</th>
<th>Dataset Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>k-means</td>
<td>353</td>
<td>6.4 GB</td>
</tr>
<tr>
<td>PCA</td>
<td>489</td>
<td>8.5 GB</td>
</tr>
<tr>
<td>Jacobi</td>
<td>6.81</td>
<td>8192 x 8192 (256 MB)</td>
</tr>
<tr>
<td>Sobel Filter</td>
<td>85.4</td>
<td>16384 x 16384 (1 GB)</td>
</tr>
</tbody>
</table>

Table 4.2: Sequential Execution Time on CPU (ENV2)

<table>
<thead>
<tr>
<th>Application</th>
<th>Execution Time (sec)</th>
<th>Dataset Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>k-means</td>
<td>310</td>
<td>6.4 GB</td>
</tr>
<tr>
<td>PCA</td>
<td>323</td>
<td>8.5 GB</td>
</tr>
<tr>
<td>Jacobi</td>
<td>12.1</td>
<td>16384 x 16384 (1 GB)</td>
</tr>
<tr>
<td>Sobel Filter</td>
<td>54.2</td>
<td>16384 x 16384 (1 GB)</td>
</tr>
</tbody>
</table>

**Experimental Setup:** Our experiments were conducted on two different heterogeneous hardware configurations. We consider the first configuration to be CPU-centric configuration, as it contains 8 CPU cores and a relatively low-end graphics card. Specifically, it
has an AMD Opteron 8350 CPU (running Redhat Linux) with 8 2.0 GHz cores and 16 GB of main memory. This machine is equipped with a GeForce 9800 GTX graphics card, which contains 512 MB of device memory. We refer to this configuration as ENV1 while presenting our results.

We consider the second configuration to be GPU-centric heterogeneous configuration, as it contains only 4 CPU cores and a high-end graphics card. This machine contains two dual core AMD Opteron CPUs (4 2.6 GHz cores) and 8 GB of main memory. This machine is equipped with a Tesla C1060 graphics card, which contains 4 GB of device memory. We refer to this configuration as ENV2. These two contrasting configurations were chosen to study the variation in application performance, and to validate the adaptability of our dynamic work distribution system.

We have chosen four applications representing two different types of computation and communication patterns. K-means clustering and PCA are applications that follow the generalized reduction pattern, whereas Jacobi and Sobel Filter both involve structured grids. The dataset sizes and the sequential execution times for these applications on a CPU are shown in Tables 4.1 and 4.2 respectively for ENV1 and ENV2.

![Different Applications](image1)

![Varying Dataset Size - Kmeans](image2)

Figure 4.5: Validation of Cost Model
4.3.1 Validation of Accuracy of Cost Model

We had earlier presented a cost model for determining the number of chunks (or the size of each chunk) that would result in the optimal performance (Section 4.2.1). In this subsection, we validate our cost model, using four applications that involve two distinct types of communication patterns. Using one of the applications, we also analyze the effectiveness of the model as the dataset size increases.

The results are shown in Figures 4.5(a) and 4.5(b). The X-axis denotes the varying number of chunks for each of the four applications or dataset sizes. In Figure 4.5(a), the dataset sizes are as indicated previously while describing the applications. Predicted implies the optimal number of chunks predicted analytically by our model. On Y-axis, we present normalized relative speedups, which are calculated as follows. First, for each application, and for each chunk number, we calculate the speedup achieved relative to sequential execution. Then, within each application, we normalize the speedup from each chunk number with the lowest speedup achieved for that application (among the chunk sizes we considered).

Consider the Equation 4.7. To apply the model, we first empirically determined the device invocation cost, $d$, which is independent of the application. The device invocation cost, $d$ was found to be 0.04 and 0.024 for ENV1 and ENV2, respectively. The other two factors, $m$ and $T$, in the Equation 4.7 were experimentally determined for each application.

From Figure 4.5(a), we can see that as an average over the four applications, there is a 53% variation in the performance achieved between the optimal chunk size and the other chunk sizes we have considered. Thus, using the appropriate chunk size is critical for obtaining high efficiency from a heterogeneous configuration. Also, we can see that for all applications, the predicted number of chunks is quite close to the number of chunks
where the best performance was achieved experimentally. Similar trends were observed with ENV2, indicating that the cost model for optimal number of chunks is adaptive to different heterogeneous configurations. The chart for ENV2 is omitted due to space limitations.

In Figure 4.5(b), we validate the model across varying problem sizes for k-means. As indicated by the Equation 4.7 the optimal number of chunks approximately doubles when the problem size is quadrupled. This is validated by the trends seen in Figure 4.5(b). Moreover, the analytical model and experimental data is aligned quite well for each of the cases.

![chart](a) ENV1

![chart](b) ENV2

Figure 4.6: Performance of CPU-only, GPU-only, and Heterogeneous Configuration
4.3.2 Performance Gains from Heterogeneous Configurations

We now evaluate the performance achieved using our optimized dynamic work distribution schemes on heterogeneous configurations. To obtain a baseline for evaluating the heterogeneous versions, we measured execution times for the applications on multi-core CPUs and GPUs separately. We refer to these versions as the CPU-only and GPU-only, respectively. For the CPU-only version, the presented results are obtained from the use of 8 cores and 4 cores on ENV1 and ENV2, respectively. For all experiments with the GPU, we report results that are obtained only from the best thread block configuration, and chunk sizes that minimize the device invocation and data transfer overheads. We also present results from the simultaneous use of the multi-core CPU and the GPU i.e., the heterogeneous configuration. We refer to heterogeneous versions as CPU+GPU, and compare it over the faster one between the CPU-only and GPU-only versions. We present the CPU+GPU results from the use of 8 CPU cores and a GPU on ENV1, and 4 CPU cores and a GPU on ENV2. The results shown for CPU+GPU are obtained from the use of optimized work distribution schemes. As k-means and PCA involve generalized reductions, we use the Non-Uniform Chunk distribution scheme (NUCS). For Jacobi and Sobel Filter (structured grids), we use the Two-level Hybrid distribution scheme. This is consistent throughout this section. The performance gains that are achieved from the use of heterogeneous configurations are shown in Figures 4.6(a) and 4.6(b).

Results from K-Means: On ENV1, both CPU-only and GPU-only achieve good scalability. CPU+GPU achieves a speedup of 32 over the 1 thread CPU version. When compared to the faster of the CPU-only and GPU-only versions, this corresponds to an improvement of about 60%. This is because, there are 8 CPU cores that can do substantial work when simultaneous work is done by the GPU. On ENV2, CPU+GPU achieves a speedup
of 21.5 relative to the 1-thread CPU version. When compared to the GPU-only version, which is the faster one between the CPU-only and GPU-only versions, the improvement is about 37.8%. This improvement is somewhat lower for ENV2, since there are only 4 CPU cores.

**Results from PCA:** On ENV1, the CPU-only version is faster than the GPU-only version. PCA involves computations spanning three iterations. Each iteration requires data transfer and the computations in the first two iterations are not intensive. Thus, GPU suffers substantial overhead. However, we can further benefit from the CPU+GPU version through optimal work distribution between CPU and GPU. The CPU+GPU version achieves a speedup of 10.4 over the 1-thread CPU version. This corresponds to a performance improvement of 63.7% when compared to the faster one of the CPU-only and GPU-only versions. In the case of ENV2, the relative performance between CPU-only and GPU-only versions are comparable. The gain in performance from the CPU+GPU version is about 47%, when compared to the faster one of the CPU-only and the GPU-only versions.

**Results from Jacobi:** The results for Jacobi is very similar to what we observed with PCA. On ENV1, the CPU-only and GPU-only results are comparable to each other. The heterogeneous version achieves a speedup of 13.8 over the 1-thread CPU version, which corresponds to a performance improvement of 75.8% over the faster one between the CPU-only and GPU-only versions. On ENV2, the performance improvement achieved with CPU+GPU is about 51.3% when compared to the faster one between the CPU-only and GPU-only versions. Thus, there are substantial benefits from exploiting the aggregate computational power of both the units.
**Results from Sobel Filter:** Sobel Filter is more compute-intensive than Jacobi, as every point in the output matrix involves a 9-point stencil computation in each dimension. On ENV1, although GPU-only version achieves a reasonably good performance, the CPU-only version with 8 cores is faster than GPU-only version. With an optimal work distribution between CPU and GPU, the CPU+GPU version achieves a performance improvement of 69.3% when compared to the faster of CPU-only and GPU-only versions. On the other hand, with ENV2, GPU-only is faster than the CPU-only version, as there are only 4 CPU cores. In this case, CPU+GPU further improves the performance of the application by 35.4% when compared to the faster baseline.

Overall, relative performance between CPU-only and GPU-only versions vary across applications and hardware configurations. For all cases shown in Figures 4.6(a) and 4.6(b), we are able to achieve from 35% up to 75% improvement from the use of heterogeneous resources. This shows that our dynamic work distribution schemes enables further improvement in the scalability of applications using heterogeneous configurations. Moreover, it is adaptable to different heterogeneous hardware configurations.

**4.3.3 Overhead of Dynamic Distribution Schemes**

To understand the effectiveness and runtime overhead incurred by the dynamic schemes, we compare them against two static work distribution schemes. We refer to these schemes as Naive-Static, and M-Opt Static (Manually Optimized), respectively. In the Naive-Static scheme, the data is partitioned equally between the group of CPU cores and the GPU. In M-Opt Static, as the name suggests, we manually find the optimal partition between the CPU and the GPU, based on autotuning or exhaustive search. For all
Figures in this subsection, the x-axis denotes the heterogeneous thread configuration. For instance, $4+1$ indicates the simultaneous use of 4-CPU cores and 1-GPU.

Note that from an application developer’s view-point, our dynamic scheme offers several advantages over $M$-$Opt$ $Static$, as it requires tuning specific to a particular application, problem size, and hardware configuration. In addition, even after finding the distribution of work, an application developer needs to manually configure the application. They also need to create appropriate chunk-size for each kernel function invocation, as the memory on most GPUs is quite limited.

**Results from NUCS Scheme:** As described earlier in 4.2.2, the NUCS scheme involves merging of smaller data chunks into a larger chunk before it is consumed by a GPU. Thus, the overhead involved with this scheme is the allocation of a larger data chunk and merging these smaller chunks. However, this in turn avoids multiple expensive data transfers between host and device memory, to improve the overall performance. The results for k-means are shown in Figures 4.7(a) and 4.7(b). As we can expect, $M$-$Opt$ $Static$ achieves the best performance in all cases. Also, as expected, $Naive$-$Static$ has a poor performance. The Dynamic scheme performs within 6% of $M$-$Opt$ $Static$. Thus, our dynamic scheme can alleviate the need for application, problem size, and hardware configuration specific tuning, while having low runtime overheads. For PCA, the results are presented in Figures 4.7(c) and 4.7(d). Dynamic scheme matches the performance of $M$-$Opt$ $Static$ well, and specifically, it is within 6.8%.

**Results from Two-Level Hybrid Scheme:** The Two-Level Hybrid scheme is also described earlier in 4.2.2. This scheme enforces total ordering in assigning chunks to CPU cores, i.e., all CPU cores must finish the processing of assigned chunk before any of them
can move to a new chunk. This enforcement introduces synchronization overhead, however, it is quite small as all CPU cores are homogeneous in nature. For Jacobi, the results are shown in Figures 4.8(a) and 4.8(b). The Dynamic scheme closely follows the performance of M-Opt Static, with runtime overheads as low as 3.7% on ENV1 and 7% on ENV2. The results from Sobel Filter are shown in Figures 4.8(c) and 4.8(d). The Dynamic scheme matches the performance of M-Opt Static closely, and the runtime overhead is within 3.2% and 5.7% for ENV1 and ENV2, respectively.

For all the four applications, and in both the heterogeneous environments, the overhead observed due to dynamic distribution of work between CPU and GPU is within 7%.

<table>
<thead>
<tr>
<th>Application</th>
<th>GPU Overhead% (UCS)</th>
<th>GPU Overhead% (NUCS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>k-means</td>
<td>33.3</td>
<td>8.3</td>
</tr>
<tr>
<td>PCA</td>
<td>24.2</td>
<td>7.6</td>
</tr>
</tbody>
</table>

4.3.4 Analysis for Benefits from Optimized Dynamic Schemes

In this subsection, we present more insights into the benefits that are obtained from the use of optimized dynamic schemes. Thus, we compare the benefits obtained from NUCS scheme and Two-level hybrid scheme against UCS scheme. All the results presented in this subsection are obtained from ENV1, however, it represents a general trend.

Optimization for Generalized Reductions: For generalized reductions, the main optimization is to minimize the data transfer overhead (for GPU), while keeping the idle time as low as possible. To meet this, in NUCS scheme, we combine some $C$ chunks before consumed by the GPU. This is done to further reduce the amount of time spent by the
GPU on data transfer between CPU and GPU. In Table 4.3, we compare the kernel invocation and data transfer overhead for GPU with UCS and NUCS scheme. To make the comparison fair, the results presented are obtained from optimal number of chunks in both cases. For both k-means and PCA, using UCS scheme, the data transfer cost accounts for a significant fraction of entire GPU processing (33.3% and 24.2% respectively). However, using NUCS scheme, the data transfer overhead on GPU has been reduced significantly by further combining the data chunks, for both k-means and PCA. This indicates that, only a small fraction of time is spent on data transfer, while the majority of GPU processing time is spent on actual computation.

**Optimization for Structured Grid Computations:** Here, the first consideration is to maximize (minimize) the local (global) updates. Another important consideration is to minimize the number of distinct subsets that update the same element(s), as a result of static partitioning within a group. We compare two versions of execution from Jacobi, which are shown in Figure 4.9. The two-level hybrid scheme performs much better than UCS, especially with larger number of threads. The reason is that, with UCS, the memory requirement increases with increasing number of threads. However, in two-level hybrid scheme, the choice of static partitioning at the second level minimizes the number of distinct subsets that update the same elements. This will reduce the memory overhead substantially and hence improved performance.

Thus, both the optimized distribution schemes (NUCS and Two-level hybrid scheme), further reduces the scheduling overheads and significantly improves the overall performance of the application in heterogeneous configuration.
Results show that our cost model can guide the user in predicting the optimal number of chunks effectively for different applications, problem sizes and heterogeneous configurations. Also, using our optimized dynamic work distribution schemes, we achieve a performance improvement from 35% up to 75% compared to the faster of CPU-only and GPU-only results. Also, we have seen that runtime overhead of dynamic schemes is within 7% of the manually tuned M-Opt Static and are adaptive to different heterogeneous configurations. Unlike the highly tuned static scheme, M-Opt-Static, our dynamic scheme does not require exhaustive manual search to find the optimal work partition. Static schemes, in comparison, require offline training specific to each application, problem size, and hardware configuration.

4.4 Summary

In this chapter, we have described a dynamic scheduling framework for data parallel loops. Our approach involves analysis of architectural trade-offs and communication patterns in the loop. Based on the analysis, we have developed a cost model for predicting the optimal chunk size, by balancing the performance critical factors. In addition to the general framework, we derive optimized instances of dynamic work distribution schemes for applications involving two distinct communication patterns, which are generalized reductions and structured grid computations. We have evaluated using four applications involving these two classes of communication patterns.

The summary of the results from our extensive evaluation is as follows. We have validated the effectiveness of our cost model in predicting optimal chunk size for various applications and problem sizes. We also show that using heterogeneous configurations, and optimized dynamic schemes, a performance improvement of up to 75.8% can be achieved.
against either of the multi-core CPU or the GPU. Also, the dynamic schemes result in a low (7% or lower) overhead compared to the best static partitioning obtained by manual tuning for the particular application, problem size, and hardware configuration.
Figure 4.7: Overhead of Non-Uniform-Chunk Distribution Scheme Against Static Schemes
Figure 4.8: Overhead of Two-Level Hybrid Distribution Scheme Against Static Schemes
Figure 4.9: Jacobi - Benefits from Optimized Distribution Scheme
Chapter 5: Supporting GPU Sharing in Cloud Environments

Driven by the emergence of GPUs as a major player in high performance computing and the rapidly growing popularity of cloud environments, GPU instances are being offered by cloud providers. In a cloud environment the important goal is to maximize the overall throughput for a set of applications as opposed to single application performance. Thus, the applications use the resources on shared basis to improve the utilization. However, the challenge of making GPU a true shared resource in the cloud has not yet been addressed. We view consolidation as a mechanism to share GPUs in the cloud and present consolidation as part of the overall scheduling problem. In this work, we focus on scheduling a set of GPU kernels in a multi-GPU environment within a node.

This chapter presents a framework to enable applications executing within virtual machines to transparently share one or more GPUs. Our contributions are twofold: we extend an open source GPU virtualization software to include efficient GPU sharing, and we propose solutions to the conceptual problem of GPU kernel consolidation. In particular, we introduce a method for computing the affinity score between two or more kernels, which provides an indication of potential performance improvements upon kernel consolidation. In addition, we explore molding as a means to achieve efficient GPU sharing also in the case of kernels with high or conflicting resource requirements. We use these concepts to develop an algorithm to efficiently map a set of kernels on a pair of GPUs.
5.1 Motivation

This section describes several motivating experiments to show that sharing of GPUs in a cloud environment is feasible and desirable. Our experiments were conducted on a machine that contains two Tesla C2050 (Fermi) cards. Before presenting the results of our experiments, we very briefly list the main architectural features of modern GPUs, and discuss how CUDA runtime schedules GPU resources.

5.1.1 Background on GPU and CUDA Mapping and Scheduling Mechanisms

A GPU comprises a set of Streaming Multiprocessors (SMs), where each SM in turn contains a set of simple in-order cores. These simple in-order cores execute the instructions in a SIMD manner. The GPU kernel is launched with an execution grid configuration that contains thread blocks and threads within each block. The number of SMs in the card we experimented with was 14.

The CUDA runtime performs mapping and scheduling decisions at the granularity of thread blocks: every thread block is mapped to a SM, and cannot be split among multiple SMs. The resource requirements (in terms of registers and shared memory) of each thread block must therefore not exceed the resource availability on a SM. CUDA runtime maps thread blocks to SMs in a round-robin fashion, and schedules their execution depending on their hardware resource requirements. If the number of thread blocks in a kernel launch is less than that of SMs, each thread block will be assigned a dedicated SM. If the number of thread blocks exceeds the number of SMs, multiple thread blocks will be mapped to the same SM. In this situation, two cases can occur. If the aggregate resource requirements (in terms of registers and shared memory) of the thread blocks mapped to the same SM do not
exceed the resource availability on the SM, then the thread blocks will be *interleaved* on the SM; else, the execution of the thread blocks will be *serialized* on the SM. Note that thread blocks interleaving will allow better memory latency hiding, potentially leading to better performance compared to the serialized case.

### 5.1.2 GPU Sharing by Workload Consolidation

We now present experiments to demonstrate the potential of consolidation as a means for improving latency and throughput in a shared GPU environment. Our experiments involved four applications: Black-Scholes, Binomial Options, Image Processing, and the PDE Solver. Among these, Black-Scholes and Binomial Options are financial applications, whereas Image Processing and PDE Solver both involve nearest-neighbor communication.

To understand the potential for consolidation, we first analyze the relationship between resource utilization and performances. To this end, we analyze the scalability of these applications as we increase the number of thread blocks (thereby increasing the number of SMs that could be used). The results from all the four applications are presented in Figure 5.1, where we show the performance improvements over using a single block of 256 threads. As we increase the number of thread blocks (for the same problem size), we can see that performance gains are significant till 8 thread blocks are in use, and beyond that performance gains are either not noticeable or sub-linear (depending on the application). However, for the Image Processing application, we get the best performance with 64 blocks. The explanations for the observed trends are as follows. Clearly, three of the applications do not exhibit enough parallelism to truly exploit all 14 available SMs. Thus, we observe a performance saturation after 8 blocks. In the case of Image Processing, not only is there enough parallelism to benefit from all SMs, we actually get better performance when the
number of thread blocks well exceeds the number of SMs. This is because of the low-overhead context switching that helps mask memory latencies in this application.

Figure 5.1: Scalability of Applications with Increasing Thread Blocks

The discussion above suggests using both space sharing and time sharing to improve throughput. Two applications that cannot each use all SMs effectively can be consolidated, such that each of them uses a subset of the available SMs (space sharing). On the other hand, two applications that have enough parallelism but also have high memory latencies can benefit from time sharing the SMs. Instead of context-switching to a thread block of the same application, latencies can also be masked by context-switching to a thread block of another application. This can be particularly valuable when applications have enough parallelism for 14 SMs, but not necessarily for a higher number of SMs.

Based on this observation, we experimented with manually consolidating kernels from different applications. In the first case, we combine Black-Scholes and Binomial Options with each kernel taking a subset of available SMs, i.e. the two kernels are space-sharing the SMs. In the next case, we combine Image Processing and PDE Solver, each using a number of thread blocks that is much larger than the available SMs, i.e., kernels are time-sharing the SMs. The results for both the cases are shown in Figure 5.2, where we report
the relative throughput benefit for kernel execution as compared to using the plain CUDA runtime scheduling (i.e., sequential execution of the GPU kernels).

From Figure 5.2 we can see that the consolidated execution of Black-Scholes and Binomial Options attains over 90% higher throughput. In addition, when executing applications with a larger number of thread blocks, we are able to efficiently time-share the SMs among different kernels. Specifically, by consolidating Image Processing and PDE Solver, we can improve the global throughput by about 51%. Overall, through these results, we demonstrate that there is a large potential for improving throughput (and thus improving resource efficiency and reducing costs) with judicious use of space sharing and time sharing, depending upon the characteristics of the applications.

![Figure 5.2: Throughput of Consolidated Applications](image)

5.2 Software Framework Design

This section describes a software framework for supporting consolidation of kernels from different applications. We start with a brief description of gVirtuS [41], which we have used in our work. Then, we discuss the design challenges involved in efficiently
sharing GPUs. Finally, we describe the framework we have implemented to enable runtime consolidation of GPU workloads.

5.2.1 Design Challenges

A runtime framework to enable efficient sharing of GPUs involves the following challenges.

How to Enable Sharing of GPU(s) across Different Applications? As we mentioned earlier, even the latest generations of GPU cards (Fermi) do not allow tasks from different applications to execute concurrently on a GPU. Concurrent execution from a single process context (application) is, however, supported. Thus, in order to enable different applications to share a GPU, we need to be able to create a virtual process context that consolidates different applications into a single application context. Note, however, this must be done without knowing in advance which applications need to be consolidated, and without any source code modifications.

What and How to Consolidate? Our experiments have demonstrated that some applications can benefit from consolidation more than others. Moreover, dynamically changing the number of threads and/or thread blocks can improve effectiveness of consolidation. However, the CUDA runtime has no a priori knowledge of the workloads that are being consolidated. This leads to the following challenges. First, given a set of applications to be executed, a software module needs to decide which of them can benefit from consolidation. Similarly, we need to be able to determine when altering the execution configuration (i.e., the number of threads and thread blocks) will be beneficial.
**How to Achieve a Low Overhead Design?** Performance is an important issue for the users of CUDA-based applications. Similarly, to ensure cost effectiveness and user satisfaction, high throughput is very important for cloud service providers. Thus, any support we develop must not introduce substantial overheads. The earlier work on GPU virtualization [41, 43, 83] has established that GPUs can be accessed from a virtual machine while incurring very low overheads. However, the additional overhead of creating and managing virtual process contexts and dynamically making consolidation decisions must also be kept low.

### 5.2.2 gVirtuS Current Design

![gVirtuS System Architecture](image_url)

**Figure 5.3: gVirtuS System Architecture**

To achieve our design goals, we leverage an existing open source tool: gVirtuS ⁵ [41]. gVirtuS uses *API remoting* to run CUDA-enabled applications within virtual machines. The software architecture of gVirtuS is shown in Figure 5.3. In particular, gVirtuS adopts

⁵[http://osl.uniparthenope.it/projects/gvirtus](http://osl.uniparthenope.it/projects/gvirtus)

72
the split-driver model and assumes the presence of a host and one or more guest machines (which can be located on the same physical node). The GPU is physically located on the host machine, whereas the CUDA-enabled applications run within the virtual machines residing on the guest machines. The CUDA runtime and device drivers are installed on the host machine where the GPU is located. Thus, a communication mechanism to redirect CUDA calls originating from the guest machines to the host machine is necessary. VM-Socket is used as a fast communication mechanism for this purpose.

gVirtuS introduces a *frontend* library on the guest machines, and runs a *backend demon* on the host machine. The backend demon forks a new *process* on the host corresponding to every CUDA application in the frontend. The frontend library acts as an *interposed* library that intercepts CUDA calls, and redirects them to the backend. The backend demon translates requests coming from the frontend into CUDA runtime library calls and issues those calls to the CUDA device driver running on the host machine. Finally, the backend demon transfers the results returned from the GPU call execution back to the frontend.

### 5.2.3 Our Runtime Consolidation Framework

We now present the overall design of our runtime consolidation framework and its components. Figure 6.1.1 shows the different components and their interaction. When compared to the entire architecture of gVirtuS, there are no modifications to the frontend library (and to the CUDA applications). Also, we do not modify the software implementing the communication channel between the guest OS and the host OS. All our modifications are on the host side. Particularly, in order to enable efficient sharing of GPUs in the cloud
through consolidation, we introduce two new components in addition to the main back-end server: the Dispatcher and Consolidation Decision Maker (DCDM), and the Virtual Context with Consolidator (VCC).

The overall end-to-end interaction between different components can be summarized as follows. As applications arrive, the backend server notifies the DCDM of the new GPU kernels. When multiple kernels are ready to be scheduled, DCDM reads the execution configuration of all the kernels. Depending on the kernels and their configuration, the DCDM performs applications-to-GPUs mapping. Thus, the DCDM enqueues CUDA calls in to the corresponding virtual context. The virtual context, in turn, consolidates and executes them on the corresponding GPU using the underlying CUDA runtime and driver. A more detailed description of the components and their functionality follows.

**Dispatcher and Consolidation Decision Maker (DCDM)** At the start of the backend daemon, we create a dispatcher thread. The first responsibility of the dispatcher thread is to read the execution configuration of the kernels that are to be scheduled on multiple...
GPUs. An execution configuration comprises of the number of thread blocks, the number of threads within each block, and the amount of shared memory requested by the kernel. This turns out to be the critical information in deciding the potential benefits of consolidating a given set of kernels, as we will explain in the next Section.

Consolidation decisions could be made with different goals, such as improving the throughput, maintaining certain priority levels among applications or users, meeting certain deadlines, and/or other considerations. In our current work, we were driven by two goals, which are improving throughput while keeping each application’s performance at a reasonable level. We provide a detailed description of policies, heuristics and the algorithm in the next section.

Virtual Context with Consolidator (VCC) Once the consolidation decisions are made, the workloads to be consolidated are dispatched to the VCC module. Prior to that, the backend demon creates a virtual context for each GPU and logically binds it to the corresponding GPU on the host machine. This is different from the original gVirtuS, where one CUDA context is created for every frontend process. Further, this virtual context can be a process or a thread. Unlike the original gVirtuS design, we use a thread for each virtual context. Two facts motivate our design. First, threads are light-weight compared to processes. Second, the procedure for setting up the interprocess communication is more involved with processes than the shared memory for threads.

Virtual contexts remain alive for the life time of the backend server and process the CUDA requests on the binding GPU. After the consolidation decisions are made, DCDM passes the workloads to be consolidated to one or more virtual contexts. The consolidator within the virtual context performs the following tasks:
For each incoming application, it executes all CUDA calls that precede a CUDA kernel launch. These CUDA calls perform tasks like allocation of memory, copying of data structures and related tasks. When there are multiple applications to handle, the consolidator iterates over these CUDA calls from each application in a round-robin fashion. If there are no pending CUDA calls from an application, the CUDA calls from the next application are considered. Thus, it is the responsibility of the VCC to ensure that all the applications make fair progress.

Next, the VCC handles the kernel launch(es). Different applications (under consolidation) may normally execute the kernel launch call at different times. However, the presence of a synchronization operation following an executing kernel may block kernels belonging to other applications. To avoid this situation and allow concurrent kernel execution, the VCC enqueues all pending kernel launches in a Launch Queue, issues all queued kernel launches (using different streams), and defers all synchronization calls. Only when all kernel launches have been issued will synchronization calls be executed.

5.2.4 Design Issues and Limitations

The use of virtual contexts leads to some limitations. CUDA runtime instantiates a GPU context for every application thread. GPU contexts are used for error handling, event handling, and other special uses. Mapping multiple application contexts onto a single virtual context creates a decoupling between errors, events and the applications from which they originate.

The error handling issue is mostly restricted to kernel launches. In fact, other CUDA calls are issued to the CUDA runtime in sequence and therefore their return code can be
reconstructed. In the case of kernel launches, an error in one kernel launch will force a failure of the set of kernel launches consolidated with it. In addition, since mapping decisions are performed after kernel launches have been issued by the frontend, negative return codes on memory operations will be deferred and returned to the frontend only after processing kernel launches.

At present, our framework supports only a subset of the CUDA API. In particular, we focus on device memory allocation, host-device and device-device memory transfers, and kernel calls. In addition, we have not yet considered applications using constant memory and texture memory. We plan to address these limitations in future work. However, we found that a good number of CUDA applications use the subset of the CUDA API supported by our design.

5.3 Consolidation Policies and Algorithm

The key conceptual component of the framework presented in the previous section is the consolidation scheduling module, which makes actual decisions about consolidating the workloads with the goal of improving overall throughput. This section describes the policies and algorithms used in this module.

5.3.1 GPU Sharing Mechanisms

Like any other resource that offers parallelism, there can be two ways of sharing a GPU between two or more independent tasks: space sharing and time sharing, as we elaborate below.

**Space Sharing:** One way to share a GPU is through dedicated allocation of a subset of SMs to each of the kernels. For example, if two kernels request 7 thread blocks each on a Fermi, then each kernel gets a dedicated set of SMs for computation. Thus, there is
no contention for SMs or resources on a SM, such as shared memory. Space sharing can clearly be beneficial when kernels individually do not use all SMs, or do not scale linearly with increasing number of SMs, as we had seen in Section 5.1.

**Time Sharing:** Another approach for sharing a GPU involves allowing more than one kernel to share the compute cycles (and other resources) on a SM. For example, if two kernels request 14 threads blocks each on a Fermi, then the 2 thread blocks mapped to each SM will be sharing the resources. Scheduling with time sharing has been referred to as gang scheduling in traditional supercomputers [48], though it has not been very successful. However, time sharing of each SM is more promising on GPUs, for several reasons. First, each SM itself involves a significant amount of parallelism. Second, very low cost context-switching is supported on each SM, and multi-tasking is an effective way of masking long-latency operations, like accesses to the device memory. In order to better understand time sharing, it is necessary to consider how the CUDA runtime maps and schedules thread blocks to SMs, as discussed in Section 5.1.1. Time sharing happens when the total number of thread blocks exceeds that of SMs. In this situation, if the cumulative resource requirements (in terms of registers and shared memory) of the thread blocks mapped to the same SM do not exceed those available on the SM, then the execution of the thread blocks will be *interleaved*; else, it will be *serialized*. We recall that memory latencies can be hidden by context switching between thread blocks only in the case of interleaved execution; therefore, interleaved time sharing will potentially lead to better performance than serialized time sharing.
5.3.2 Causes of Resource Contention

Clearly, sharing of GPUs (especially time sharing) is not always beneficial, because of the resource contention it introduces. Our framework focuses only on the contentions introduced with respect to compute resources on SMs and the shared memory. Another possible contention could be with respect to the device memory. We do not currently consider it, as the device memory on the latest GPUs is quite large (up to 6 GB). Moreover, because earlier GPUs had much more limited device memory (only a few 100 MB), most kernels have been written to use only a small fraction of the device memory available today. Thus, several such kernels can easily be consolidated without device memory becoming a constraint.

Contention for (and within) Streaming Multiprocessors (SMs): If the sum of thread blocks requested by the kernels that have been consolidated is more than the number of available SMs, resources on each SM will be shared. Each thread block contains a certain number of threads that actually perform the computation. Most applications studies on GPUs have shown that the best performance is obtained from the use of 256 or 512 threads per block, and the use of a larger number of threads (say 1024) usually results in a slowdown. This is because increasing the number of threads up to a certain level can help mask high latency operations, whereas an even larger number makes the scheduling or thread management overheads dominant. Moreover, a large number of threads can cause register pressure, potentially leading to serialized time sharing.

This observation has the following implications for consolidation. If a kernel does not have enough parallelism to profitably use 512 threads, time-sharing it with another application is likely to increase the overall throughput. However, if a kernel is already
using 512 threads profitably, time-sharing it with another kernel will likely not increase the throughput.

**Shared Memory:** Shared Memory is a small fast access memory, which is private to each SM. For example, on a Fermi GPU, the maximum possible size is 48 KB for each SM. Thread blocks can effectively share a SM only if their aggregate shared memory requirements do not exceed the amount of available shared memory (interleaved time sharing). Otherwise, as mentioned above, serialized time sharing will take place.

### 5.3.3 Advanced Policies with Molding

We now discuss how we can *mold* the number of thread blocks and/or number of threads for improving consolidation. Our ideas have some similarities with moldable job scheduling [85] for supercomputing centers, but the details of our approach are specific to the modern GPUs. Our discussion here assumes that there is no hard coding of the computations performed by a particular thread in a particular thread block, and therefore, the number of threads in each block and/or the number of thread blocks can be altered by the scheduling framework. While examining a number of available GPU kernels, this appears like a reasonable assumption. Moreover, as GPU architectures are changing frequently, only the kernels where the number of thread blocks and number of threads within each block can be changed at kernel launch time will be portable. These two parameters have also been identified as *auto-tuning* parameters in some of the recent work [62], which again requires that kernels are written to be flexible.

Two possible policies for molding are applicable on GPUs:

- **Forced Space Sharing:** In this policy, we reduce the number of thread blocks requested by the user to utilize only a subset of the available SMs. For example, the
actual request could be 14 threads blocks, and number of thread blocks after molding could be 7, thus ensuring that only 7 SMs are used. Moreover, while we reduce the number of blocks, we may increase the number of threads within each block. For instance, a request for $14 \times 256$ can be molded to $7 \times 512$.

- **Time Sharing with Reduced Threads**: In this scheme, instead of reducing the number of thread blocks, we only reduce the number of threads within a block. Thus, if a kernel requests the configuration $14 \times 512$, then we may change it to $14 \times 128$.

We now examine how the above schemes can help improve the throughput across different applications. The key underlying idea is that one application often does not have enough parallelism to fully utilize the capabilities of a modern GPU like a Fermi card. Thus, changing the thread configuration from $14 \times 512$ to $14 \times 256$ or even $14 \times 128$ may not reduce the performance of a single application significantly. At the same time, it can open up an opportunity for other applications to use the available resources effectively, thus increasing the overall throughput.

The use of the specific policy can depend upon the resource constraints between the applications. For example, if the aggregate shared memory requirements of two kernels exceeds the amount of available shared memory on each SM, time sharing is not going to be beneficial. However, the Forced Space Sharing policy can convert time sharing to space sharing, eliminating the contention for shared memory.

Clearly, there will be cases where molding will not improve overall throughput, while it can increase the turnaround time for a single application. Thus, it is important for the consolidation algorithm to carefully decide which kernels are appropriate for molding, and which particular molding policy should be used.
Algorithm 1 Runtime Consolidation Scheduling Algorithm

1: Configuration List of all $N$ Kernels
2: $WQ_1 = \phi$, $WQ_2 = \phi$
3: $A[][] = \text{GeneratePairwiseAffinity}(K)$
4: $[k_i, k_j] = \text{FindMinAffinityPair}(A[][])$
5: Push $k_i$ into $WQ_1$
6: Push $k_j$ into $WQ_2$
7: for all Kernels $K - k_i$ and $k_j$ do
8: $k_l = \text{GetNextKernel}()$
9: $a_1 = \text{GetAffinityForList}(\text{ConfigList}(k_l, WQ_1))$
10: $a_2 = \text{GetAffinityForList}(\text{ConfigList}(k_l, WQ_2))$
11: $[a_3, \text{NewConfigSet}] = \text{GetAffinityByMolding}(\text{ConfigList}(k_l, WQ_1))$
12: $[a_4, \text{NewConfigSet}] = \text{GetAffinityByMolding}(\text{ConfigList}(k_l, WQ_2))$
13: if MaxAffinity($a_1$) then
14: Push $k_l$ into $WQ_1$
15: else if MaxAffinity($a_2$) then
16: Push $k_l$ into $WQ_2$
17: else if MaxAffinity($a_3$) then
18: Apply $\text{NewConfigSet}$ to $k_l$ and $WQ_1$
19: Push $k_l$ into $WQ_1$
20: else
21: Apply $\text{NewConfigSet}$ to $k_l$ and $WQ_2$
22: Push $k_l$ into $WQ_2$
23: end if
24: end for
25: Dispatch $WQ_1$ to Virtual Context$_1$ for consolidation
26: Dispatch $WQ_2$ to Virtual Context$_2$ for consolidation

5.3.4 Runtime Consolidation Algorithm

We now present our consolidation algorithm. Specifically, we use a greedy algorithm that consolidates $N$ given kernels on 2 GPUs (Algorithm 1). Note that our algorithm can easily be extended to the case when $N$ kernels need to be consolidated on $k$ GPUs. In practice, with current technologies, VMs from a single node can access up to 4, but most likely 2 GPUs.
Algorithm 2 Generate Pairwise Affinity

1: Input: Kernel Configuration set $K_1...K_N$
2: for $i = 1$ to $N$ do
3:     for $j = 1$ to $N$ do
4:         if $i \neq j$ and $j > i$ then
5:             if SpaceSharing($k_i$, $k_j$) then
6:                 Affinity[$i,j$] = 1
7:             else
8:                 if SHMEM$_i$ + SHMEM$_j$ ≤ MAXSHMEM then
9:                     Affinity[$i,j$] = 0
10:                else
11:                    Affinity[$i,j$] = 1-(THREADS$_i$+THREADS$_j$)/1000
12:             end if
13:         end if
14:     end for
15: end for
16: Return Affinity[$i,j$]

The input for the consolidation algorithm is a list of execution configurations, each of which is a 3-tuple (Number of thread blocks, Number of threads, and amount of shared memory), representing the execution of each kernel. An important data structure in the algorithm is the Work Queue (WQ), where the workloads to be consolidated are enqueued.

The first step in our algorithm is to compute an affinity score for a pair or a set of kernels. An affinity score determines the likely benefit of consolidating 2 or more kernels. We initially generate a pair-wise affinity for all the $N$ workloads. The pseudo-code for this step is shown as Algorithm 2. If two kernels are trying to space share, then there is no resource contention between them, and we assign the highest affinity score, 1. On the other hand, if the workloads will be time-sharing the resources, contention needs to be considered. If the aggregate of the shared memory requirements of the kernels is more than the maximum available shared memory, then clearly there will not be any benefit
from consolidation, and thus, the affinity is 0. Next, we consider the number of threads that contend for the SM’s cores. We assign an affinity score depending on the pairs’ total number of threads, with the idea that smaller the total number of threads, higher the affinity.

Now, the overall consolidation algorithm works as follows. From the array of generated affinity scores, we find the pair with the lowest mutual affinity. We first split these two workloads on to different GPUs. We now iterate over each of the remaining \( N - 2 \) workloads to find the appropriate GPU where each of them should be placed. To do this, we need to be able to determine the affinity score for a particular kernel with a list of kernels that are already in the work queue of each GPU. This method is very similar to the one for generating pair-wise affinity, with the difference that here we consider the resource contention for a list of workloads.

Note that, so far, the affinity scores were calculated without performing any molding on the workload’s requirements. To consider the potential benefits from molding, we also calculate the affinity based on molding with each work queue. The procedure is shown as
Algorithm 3. As we had described earlier, we consider two different ways of molding the resource requirements. We first determine the set of time sharing kernels that are involved in shared memory contention. The only possible way to break this tie is by molding them with the *Forced Space Sharing* policy. Next, we check, if for time sharing cases, the total number of threads scheduled on each SM is too large. If so, we can mold these kernels to use fewer threads or thread blocks, i.e., it is possible to apply either of the two molding policies.

The challenge, however, lies in finding the best molding configuration at runtime. Besides the question of choosing between time-sharing and space-sharing, there is also the question of the factor by which the number of thread blocks or the number of threads should be changed. These decisions really depend on the performance of the individual kernel under different execution configurations. Thus, our approach is to predetermine the performance of each workload across different number of thread blocks as well as the number of threads in each thread block. This information is stored in a *table* in advance, and at runtime, we lookup the table to find the most suitable new configuration. This operation is encapsulated in the method *FindConfigForMold()* in the line 8.

Returning to our overall method for consolidation, we find the affinity of a kernel with existing kernels on each work queue, with and without molding. From this set of affinity scores, we choose the work queue in which the kernel should be placed. After all the kernels have been mapped to the work queues, the dispatcher passes the work queue to the corresponding virtual context for execution with consolidation.
5.4 Experimental Results

In this section, we first describe our evaluation methodology, and then present the results from experiments performed to evaluate the effectiveness of our consolidation framework.

5.4.1 Evaluation Methodology

**Experimental Setup:** Our experiments were conducted on a machine equipped with two Intel Quad core Xeon E5520 CPUs (running at 2.27 GHz), 48 GB main memory and two Nvidia Tesla C2050 (Fermi) GPU cards. Each Tesla C2050 card has 14 streaming multi-processors (SMs), each containing 32 cores (total 448 cores) running at 1.15 GHz, and a 3 GB device memory capacity. The maximum amount of shared memory that is available per SM is 48 KB. This machine with two C2050 GPUs was virtualized using gVirtuS version 2.0, which is based on QEMU.

**Benchmarks:** In our evaluation we use eight benchmark applications that vary in several ways, including in their use of shared memory. The benchmarks and their shared memory use are listed in Figure 5.5. As we can observe, the shared memory usage varies from

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Memory characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Processing (IP)</td>
<td>No ShMem</td>
</tr>
<tr>
<td>PDE Solver (PDE)</td>
<td>No ShMem</td>
</tr>
<tr>
<td>BlackScholes (BS)</td>
<td>No ShMem</td>
</tr>
<tr>
<td>Binomial Options (BO)</td>
<td>Low ShMem (up to 3KB)</td>
</tr>
<tr>
<td>K-Means Clustering (KM)</td>
<td>Med ShMem (up to 16KB)</td>
</tr>
<tr>
<td>K-Nearest Neighbour (KNN)</td>
<td>Med ShMem (up to 16KB)</td>
</tr>
<tr>
<td>Euler (EU)</td>
<td>Heavy ShMem (up to 48KB)</td>
</tr>
<tr>
<td>Molecular Dynamics (MD)</td>
<td>Heavy ShMem (up to 48KB)</td>
</tr>
</tbody>
</table>

Figure 5.5: Benchmarks and its Memory Characteristics
none to heavy going from Image Processing (IP) to Molecular Dynamics (MD). For our experiments, molding could be performed on all eight applications. As we have mentioned earlier, molding is possible only if a change in the execution configuration of the involved CUDA kernels does not affect the produced results (but only the performance). This property holds for all benchmarks except Binomial Options (BO): we modified the source code of BO to allow molding.

A brief description of our benchmarks is as follows. Image Processing (IP) and PDE solver (PDE) are \textit{structured grid} applications. In particular, IP uses \textit{sobel filter} to solve the edge detection problem in image compression, whereas PDE uses a iterative jacobi method to solve partial differential equations. Black Scholes (BS) and Binomial Options (BO) are popular models for \textit{option pricing} in the computational finance domain. K-Means Clustering (KM) and K-Nearest Neighbor (KNN) are popular applications in the data mining domain for clustering and classification, respectively. Euler (EU) represents an unstructured mesh computations widely used in computational fluid dynamics. Molecular Dynamics (MD), also an irregular application, is used to study the structural equilibrium and dynamic properties of molecules represented in an unstructured mesh. The data sets used in our evaluation are summarized in Figure 5.6.

\textbf{Evaluation Metric and Goals:} The evaluation metric used throughout this section is the relative throughput benefit of GPU workload consolidation over sequential execution, i.e. executing applications one at a time on each GPU. Specifically, if for a set of $K$ applications, $T_s$ is the running time in the case of sequential execution on a GPU, and $T_c$ is the running time in case of consolidated execution, then the relative throughput benefit can be defined as $T_s/T_c$.

We had the following goals in designing our experiments.
Figure 5.6: Dataset Description for Benchmarks

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Data Set Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Processing (IP)</td>
<td>2<em>3584</em>3584 points</td>
</tr>
<tr>
<td>PDE Solver (PDE)</td>
<td>2<em>3584</em>3584 points</td>
</tr>
<tr>
<td>BlackScholes (BS)</td>
<td>1,000,000 options</td>
</tr>
<tr>
<td>Binomial Options (BO)</td>
<td>256 options, 2048 steps</td>
</tr>
<tr>
<td>K-Means Clustering (KM)</td>
<td>4194304 points</td>
</tr>
<tr>
<td>K-Nearest Neighbour (KNN)</td>
<td>4194304 points</td>
</tr>
<tr>
<td>Euler (EU)</td>
<td>10,000 nodes, 60,000 edges</td>
</tr>
<tr>
<td>Molecular Dynamics (MD)</td>
<td>130,000 nodes, 16,200,000 edges</td>
</tr>
</tbody>
</table>

- We want to evaluate the space and time sharing mechanisms described in Section 5.3.1 and demonstrate the potential of GPU consolidation in the cloud.

- Then, to show how the advanced policies described in Section 5.3.3 can lead to a significant improvement in the throughput even for GPU kernels exhibiting high or conflicting resource requirements.

- Finally, we want to evaluate the overheads of our consolidation framework.

### 5.4.2 Evaluation

In all the following experiments, a set of applications are scheduled on two GPUs. The configuration \((X, Y)\) on each bar represents the following. \(X\) denotes the cumulative sum (from consolidated kernels) of thread blocks on the entire GPU. \(Y\) denotes the cumulative sum (from consolidated kernels) of threads per SM in a GPU. This helps in understanding the type of molding applied.

**Impact of Basic Policies on Throughput**

We now evaluate the mechanisms described in Section 5.3.1, namely space sharing and time sharing. We start by using a blind policy to perform consolidation. In other words,
the consolidation will be oblivious of the execution configurations of the workloads. Thus, applications are simply mapped to GPUs in a round robin fashion, and CUDA kernels mapped on the same GPU are consolidated.

**Benefits from Space Sharing:** The results from space sharing are shown in Figure 5.7. In this experiment all the applications are started with an execution configuration of either $7 \times 512$ or $7 \times 256$. Since each GPU has 14 SMs, if two applications are assigned to a GPU, then the corresponding kernels will be space shared. We run the 8 benchmarks in two batches. In $batch_1$ we run BS, BO, KM and KNN; whereas in $batch_2$ we run PDE, MD, EU and IP. In all cases, the space-shared execution of two kernels per GPU has almost
doubled the throughput on both the GPUs, when compared to sequential execution. The high throughput benefit is due to the lack of resource contention: each thread block runs on a dedicated SM.

**Benefits from Time Sharing:** The results from time sharing are presented in Figure 5.8. In this case, we run three batches of execution, with each batch containing four applications. The execution configurations are chosen so that the number of thread blocks will lead to time sharing the SMs. The throughput benefit ranges in this case from 31% to 86%. This experiment shows that performance improvements can be obtained also in case of time sharing. However, as we will show next, such benefits cannot be always achieved by performing time sharing through simple round robin mapping policies.

**Impact of Contention on Basic Policies:** In the previous experiment the resource contention is low. In fact, even if multiple thread blocks are mapped to the same SM, the cumulative sum of shared memory requirement per SM is always below the maximum availability. In addition, the maximum number of threads per SM is limited to 512. We recall that, when multiple thread blocks are mapped to the same SM, as long as their overall resource requirements (in terms of shared memory and registers) fit the SM availability, they will time share the SM in an interleaved fashion. Context switch between thread blocks
will help hiding memory latencies. On the other hand, if the overall resource requirements exceed the SM availability, serialized time sharing will occur. Figure 5.9 shows different scenarios where no throughput benefit from consolidation can be observed, either because the number of threads per GPU is large ($batch_1$ and $batch_2$), or because the cumulative shared memory requirement exceeds the SM availability ($batch_3$). Figure 5.10 shows how increasing the cumulative number of threads can decrease the throughput. We recall that increasing the number of threads introduces register pressure: when the cumulative number of threads exceeds the GPU availability, the execution of thread blocks mapped to the same SM is serialized.

Overall, the basic policies have the potential for introducing throughput benefits by means of kernel consolidation. However, the lack of consideration about resource contention can be a big obstacle in achieving throughput improvements from consolidation in many cases. Next, we show how introducing affinity scores and using molding based policies can further improve the throughput.
We now evaluate the benefits of using the proposed advanced policies for GPU consolidation, namely affinity score based policies and kernel moldability.

**Benefits from Affinity Scores:** We first evaluate the benefits that can be obtained from using only affinity scores. We consider a scenario characterized by the presence of both space sharing and time sharing workloads. In particular, we run BO and BS with $7 \times 512$ threads and KM and KNN with $14 \times 256$ threads each. We consider two cases of execution: one using a basic consolidation policy, and the other using affinity scores. The results are reported in Figure 5.11. Note that, in absence of affinity scores, consolidation may involve a mix of time sharing and space sharing kernels, ultimately leading to 21 blocks per GPU and 768 threads per SM on both GPUs. On the other hand, if we take advantage of the affinity scores described in Section 5.3.4, we can reduce the contention for SM by running the following configurations on the two GPUs: 14 blocks per GPU and 512 threads per SM on one GPU, 28 blocks per GPU and 512 threads per SM on the other. In conclusion, the use of affinity scores allows better block and thread balancing, thus improving the throughput by about 40% on each GPU.

Figure 5.11: Effect of Affinity Scores on Throughput
Figure 5.12: Impact of Molding on Large Threads - Time Sharing with Reduced Threads

Figure 5.13: Impact of Molding on Large Threads - Time Sharing with Reduced Threads and Forced Space Sharing

Figure 5.14: Impact of Molding on Shared Memory Contention using Forced Space Sharing
Benefits of Enabling Molding Capabilities: We now enable molding capabilities in addition to affinity scores. We consider the three scenarios in Figure 5.9. In particular, in the experiments in Figures 5.12, 5.13, and 5.14 we enable molding, and compare the performance with those previously achieved using a basic (Round Robin) mapping policy.

First, consider the results presented in Figure 5.12. In the Molding case, one application per GPU is molded (IP and PDE in the example). After molding, the total number of threads per GPU is reduced, while the number of blocks remains the same. Thus, Time Sharing with Reduced Threads has been applied. Overall, we observe a throughput improvement of about 38% on one GPU and 21% on the other.

Next, consider the results presented in Figure 5.13. In this case, Time Sharing with Reduced Threads is performed on the first GPU, whereas Forced Space Sharing is performed on the second. In other words, the number of threads (for IP) is reduced on the first GPU, whereas the number of blocks is reduced (by a factor of 2) on the second. We can observe a throughput improvement of 31% and 62% on each GPU respectively.

Finally, the only way to avoid the shared memory contention on the configuration in Figure 5.14 is to force space sharing. This can be done by reducing the number of thread blocks for each application (and maintaining the number of threads per block). We can observe a performance improvement of about 30.6% and 77.8% on each GPU respectively.

It should be noted that all the molding decisions are automatically taken based on the scheduling algorithm. That is, which type of molding (Time Shared with Reduced Threads or Forced Space Sharing) to use is determined based on the scalability characteristics of each application. Also notice that, in all the cases above, molding could lead to performance losses due to the reduction in the number of worker threads. We analyze this
per-application performance degradation as well as choice of molding type in the next experiment.

**Impact of Molding on Applications:** In Figures 5.12, 5.13, and 5.14 some applications have been molded. In this process, in some cases (*IP, PDE and KM*) the number of worker threads was reduced by a factor of two, whereas in other cases (*BS, KNN, EU, and MD*) the number of worker threads was maintained despite molding. In the former case, there can be a noticeable loss in the performance, however not large. The per-application slowdown is shown in the Figure 5.15.

---

**Figure 5.15:** Impact of Molding on Per Application Slowdown

**Figure 5.16:** Choice of Molding Type for Workloads
Choice of Molding Type: We now analyze the need for making the right choice of molding type. Independent of the previous experiments, we analyze the influence of both the types of molding on different applications. The results are presented in Figure 5.16. We analyze the slowdown for each workload due to both types of molding. As we can see, each type of molding influences every application differently. It is important to make the right choice of molding type; otherwise, throughput due to consolidation can be degraded. This choice is automatically performed by predetermined information described in algorithm 3 (line 8).

Effect of Consolidation in High Contention Scenarios: We now consider a scenario where all eight benchmarks are scheduled and consolidated on two GPUs. The results are presented in Figure 5.17. On the left hand side, the results from applying a basic policy (no affinity or molding) are shown. On the right hand side, our advanced scheduling algorithm with the ability of affinity scores and moldability is used. The consolidation benefits obtained from the basic policy are marginal on both GPUs. However, molding leads to a performance improvement of 30% and 33% (compared to basic policies) on the two GPUs, respectively. This has been achieved by molding 6 out of 8 kernels. In particular, forced space sharing has been applied to EU, MD, KM, and KNN, whereas
time sharing with reduced threads was applied to IP and PDE. The choice of molding for these applications was based on the per-application predetermined information described in algorithm 3 (line 8). Overall, we show that by efficiently consolidating the workloads, we are able to share eight kernels on two GPUs, thus improving the GPU utilization.

To summarize, we showed that using affinity information can help to improve the thread balancing and hence, the throughput. In addition, molding can be particularly useful in high contention situations characterized by resource contention, either by forcing space sharing, or by avoiding sequential execution of thread blocks during time sharing. Moreover, we make the choice for right type of molding to further improve the throughput.

**Framework Overhead Analysis**

We now analyze the overheads incurred by our consolidation framework.

First, we report the overhead of our consolidation framework when there is no need for consolidation (the number of applications is less than the available GPUs). The baseline for comparison is the plain gVirtuS execution (without consolidation framework). The results are presented in Figure 5.18. As we can see, the overhead is always less than 1% for all applications. This is because, in the absence of workloads to consolidate, the DCDM and the VCC modules do not have real work to do.

Second, we report the overheads when there are more workloads to schedule than the available GPUs. The baseline for comparison in this case is the manually consolidated execution. The sources for overheads in this experiment are the consolidation scheduling algorithm, and the consolidator in the VCC. The results are presented in Figure 5.19. We ran the applications in two batches. The execution configuration had enough contention to require molding in the scheduler. In both batches, the overheads are low (within 4%).
Thus, we have shown that significant throughput benefits can be achieved while keeping the overheads low.

5.5 Summary

We have presented a framework to transparently enable one or more GPUs to be shared by applications executing within a number of VMs. Our contributions include extending an existing open source software, gVirtuS, to provide software mechanisms for enabling consolidation. We have also addressed the conceptual consolidation problem. Particularly, we have carefully studied the characteristics of GPU kernels that can and cannot benefit
from consolidation, and based on this, have developed a method for computing the *affinity score* between two or more kernels. We have also introduced two new *molding* techniques, and have developed an overall algorithm for mapping a set of given kernels to a pair of GPUs.

We have extensively evaluated our framework using eight popular GPU kernels executed in a shared environment with two Fermi GPUs. Our key observations are as follows. First, depending on the execution configuration of the GPU kernels, kernel consolidation will take advantage of one of two mechanisms: space or time sharing. Second, when the applications do not have competing resource requirements, even a blind consolidation policy can guarantee performance benefits over sequential execution (which would be performed using the plain CUDA runtime). Third, when contention is higher (i.e. when more kernels are sharing a GPU, or when the kernels have competing resource requirements) our overall consolidation algorithm, which uses affinity scores and molding capabilities, is effective in improving the throughput. For example, when eight applications are sharing two GPUs, we achieve an average throughput improvement of 50% over sequential execution. Finally, the overhead of our consolidation framework is less than 4%.
Chapter 6: Scheduling Concurrent Applications on a Cluster of CPU-GPU Nodes

Heterogeneous architectures comprising a multi-core CPU and many-core GPU(s) are increasingly being used within clusters and cloud environments. In this chapter, we consider a cluster where each node has a multi-core CPU and a GPU. Our goal is to accelerate a set of applications using the aggregate set of resources in the cluster. We focus on two distinct scheduling problems.

Single Node Job Scheduling: In the first problem, we assume that an application can either execute on a single GPU or a multi-core CPU, on any one node in the cluster. This is consistent with some of the most recent trends in application development for multi-core and many-core architectures. For instance, OpenCL [59] is becoming a popular programming model for heterogeneous architectures, and is device agnostic. Thus, a kernel can be written once and compiled for many devices to produce different binaries. Furthermore, recent research has proposed compilation techniques to dynamically transform CPU into GPU code and vice-versa. For example, MCUDA [87] performs basic transformations of CUDA code into C, Ocelot [31] and PGI CUDA-x86 [5] transform CUDA into optimized x86 compatible code, SWAN [45] performs a CUDA-to-OpenCL transformation, whereas OpenMPC [62] allows OpenMP-to-CUDA translation. In comparison, however, the available support for developing applications for clusters of GPUs is much limited, and
the existing GPU benchmarks like Rodinia [21] and Parboil [4] comprise only single-GPU applications.

Our first scheduling formulation, therefore, views a cluster as having two sets of independent resources (CPU and GPU), and each job can be mapped to any one resource. Such mapping needs to be performed with the goal of maximizing overall throughput, while keeping the latency of each application at a reasonable level.

**Multi-Node Job Scheduling:** The second scheduling problem we consider assumes that applications can be developed to flexibly utilize available resources in a heterogeneous cluster. Thus, an application can either use the GPU, the multi-core CPU, or both, on any number of nodes in the cluster. Thus, the same job can be executed on a cluster of GPUs, or a cluster of CPUs, or both. While there is very limited amount of work on developing applications with these properties today, it is clearly desirable to have such flexibility in the future.

The scheduling problem now involves determining the number of nodes for execution of each application, and whether the application should just use the GPU on each node, the multi-core CPU on each node, or both. Furthermore, we assume that a job using a set of GPUs and a job using CPUs on the same set of nodes can share interprocess communication resources.

In this chapter, we study the problem of optimizing the overall throughput of a set of applications deployed on a cluster of such heterogeneous nodes. We consider two different scheduling formulations. In the first formulation, we consider jobs that can be executed on either the GPU or the CPU of a single node. In the second formulation, we consider jobs that can be executed on the CPU, GPU, or both, of any number of nodes in the system. We have developed scheduling schemes addressing both of the problems. In our evaluation, we
first show that the schemes proposed for first formulation outperform a blind round-robin scheduler and approximate the performances of an ideal scheduler that involves an impractical exhaustive exploration of all possible schedules. Next, we show that the scheme proposed for the second formulation outperforms the best of existing schemes for heterogeneous clusters, TORQUE and MCT, by up to 42%.

6.1 System Model and Challenges

In this section, we describe our system model, i.e., the nature of the environment and applications we consider. Then, we make several observations about possible approaches and challenges.

6.1.1 System Model

In this chapter, we target a cluster of nodes, where each node contains a multi-core CPU and a GPU. Note that with current technologies, there could be multiple GPUs on a single node, or there could be up to 4 GPUs in a box (like the S2050/S2070 boxes from Nvidia) that are accessible from 2 nodes in the cluster. For simplicity, we assume 1 GPU per node, directly accessible only from the multi-core CPU located on the same node. The CPU and the GPU are connected through a PCI-Express interconnect. Each node communicates with the other nodes in the cluster using an interconnection network.

Any given application can either use all cores on the multi-core CPU, or can primarily use the GPU. It should be noted that GPU intensive applications still require one CPU thread. Typically, such thread is used to issue GPU calls and does not consume many computing cycles. Therefore, we can assume that all cores in the CPU can be effectively used by the other application scheduled on the same node to use the CPU. In practice, it is possible that some applications can be executed only on the multi-core CPU or primarily
on the GPU, but for simplicity, we assume that each application can be executed on either platform. As mentioned in the Introduction, recent research has focused on compilation techniques to dynamically transform CPU into GPU code, and vice-versa.

We also assume that there is a shared file system across all the nodes in the cluster. Thus, binaries of the workloads and the corresponding data files need not be communicated across nodes. This is a reasonable assumption for most clusters today. In our scheduling problem, we further assume that no job pre-emption is performed, and the scheduler operates on batches of jobs. However, batches of jobs enter the system stochastically and are not known a priori.

Existing techniques for scheduling across multiple sites [81, 80] and widely used resource managers like SLURM [6] make scheduling decisions based on user inputs regarding execution times and speedups. Similarly to these approaches, we assume that information about relative speedups/execution times are provided by the user or are available through analytical models [58] or profiling [39].

6.1.2 Problem Formulation and Challenges

Our goal is to address two scheduling problems: one targeting single-node and the other targeting multi-node applications. First, given a CPU-GPU cluster and a set of single-node jobs, we want to effectively schedule each job on either one of the multi-core CPUs or one of the GPUs available in the cluster. Second, given a CPU-GPU cluster and a set of multi-node jobs, we want to effectively schedule each job on either a set of CPUs, or a set of GPUs, or a set of combined CPU-GPU resources. In both cases, the goal is to maximize the overall system throughput while keeping the average execution latency at a reasonable level.
To understand the main challenges for our scheduler, we make the following observations. CPUs and GPUs differ not only in their computational capabilities, but also in the memory capacities and latencies. On one hand, a multi-core CPU has a small number of cores, where each core is quite powerful, and MIMD parallelism can be applied across the cores. On the other hand, a GPU has a large number of simple cores, each of them working in an in-order SIMD fashion. Moreover, on CPUs, the best performance can generally be achieved when the number of running threads is at most twice the number of cores. However, on GPUs, a much higher number of threads can help to hide the memory latencies. Finally, GPU programs are hosted by the CPU threads and need to be launched from the CPU. As a result, data are transferred back and forth between CPU and GPU. This cost can sometimes be significant and could reduce the benefits from using a GPU.

CPUs generally perform better than GPUs on small problem sizes and short jobs. As mentioned earlier, GPUs have many cores and typically thousands of threads need to be launched to achieve good performance. Therefore, small problem sizes lead to low GPU utilization. On the other hand, larger problem sizes can allow an effective use of the GPU resources, leading to higher speedups. Besides the problem size, the analysis of the computation to memory ratio and of the computation patterns can help identify the suitability of the workload to a resource. Workloads performing frequent GPU device memory accesses are likely to report lower speedups on the GPU. Moreover, the ratio of data copy between CPU main memory and GPU device memory to the computation is also a key factor in deciding whether a workload is suitable for CPU or GPU. In general, compute-intensive applications can benefit more from the GPU, while memory intensive applications are better suited to multi-core CPUs.
The main challenge for the scheduler is to decide to which resource(s) (multi-core CPU, GPU, or possibly both) a particular workload should be assigned. Since we want to support a high overall throughput, and since different jobs have different performances on different resources, the scheduler will privilege the resources which represent the best match for each task. In addition, the scheduler may perform scheduling decisions that maximize the aggregate throughput of the workload at the expenses of the execution time of each single application. To this end, the scheduler may perform molding, that is, it may assign to some jobs fewer resources than optimally required when such jobs are run on a dedicated cluster.

6.2 Scheduling Policies

We now present scheduling policies for both the problems we have formulated.

6.2.1 Scheduling of Single Node Jobs

The policies we have proposed and evaluated for this case fall into two groups, based on the user input that they require. The first group (Relative Speedup based policies) require the relative multi-core ($MP$) as well as the GPU ($GP$) speedups compared to the sequential CPU version. The second category (Adaptive Shortest Job First policy), in addition to $MP$ and $GP$, also requires the sequential CPU execution time ($SQ$) of the jobs being submitted.

**Relative Speedup Based - Aggressive or Conservative:** This scheme operates on batches of $N$ jobs, and, as mentioned, takes the relative multi-core ($MP$) and the GPU ($GP$) speedups of each job as input. It first creates two job queues ($CJQ$) and ($GJQ$), which contain the jobs that report better speedup on the multi-core CPU and on the GPU, respectively. Each of these queues is sorted in descending order with respect to the difference in performance of the jobs on the two resources. This difference in performance is used as an indicator as to where non-optimal placement would incur less penalty.
Whenever a resource becomes free, the scheduler assigns to it the job at the top of the corresponding queue. This decision is straightforward if such queue is not empty. However, the non-optimal case arises when the corresponding job queue is empty, e.g. when a CPU (GPU) becomes available for execution, but there are no pending jobs that would prefer to execute on the CPU (GPU). We propose two schemes that handle this case differently.

In the Aggressive scheme, a job available for scheduling is assigned to the non-preferred resource. This is done so as to avoid overall system throughput degradation due to waiting (queuing delay) for the optimal resource. In particular, if a CPU (GPU) is idle, the aggressive scheme dequeues the task from the bottom of $GJQ$ ($CJQ$), and schedules it on the CPU (GPU). In the Conservative scheme, the jobs in the queue are not scheduled until the preferred resource becomes available. In this case, idle times are preferred over slower execution of any task due to non-optimal assignment.

Note that the Relative Speedup based schemes operate oblivious of the actual execution times of the tasks. Without this information, it is not possible to determine the amount of idle time or the amount of slowdown expected by scheduling a task on a non-preferred resource.

**Adaptive Shortest Job First:** This scheme, in addition to the inputs taken by the previous policies, requires the sequential CPU execution time ($SQ$) of each job being submitted. In particular, this additional information makes the Adaptive Shortest Job First scheme an enhanced version of the Relative Speedup based schemes. This scheme improves over the previous one in two ways.

First, the knowledge of the expected execution times allows to easily determine the penalty for non-optimal resource assignments. Thus, when faced with a non-optimal assignment, the scheduler can choose between the aggressive or conservative decision, based
on the calculation of the penalty of each. Specifically, the scheduler compares the minimum penalty for scheduling a job on the non-optimal resource to the wait time for the optimal resource to become free. Second, this policy schedules shorter jobs before longer ones. This reduces the latency perceived by the user.

Overall, this policy will get the best of the aggressive and conservative policies above, and further reduce the latency caused by queuing delays.

6.2.2 Scheduling of Multi-Node Jobs

We now describe a scheme we have developed specifically for multi-node jobs. As compared to the scheduling in the previous subsection, we assume that jobs are flexible in two other ways: 1) A job can simultaneously utilize both the multi-core CPU and the GPU within a node, i.e. in addition to the possibility of using either only the multi-core CPU or the GPU within a node, 2) A job can request multiple nodes in the cluster as specified by the user. As we had stated earlier, our work is assuming a programming model where such flexibility is possible. For example, MATE-CG [52] is a map-reduce like system which can execute applications to use either multi-core CPU (CPU-only), a GPU (GPU-only), or both (CPU+GPU), on any number of nodes of a cluster, starting from the specification of reduction functions. We expect further developments in this area in the near future, allowing a larger class of applications to be developed for such flexible execution.

Before presenting our scheduling scheme, we present the main insight for our scheduling decisions. When run in isolation, a job may achieve the best performance by simultaneously utilizing both the multi-core CPU and the GPU within each node. However, when we are trying to concurrently schedule a set of jobs, always allocating both the multi-core CPU and the GPU to one job may not lead to the best overall throughput. This is because
jobs can differ in their relative performance between the CPU and the GPU. If we want to schedule two jobs, one of which uses the CPU very well and another uses the GPU very well, it may be desirable to give a set of CPUs to one job, and the set of GPUs (from the same nodes) to another job.

Similarly, a job may achieve the best performance when the allocation is equal to the number of nodes requested by the user. However, there are two factors that need to be considered before allocating as many nodes as requested by the user. First, the jobs may not have a perfect linear scalability on the requested number of nodes. Next, the job may have to wait for a much longer time in the queue for requested number of nodes. However, a smaller number nodes (say half of requested) may be available much sooner. Thus, in many cases, allocations with small number of nodes as requested may lead to effective global throughput.

To enable such scheduling decisions, we assume that the execution times of the jobs are available when utilizing both CPUs and GPUs, only the multi-core CPUs, or only the GPUs. Since our scheduling algorithm consider the possibility of flexibly molding the number of nodes, we also take the execution times of the jobs on half and quarter of the number of nodes originally requested for the job. We assume that this information can be provided by the user, or is available through profiling and/or modeling.

**Flexible Moldable Scheduling Scheme (FMS):** As the jobs arrive in batches, we first group and sort the jobs in the following way. The submitted jobs are grouped based on the resources requested for each job (in our case, the number of nodes). Thus, jobs with similar resource requirements are grouped. Now, within each group, the jobs will be sorted in the ascending order of their execution times corresponding to the $CPU + GPU$ version.
Grouping the jobs based on the resource requirement (number of nodes) improves the minimization of resource fragmentation. For instance, in most cases, since \( i^{th} \) and \( (i+1)^{st} \) jobs request the same (or similar) number of nodes, it reduces the need to wait for additional resources due to disparity in the request. Sorting the jobs within a group increases the chance of bringing two jobs with similar execution times next to each other during scheduling considerations. This is a key step that help our decision making process which is discussed below.

In our scheme, we consider the scheduling options for a pair of jobs at every scheduling event, rather than one job at a time. This is done to obtain a good global view for scheduling and to make more informed decision for coscheduling jobs on the same set of nodes, without having a very high cost that would arise from considering all possible options.

We earlier mentioned that three types of resource allocations (CPU-only, GPU-only, and CPU+GPU) are possible. Let us assume that the execution times (for three versions) of a given job \( J_i \) on a requested number of \( N \) nodes (\( N \) is less than the number of nodes in the cluster) are \( T(i, N, C) \), \( T(i, N, G) \), and \( T(i, N, CG) \), denoting the CPU, GPU, and CPU+GPU executions, respectively, on \( N \) nodes. Now, for a number of jobs that are already executing or submitted for execution, we maintain the Current Completion Time (CCT) for these jobs in the system. When scheduling two new jobs \( J_1 \) and \( J_2 \), both of which are requesting \( N \) nodes, we have the following possibilities.

First, both the jobs can be allocated CPU+GPU on \( N \) nodes sequentially. Second, they can be given \( N \) nodes each in parallel (if 2 times \( N \) nodes are currently available). Third possibility is to map one job to only CPUs and another to only GPUs, both on the same set of \( N \) nodes, at the same time. For each of these possibilities, we predict the completion times for jobs in the system, and choose the option that leads to the lowest completion times.
time. Note that while second option will result in lowest completion time for two jobs under consideration, it will not allow other jobs in the queue to be scheduled for a longer time. However, if the system is lightly loaded, it may be the best option overall.

Another possibility we need to consider is as follows. When the requested number of nodes, \( N \), are not available at the time of scheduling, or when the system is very highly loaded, we consider the possibility of molding the number of requested nodes. Specifically, we consider the possibility of molding the number of nodes to \( N/2 \) and \( N/4 \). These possibilities are considered in addition to the possibilities listed above. Among all these options, we choose the one that results in fastest global completion time.

It is easy to see that this scheme is not optimal, as it considers sharing only among a pair of jobs, and does not consider the impact on jobs that may arrive in the future. However, as we stated above, by comparing options across only two similar jobs, we keep the scheduling overheads low. Our scheme can also be extended to consider more than two jobs for possible resource sharing.

### 6.3 Experimental Results

#### 6.3.1 Experimental Setup

Our experiments are conducted on a cluster consisting of sixteen heterogeneous nodes. Specifically, each machine is equipped with two Intel Quad-core Xeon E5520 CPUs (8 cores each running at 2.27 GHz), 48 GB main memory and an Nvidia Tesla C2050 (Fermi) GPU card. Each Tesla C2050 card has 14 streaming multiprocessors (SMs), each containing 32 cores (for a total of 448 cores) running at 1.15 GHz, and 3 GB device memory. The machines are connected through an Infiniband network.
6.3.2 Benchmarks

In our evaluation, we use two sets of benchmarks: one consisting of single-node, and the other consisting of multi-node applications.

**Single-node applications:** Our single-node applications workload includes the ten applications listed in Figure 6.1. As can be seen, this benchmark covers different application domains: Scientific (PDE Solver and PCA), computational finance (Black Scholes and Binomial Options), physics (FDTD, Monte Carlo and Molecular Dynamics), image processing (Image Processing), and machine learning (Kmeans and KNN). For each of these applications, we consider three execution configurations: small dataset, large dataset, and large number of iterations (on large dataset). In Figure 6.1, we show the results obtained by running each benchmark program with the large dataset configuration. Specifically, we report the sequential execution time (on CPU), as well as the GPU and CPU speedups relative to sequential execution. Due to space limitations, we only show the results obtained from large dataset, however, we discuss the trends from all the three configurations below.

The significance of using three execution configurations is the following. When large datasets are used, the majority of the benchmarks (6 out of 10) run faster on the GPU as compared to the multi-core CPU. The trend changes when small datasets are used; in this case, the majority of the benchmarks (7 out of 10) perform better on the multi-core CPU. This is expected, since with small datasets the utilization of the 448-core GPU is limited, and the initial overheads on the GPU can limit the speedup. Similarly, when the number of iterations increases, the ratio of data transfer overhead to the computation time decreases, leading to increased performance improvement on the GPU.
Having considered 10 applications and 3 configurations, we have created a pool of 30 jobs that perform differently on CPU and GPU. In the experiments described in the next section, we create workload mixes by randomly selecting jobs from the described pool.

**Multi-node applications:** Our multi-node applications workload includes three benchmark applications: Gridding Kernel (GK), Expectation Maximization (EM), and PageRank (PR). These applications arise from scientific data processing, data mining, and web search, respectively. These applications were developed using a middleware MATE-CG [52], which allow map-reduce like applications to be developed to use a multi-core CPU, a GPU, and/or both, on any number of nodes in a system.

We execute these applications using two datasets, three numbers of nodes (2, 4 and 8), and three kinds of resources (only multi-core CPU, primarily GPU, and hybrid CPU-GPU). The results for all execution configurations are reported in Figure 6.2. Note that the CPU+GPU configuration exhibits good scalability for GK, but does not noticeably scale in case of EM and PR. Moreover, all three applications scale well with the number of nodes. Thus, by considering three applications, two datasets and three resource requirements (i.e., number of nodes), we create a pool of 18 jobs that we will use in our multi-node experiments.

### 6.3.3 Experiments on Single-Node Applications

In our single-node applications experiments, we compare our proposed scheduling policies - Relative Speedup Based with Aggressive Option (RSA), Relative Speedup Based with Conservative Option (RSC), and Adaptive Shortest Job First policy (ASJF) - with two baseline schemes: Blind Round Robin (BRR) and Manual Optimal. The
Figure 6.1: Single-node Benchmark Applications Characteristics - Large Datasets

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PDE Solver</td>
<td>7.3</td>
<td>4.7</td>
<td>6.8</td>
<td>14396*14396</td>
</tr>
<tr>
<td>Image Processing</td>
<td>33.8</td>
<td>5.1</td>
<td>7.8</td>
<td>14396*14396</td>
</tr>
<tr>
<td>FDTD</td>
<td>8.4</td>
<td>2.2</td>
<td>7.6</td>
<td>14396*14396</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>2.6</td>
<td>2.1</td>
<td>7.2</td>
<td>10 mil options</td>
</tr>
<tr>
<td>Binomial Options</td>
<td>11.8</td>
<td>5.6</td>
<td>4.2</td>
<td>1024 options</td>
</tr>
<tr>
<td>MonteCarlo</td>
<td>45.4</td>
<td>38.4</td>
<td>7.9</td>
<td>1024 options</td>
</tr>
<tr>
<td>Kmeans</td>
<td>330.0</td>
<td>12.1</td>
<td>7.8</td>
<td>1.0 * 10^8 points</td>
</tr>
<tr>
<td>KNN</td>
<td>67.3</td>
<td>7.8</td>
<td>6.2</td>
<td>670108864 points</td>
</tr>
<tr>
<td>PCA</td>
<td>142.0</td>
<td>9.7</td>
<td>5.6</td>
<td>250000 nodes</td>
</tr>
<tr>
<td>Molecular</td>
<td>46.6</td>
<td>12.9</td>
<td>7.9</td>
<td>2170400 nodes</td>
</tr>
<tr>
<td>Dynamics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.2: Multi-node Benchmark Applications Characteristics

<table>
<thead>
<tr>
<th>Application</th>
<th>Dataset Size</th>
<th>No. of Nodes</th>
<th>CPU Time [s]</th>
<th>CPU+GPU Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Griding Kernel (GK)</td>
<td>680 MB</td>
<td>2</td>
<td>314</td>
<td>171</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>179</td>
<td>135</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>163</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>2.7 GB</td>
<td>2</td>
<td>1290</td>
<td>1152</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>618</td>
<td>591</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>338</td>
<td>301</td>
</tr>
<tr>
<td>Expectation</td>
<td>650 MB</td>
<td>2</td>
<td>109</td>
<td>39</td>
</tr>
<tr>
<td>Maximization</td>
<td></td>
<td>4</td>
<td>55</td>
<td>19</td>
</tr>
<tr>
<td>(EM)</td>
<td></td>
<td>6</td>
<td>28</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>900 MB</td>
<td>2</td>
<td>215</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>106</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>54</td>
<td>16</td>
</tr>
<tr>
<td>Page Rank (PR)</td>
<td>900 MB</td>
<td>2</td>
<td>8.5</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>3.3</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>1.8</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>3.6 GB</td>
<td>2</td>
<td>26</td>
<td>4.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>13</td>
<td>2.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>8.2</td>
<td>1.5</td>
</tr>
</tbody>
</table>
Blind Round Robin (BRR) scheme is oblivious to the fact that jobs perform differently on different resources; the Manual Optimal scheme assumes that the performances on CPU and GPU of all jobs in the workload mix are known a priori and an exhaustive search can be applied to find the schedule leading to the best overall system throughput.

Our evaluation is based on four metrics. First, we use the total completion time for the entire workload mix (Comp. Time or $CT$) as an indicator of the system throughput. The next two metrics focus on the scheduling related delays incurred in the execution of each job. If there is only one job to be scheduled, it will be executed on the resource on which it will perform better, and its completion time will equal its execution time on this resource. In practice, competition for resources may make the completion time of each job higher. In fact, competing jobs may be scheduled on a non optimal resource, or may incur queuing delays. The second metric we consider is the average delay introduced due to the non-optimal assignment for all the jobs in the workload mix (Ave. NOA. Lat or ANL). The third metric is the average queuing delay (Ave. QD Lat. or AQL). Finally, our last metric is the maximum idle time (Max. Idle Time or MIT), or the amount of time the least occupied resource in the entire system is not used. This metric can be seen as an indicator of resource utilization.

**Results from Different Workload Mix** The first experiment is conducted on a set of two nodes from the cluster described in Section 7.4.1. Three workload mixes, each consisting of 24 randomly selected jobs, are considered. Workload mixes are generated as follows. We categorize the jobs as CPU- and GPU-friendly (depending on their performances on the different computing resources). The first workload mix is characterized by an equal number of CPU-friendly and GPU-friendly jobs; in the second and third workload mixes, 75% of the jobs are CPU- and GPU-friendly, respectively.
Figure 6.3: System Throughput and Latency for Uniform CPU/GPU Workload

Figure 6.4: System Throughput and Latency for CPU Biased Workload

Figure 6.5: System Throughput and Latency for GPU Biased Workload
The results reported on all considered workload mixes are shown in Figures 6.3, 6.4, and 6.5. In the charts, the performance of each policy is normalized with respect to the best reported result. When averaging the results across the three considered workload mixes, we can observe that our proposed schemes (RSA, RSC, and ASJF) exhibit a 108% lower completion time over BRR. In addition, our policies show significantly better results with respect to the two latency factors ANL and AQL, as well as the utilization metric MIT. When compared to the ideal Manual Optimal policy and considering the completion time, our proposed schemes perform on the average within 20.6% and the best of the three proposed schemes performs within 12%.

We next analyze the trade-offs between RSA, RSC, and ASJF. RSA, as expected, incurs a higher ANL and a low AQL across all workload mixes. In the case of RSC, ANL is always 0 (by default), and correspondingly, AQL is higher. Thus, the relative total completion time between RSA and RSC is dependent on the dominant factor between ANL and AQL. In the case of ASJF, AQL is very low. Since the RSC policy waits until the optimal resource is available, it results in a high MIT as compared to RSA and ASJF. Overall, we can see that ASJF is able to almost always achieve the best results in completion time as well as application latency. However, it achieves these benefits at the cost of needing an additional input value from the user.

While our charts have reported normalized values for all metrics, it is worth comparing the absolute values of maximum idle time and the completion time. For the balanced workload, these values are 226.6 and 705.2 seconds, respectively. This shows that our schemes are quite successful in keeping the resources utilized, i.e, each resource is occupied for 68% of the time or more.
6.3.4 Experimental Results on Multi-node Applications

In our multi-node applications experiments, we compare our scheduling schemes with two existing policies: a practical scheme used by TORQUE [9], a popular scheduler for today’s heterogeneous clusters, and a published scheme, called Minimum Completion Time [69]. We refer to these two schemes as TORQUE and MCT. TORQUE always assigns the resources (type of resource as well as number of nodes) as requested by the user. Thus, for TORQUE, we assume that the user always requests the resource that provides the best performance for the specific job. On the other hand, MCT considers the option of molding the resource type (selecting either multi-core CPU only, or GPU-only, or CPU+GPU), and assigns a job to the resource where it will complete the earliest depending on the waiting times on each resource. However, our algorithm for molding the resource type is different from MCT. In particular, we consider the opportunity to co-schedule jobs on a (or set of) node(s) so to maximize the resource utilization. In addition, we consider downgrading the number of nodes requested by the user, which is not performed by TORQUE and MCT.

The experiments were performed using all 16 nodes in the cluster described in Section 7.4.1. Specifically, we run 32 jobs randomly selected from the pool of multi-node jobs described earlier, and we dispatch them in batches of 8. The inter-arrival times between batches are exponentially distributed [33].

Varying Execution Length of Jobs In the first experiment, we use three job mixes created by varying the ratio of long and short running jobs. The results are shown in Figure 6.6. On the x-axis, $x \text{ SJ}/y \text{ LJ}$ stands for $x\%$ short and $y\%$ long running jobs.

As mentioned earlier, we compare our scheduling scheme with TORQUE and MCT. Our proposed FMS scheme (Molding ResType+NumNodes) performs molding on both resource type and number of nodes. To better understand how different factors impact the
performances, we show the results reported when molding only along one dimension: either the resource type (MoldingResType Only), or the number of nodes (MoldingNumNodes only). In particular, we report the total completion time normalized over the completion time obtained using the (Molding ResType+NumNodes) scheme.

As can be observed, our FMS (Molding ResType+NumNodes) scheme achieves the least completion time and is up to 42% better than the faster of TORQUE and MCT. It is worth noting that each type of molding when applied in isolation leads to good performance improvement when compared to TORQUE and MCT. However, when both the molding techniques are applied together, we obtain an additional performance improvement of about 12%. In addition, the presence of long jobs creates an imbalance in the simultaneous availability of the nodes. Our schemes handle those imbalances much better than the considered baselines.

Varying Resource Request Size of Jobs Our second experiment studies the effect of varying the size of the resources requested by the jobs. In particular, in the results shown in Figure 6.7, we varied the ratio of the number of nodes requested by a job. On the x-axis, $x\ SR/y\ LR$ stands for $x\%/y\%$ of the jobs request a small/large number of nodes.
As can be seen, our scheme reduces the completion time by a factor up to 32% compared to the faster of TORQUE and MCT. Consistently with the previous experiment, we can observe that good performance improvements can be achieved by applying molding only in one dimension. Moreover, the combination of the two molding techniques leads to an additional 11.3% reduction in the completion time. It is interesting to note that, as we increase the ratio of large requests, the benefit from MoldingResType Only is higher than MoldingNumNodes Only. In other words, as the resource contention increases, resource sharing (by using CPU-only and GPU-only implementations) is an effective mechanism to guarantee good aggregate performances.
6.3.5 Additional Considerations

As anticipated, jobs using primarily the GPU (for brevity, GPU-only jobs) need a CPU-core to issue GPU calls. We evaluate the penalty due to sharing a CPU-core between a CPU-only and a GPU-only job deployed on the same set of nodes. We recall that CPU-only jobs are intended to use all cores on one or more multi-core CPUs. The results of experiments performed on three combinations of two multi-node applications are shown in Figure 6.8. In all cases, CPU-only and GPU-only jobs are deployed on the same set of nodes. As can be seen, the average sharing penalty is low (less than 7%) in all cases. This indicates that resource sharing is feasible and desirable in today’s CPU/GPU heterogeneous clusters.

6.4 Conclusions

In this chapter, we have considered the problem of optimizing the overall throughput of a set of applications deployed on a cluster of heterogeneous nodes comprising multi-core CPUs as well as many-core GPUs. Specifically, we have addressed two scheduling problems: one targeting single-node (first problem) and the other targeting multi-node (second problem) applications.

For the first problem, we have proposed three scheduling heuristics - namely Relative Speedup based policy with Aggressive Assignment (RSA), Relative Speedup based policy with Conservative assignment (RSC), and Adaptive Shortest Job First policy (ASJF). Our proposed schemes are based on different amount of information expected from the user, and focus on different tradeoffs between application wait and completion times. We have shown that our scheduling policies outperform a blind round robin scheduler and approximate the performances of an ideal scheduler that involves an impractical exhaustive
exploration of all possible schedules. For the second scheduling problem, we have developed a Flexible Moldable Scheduling scheme (FMS) that considers molding of jobs along two dimensions: resource type and the number of requested nodes. We also demonstrate that our FMS scheme improves the overall throughput by up to 42% when compared to the best of two existing schemes in the literature, TORQUE and MCT.
Chapter 7: Value-Based Scheduling Framework for Modern Heterogeneous Clusters

Heterogeneous computing nodes are becoming commonplace today, and recent trends strongly indicate that clusters, supercomputers, and cloud environments will increasingly host more heterogeneous resources, with some being massively parallel (e.g., GPU, MIC). It is important to revisit scheduling problems for such cluster and cloud environments. Today, most batch scheduling systems that can support heterogeneous clusters do not exploit the portability offered by emerging programming models (e.g., OpenCL), and do not adequately capture the needs of users and resource provider.

This chapter formulates and solves a scheduling problem that captures the requirements of current clusters and cloud environments, while factoring the features of modern heterogeneous systems. Particularly, we consider both the notion of urgency and the concept of priority, capturing both through value functions associated with different jobs. Specifically, for each job, the value function determines the “price” that the user is willing to pay as a function of the execution time. While such differential pricing may not be explicitly supported by the provider, delays captured in our framework can indicate the likelihood of a user dissatisfaction, and thus, their choosing another provider in the future. This, in turn, indicates the likely revenue loss to the provider. In addition, there can be several reasons for service providers to consider priority across jobs and/or users. Our value function captures
these priority levels by allowing different maximum values and/or different decay slopes to the value functions of different jobs. While our model matches some of the considerations in existing literature [19, 20, 22, 50, 82, 61], the existing work in this area has not considered heterogeneous clusters.

To address this scheduling problem, we propose four novel dynamic mapping and scheduling schemes, broadly categorized as Uncoordinated (three schemes) and Coordinated (one scheme). These schemes use resource-specific queues and operate by minimizing the imbalance between them. In our coordinated scheme, we allow periodic monitoring of the state of each queue to dynamically readjust the mapping and scheduling at a finer granularity. Uncoordinated schemes lack this periodic monitoring; however, they can dynamically readjust the mapping and scheduling at a coarser-granularity.

Another factor that needs to be considered is related to the nature of these high-end compute resources. Today’s processors are massively parallel, i.e. multi-core CPUs typically have up to 16 cores, and GPUs typically have up to 512 cores. It is feasible and desirable to co-schedule multiple jobs on a single resource. Our earlier work [76] has demonstrated effective mechanisms for sharing a GPU in a cluster or cloud environment, using system throughput as the metric of interest. In this chapter, we aim at maximizing system profit/user satisfaction, and develop a novel scheduling heuristic to allow jobs to efficiently share multi-core CPUs and GPUs. In particular, we observe that some jobs are less urgent, and/or exhibit performances that do not scale well beyond a certain number of cores, and hence are preferred candidates for sharing without incurring significant penalties.

To demonstrate the effectiveness of our schemes in a realistic setting, we have implemented a prototype scheduler for heterogeneous clusters that include multi-core CPUs
and GPUs. Our framework consists of a *Cluster Level Scheduler* and multiple *Node Level Schedulers*. The former runs on the master node, whereas each of the latter run on compute nodes and is integrated a runtime consolidation framework that enables GPU sharing. In order to evaluate our scheduling schemes, we have compared them against two baseline schemes, which include the scheduling policy used in TORQUE [9] and another published scheme [69]. Our results show that under varying conditions, we can reduce the *completion time* by up to 22% and improve the *aggregate yield* by a factor of 8.8 times. Furthermore, efficient resource sharing allows an additional 23% gain in the yield.

7.1 Problem Definition

We start by explaining the notion of a value function, and relating it to the considerations involved with scheduling on clusters and cloud environments. We then formulate our scheduling problem.

7.1.1 Value Function

The *value function* of a job is a user or system defined function that specifies the value, utility, or yield of that job, based upon the *turnaround time*, i.e. the difference between

![Figure 7.1: An Example Value Function (Linear Decay Slope)](image-url)
the time of completion of the job, and the time the job is initially submitted. A simple illustration of our value function is shown in Figure 7.1. In this example, we have linear decay in the value function, similar to an earlier formulation [50]. Each job is associated with a maximum value which is obtained when the job completes within a specified time limit. If the job is delayed beyond the given time limit, then the value associated with executing the job decays as dictated by the function (decay rate). At each job completion, an yield is obtained from the job. The value, utility, or yield could be viewed from the perspective of both the user and the system provider.

The value function combines the notion of importance and urgency of jobs. It can also represent the Service Level Agreement (SLA) between a user and a service provider. From the users’ perspective, the yield indicates the user satisfaction associated with the execution of the job. On the other hand, from the system perspective, the yield indicates the price that the user will pay for such a quality of service. The overall goal of the system is to maximize the aggregate yield across all jobs entering the system. Consequently, the value function can be used by the scheduler to determine the relative priority of the jobs.

We now discuss how the value function we consider captures existing (or desired) scheduling considerations that apply in supercomputing centers and cloud environments. A large number of supercomputing centers within commercial and academic organizations use the Moab scheduler [3]. The commercial enterprise edition of the Moab workload manager allows for configuring different SLAs between the users and the system. We believe that these SLAs can be captured by different value functions. As an example, an SLA might be that $X\%$ of the jobs from the user will be finished without a delay higher than $Y$. Such an SLA can be captured by a value function that steps the value (or yield) down to 0 after a delay of $Y$. Note that due to the proprietary nature of Moab, the details of underlying
scheduling schemes are not available. However, Moab clearly still has the last two of the
three limitations of TORQUE that we pointed out in the previous section.

Cloud environments support a \textit{pay-as-you-go} model. Resource utilization is a key fac-
tor for cloud providers, and it is well-known that, within cloud environments, resource
\textit{over-subscription} (or under-provisioning) is a commonly used technique to reduce the in-
frastructural costs\(^6\). Over-subscription implies that \(N\) purchased cores may be physically
mapped to \(M\) actual cores, where \((N > M)\). While scheduling different tasks on poten-
tially over-subscribed resources, cloud providers need to pay attention to the satisfaction
of different classes of users. For example, cloud providers often offer \textit{reserved instances},
and owners of these resources will not like to see the performance of their jobs suffer due
to over-subscription. At the same time, a cloud environment may have \textit{free} users (e.g. aca-
demic users who have been given free credits for courses or research activities) and/or users
who have purchased discounted \textit{spot} instances. A cloud provider does not need to be so
cconcerned about the satisfaction of these users. Thus, a value function based formulation
that uses different decay functions can capture the service provider’s objective in a cloud
environment with different classes of users.

\subsection{Problem Formulation}

The problem that we address can be stated as follows. \textit{Given a heterogeneous cluster
with each node comprising a multi-core CPU and a GPU, we want to schedule each job
on either one of the multi-core CPUs or one of the GPUs available in the cluster, so as to
maximize the aggregate yield obtained over all jobs entering the system.}

\(^6\)For example, see http://cloudscaling.com/blog/cloud-computing/subscription-modeling-cloud-
performance
Before showing how we address this problem, we want to clarify two key aspects of our formulation. First, each job will either be allocated a multi-core CPU or a GPU, and not both. However, when a job is allocated a GPU for execution, one additional CPU core is also allocated to the job. This is because GPUs are handled as accelerators and its computations are launched from within a CPU thread (also called host thread), and the results are transferred back to the CPU upon completion of the GPU task. In section 7.4.4, we evaluate the performance interference due to sharing one CPU core between a CPU and a GPU job running on the same node. Moreover, there are no established standards for developing applications to use all cores in a multi-core CPU and a GPU at the same time. This is further confirmed through the nature of applications available in popular heterogeneous benchmark suites like Rodinia [21] and Parboil [4]. It has also been observed that most jobs run more effectively either on a GPU or on a multi-core CPU, and only marginally better when trying to use both [12, 30, 67, 78, 75, 91]. Thus, dedicating an entire node (unless otherwise required) to a single application at any given time may lead to a relatively poor resource utilization. Our proposed schemes can be extended to different cluster configurations (e.g., to nodes comprising multiple GPUs).

The second key aspect of our problem formulation is that each submitted job can be flexibly scheduled either on the multi-core CPU or the GPU. This assumption is justified by the following considerations. First, OpenCL [59] is becoming a preferred choice of programming model for heterogeneous architectures, and is device agnostic. Using OpenCL, an application can be written once and compiled for many devices to produce different binaries. Second, other commercial compilation products like PGI CUDA-x86 [5] transform CUDA into optimized x86 compatible code. Furthermore, MCUDA [87] performs
transformations of CUDA code into C, Ocelot [31] and SWAN [45] performs a CUDA-to-OpenCL transformation, whereas OpenMPC [62] allows OpenMP-to-CUDA translation. Clearly, the community is moving towards a direction where the application developer can write a program once and flexibly generate code for different target platforms. Thus, our scheduling formulation actively exploits such portability to improve efficiency.

7.2 Scheduling Heuristics

In this section, we describe various scheduling heuristics we have developed for the problem described in the previous section.

7.2.1 Overall Approach

The overall approach is as follows. The jobs are submitted in batches and the arrival rate of the jobs is not known a priori. Each submitted job is added into either the CPU queue or the GPU queue. The jobs within each queue are sorted for scheduling based on a criteria we will describe in section 7.2.2. Under normal conditions, jobs from a particular type of queue (CPU queue) will be executed on the corresponding resource (multi-core CPU). But, because imbalance between the queues is likely, a resource type (for instance CPU) may have an empty queue, while it is free. To address such imbalance across queues, we use a number of schemes to dynamically move a job from one queue to another (as described in section 7.2.3). Note that we do not allow pre-emption of an already executing job. This assumption is justified by the overheads and lack of software technology to pre-empt a GPU job. Practical batch scheduler like TORQUE [9] use the walltime (parameter of the qsub command) from the user to estimate how long the job will occupy the resource and, consequently, to guide scheduling decisions. Techniques have been proposed for system-generated prediction [93], however, still relies on user-estimates for scheduling. Similarly,
our scheduler expects a \textit{walltime} for each type of resource (i.e., CPU and GPU). We refer to them as \textit{optimal walltime} and \textit{non-optimal walltime} corresponding to the optimal and non-optimal resource respectively.

\subsection*{7.2.2 Initial Placement and Ordering}

When a batch of jobs enters the system, we first determine the resource type (CPU or GPU) where each job runs with the optimal walltime, i.e. the optimal resource (type) for the job, and enqueue it into the corresponding resource queue. The queue in which the job gets initially enqueued into is referred to as \textit{Optimal Resource Queue} for that job, whereas the other queue is referred to as \textit{Non-Optimal Resource Queue}. Next, the jobs in each queue have to be ordered to maximize the aggregate yield. For this purpose, we use a \textit{market-based} scheme (referred to as \textit{Reward}), which can be viewed as an adaptation of earlier work from Irwin \textit{et al} [50] to our problem formulation (particularly the fact that we are targeting heterogeneous clusters).

Recall from our value function (as shown in section 7.1.1) that the \textit{yield} of the job $i$ is the value obtained from the job when its execution is finished, and is given by:

$$\text{yield}_i = \max \text{value}_i - (\text{delay}_i \times \text{decay}_i)$$ \hspace{1cm} (7.1)

Here, $\max \text{value}_i$, $\text{delay}_i$, and $\text{decay}_i$ are the maximum possible value (or yield) achievable from the job, the delay in finishing its execution, and the decay rate of the value for this job, respectively. Note that, the \textit{delay} factor, in turn, could be a sum of four penalty components: 1) Queuing delay, 2) Slowdown due to sharing a CPU core between a CPU job and GPU job running simultaneously within the same node, 3) Slowdown in execution if the job is scheduled on a non-optimal resource, and 4) Slowdown due to sharing a resource, if its execution is shared with another job.
The Reward scheme for ordering jobs is based on two terms: present value and the opportunity cost. Present Value (PV) from running a job can simply be defined as the value (or yield) that could be obtained after time $t$, which is the optimal walltime of the job, but after discounting the risk involved in running this job. For instance, if two given jobs have same gain, then it might be preferable to run the job with the shorter execution time first. This is because, the shorter job will have a lower risk of deferring an urgent job that may arrive later. Formally, the present value (PV) of a job $i$ per unit time is defined as follows.

\[
PVi \quad \text{OptimalWalltime}_i = \frac{\text{yield}_i}{(1 + \text{discount}\_\text{rate} \times \text{OptimalWalltime}_i)}
\]  

(7.2)

Note that discount\_rate is a tunable factor, which can be chosen based on how significant this term should be.

Next, the second component of the Reward heuristic considers the losses from the opportunity cost of selecting a job $i$ over another job $j$ in the queue. The idea is that if the job $j$ has a higher decay rate than the job $i$, then it may be preferable to run the job $j$ ahead of the job $i$, even if it has a lower value. Thus, the opportunity cost of choosing to run the job $i$ is defined as the aggregate degradation caused in the yield of other jobs in contention at the same time. Formally, the opportunity cost $\text{cost}_i$ of the job $i$ is defined as:

\[
\text{cost}_i \quad \text{OptimalWalltime}_i = \sum_{j=0}^{n} \text{decay}_j - \text{decay}_i
\]

(7.3)

This summation is performed over all other jobs that are currently in the queue. Overall, giving equal weight to the two factors shown in Equations 7.2 and 7.3, the Reward is computed for each job in each queue as follows:

\[
\text{Reward}_i = \frac{(PV_i - \text{cost}_i)}{\text{OptimalWalltime}_i}
\]

(7.4)
After this, the jobs within each pending queue are ordered on to the corresponding resources in the order of the reward score.

### 7.2.3 Scheduling Schemes

We have so far shown how we separate the jobs into two different pending queues (CPU and GPU), and how to order each queue. The remaining questions are, when and which jobs could be moved from one queue to another. We have developed a number of schemes for dynamically moving a job from one queue to another.

The motivation for moving jobs from one queue to another is as follows. Whenever a resource becomes free, and if the corresponding queue is empty, the resource might stay idle, unless a job from the other queue is selected for execution. Allowing a resource to stay idle reduces utilization, and is not desirable. Moreover, this means that jobs in the other queue are being delayed unnecessarily, and thus their yield is being lowered. However, now the challenge is in deciding which job should be moved from their optimal resource queue to execution on the non-optimal resource. Note that different jobs may have different amounts of slowdown when they are moved for execution on the non-optimal resource. We have developed three different uncoordinated schemes to select jobs for execution on the non-optimal resource.

Furthermore, it is interesting to note that even if a resource queue is not empty, there could be a large difference in the amount of delays being incurred by jobs in the two queues. Such an imbalance will degrade the yield of one queue relative to the other, unless there is a coordination between the two queues. To address this problem, we have also developed a coordinated scheme, which can execute job(s) on non-optimal resources even if they are not idle.
Uncoordinated Schemes

In the case of uncoordinated schemes, a job is moved from the optimal resource queue to the non-optimal resource queue only when the non-optimal resource is free and its job queue is empty. Otherwise, there is no coordination between the queues. For uncoordinated schemes, there could be multiple factors based on which a job can be selected for execution on the non-optimal resource queue. The schemes we have developed are based on two key questions that can help maximize aggregate yield: 1) Which of the jobs in the optimal resource queue, when moved, will have the best reward on non-optimal resource?, 2) Which of the jobs in the optimal resource queue, when moved, will suffer the least reward penalty on non-optimal resource? Based on these two factors, we have developed three new heuristics.

Last Optimal Reward (LOR): Recall that after initial placement of jobs (discussed in Section 7.2.2) in each queue, we computed Reward (Equation 7.4) for each job in each queue specific to its resource. These queues are sorted based on the Reward factor. Our first scheme, Last Optimal Reward, exploits this reward information. Suppose that a CPU becomes free, but the CPU queue is empty. So, we want to steal a job from the GPU queue to execute on the available (idle) CPU. Last Optimal Reward scheme chooses the last job from the GPU queue, to be executed on the CPU, i.e. the job that will have the least reward score on the optimal resource queue.

The reasoning behind this scheme is as follows. The last job in the optimal queue will anyway produce the least reward (or may be less urgent), among all current jobs, in its optimal resource. Thus, one could argue that there is likely the lowest amount of risk involved in moving this job into the non-optimal resource. Moreover, this job is likely to be executed much sooner in this case. What this simple scheme does not consider, however,
is the non-optimal penalty of the job to be moved with that of the other jobs in the queue. Therefore, if the last job happens to have a high non-optimal penalty, then this scheme can possibly suffer more loss than the yield.

**First Non-Optimal Reward (FNOR):** In contrast to the previous scheme ($LOR$), $FNOR$ exploits the reward the job could produce when executed on the non-optimal resource. Note that this scheme is very different from $LOR$ as it explicitly considers the non-optimal penalty of the job. Thus, $FNOR$ is based on the rationale that it is beneficial to move a job that will have the highest reward score on the non-optimal resource. In order to compute the reward score on the non-optimal resource, we combine the existing reward score of the job for optimal resource, with the penalty the job will suffer, if executed on the non-optimal resource. For each job $i$, we first determine the **Sufferage Factor**, defined as,

$$Sufferage\_Factor_i = \frac{Non - Optimal\_Walltime_i}{Optimal\_Walltime_i}$$  \hspace{1cm} (7.5)

Next, we compute the **Non-Optimal Reward** score, by combining the optimal Reward shown in Equation 7.4 and the sufferage factor, as follows:

$$Non - Optimal\_Reward_i = \frac{Optimal\_Reward_i}{Sufferage\_Factor_i}$$  \hspace{1cm} (7.6)

Note that sufferage factor is always greater than 1, hence, the **Non-Optimal Reward** score is always less than the **OptimalReward** score.

Finally, the job that gives the highest Non-Optimal Reward is chosen to be scheduled on the non-optimal resource. Consequently, if there are two jobs that suffer similar penalties when moved to the non-optimal resource, then this heuristic prefers the job that will result in the better reward, which corresponds to the more valuable and/or the more urgent job.

Unlike the previous scheme, there is more computational cost associated with this scheme. The complexity for computing the **Non-Optimal Reward** for each job in a queue
of length \( N \) is \( O(N) \), and moreover, the complexity of choosing the job with the highest Non-Optimal Reward is \( O(N\log(N)) \).

**Last Non-optimal Reward Penalty (LNORP)** In this scheme, we take the scheduling considerations one step further when compared to our previous scheme, \( FNOR \). Specifically, while \( FNOR \) gives preference to the job that has the highest reward score on the non-optimal resource, it fails to consider the reward score degradation experienced due to non-optimal penalty. Thus, if there are two jobs that have similar Non-Optimal Reward score, unlike \( FNOR \), \( LNORP \) will prefer to move the job that will have the lesser reward penalty. The Non-Optimal Reward Penalty (NORP) for the job \( i \), is computed from both Non-Optimal Reward and the Optimal Reward, which were shown earlier in Equations 7.6 and 7.4, respectively. NORP is calculated as follows:

\[
\text{NORP}_i = \text{OptimalReward}_i - \text{NonOptimalReward}_i
\]  

(7.7)

This heuristic can be better than \( FNOR \) when there are jobs with similar rewards, but vary in their degree of penalty on the non-optimal resource. Similar to \( FNOR \), this scheme also has a complexity of \( O(N\log(N)) \) for a queue size \( N \).

**Coordinated Scheme**

The three uncoordinated schemes described above consider moving a job to a different queue *only* when a resource is idle. These schemes can be effective in improving the aggregate yield by handling the imbalance and reducing the idle time of the resources. However, interestingly, in some scenarios, it may be necessary to move a job from one queue to another, even though neither of the resources is idle. Consider a case when there is an imbalance in the urgency of the jobs between the two queues, i.e, one queue may be skewed towards less urgent jobs, while the other queue is skewed towards more urgent jobs.
Then, the jobs in one queue will decay much faster than the other, rendering uncoordinated schemes ineffective due to loss of the global view of both the queues. This consideration leads to a new coordinated scheme.

We have developed a coordinated scheme that decides when to move a job from one queue to another, even when no resource is idle and when neither queue is empty. This scheme has two components: (i) A logic for detecting when coordination is required between the two queues, and (ii) As a result of coordination, a heuristic for which job to move into the non-optimal queue.

First, we describe the factor based on which coordination is invoked. Consider the causes of imbalance in the queue that makes one resource idle while the other is busy. Primarily, it is either because the number of jobs in one queue is more than the other, or that the length of execution of the jobs is much longer than the other. This can be captured by calculating the sum of the queuing delay for each job in the entire queue. Another potential factor causing imbalance is the decay rates across all jobs in each queue. Thus, in a case when imbalance is due to the queuing delay and/or the decay rate, then the yield of jobs in the heavy queue will be affected significantly. Hence, we introduce **Total Queuing-Delay Decay-Rate Product** ($TQDP$) as a metric for detecting when such coordination between queues is desirable. Formally, for each job $j$ and for each queue $i$, we calculate $TQDP$ as follows.

$$TQDP_i = \sum_{j=0}^{n} Queuing\_delay_j \times decay_j$$ \hspace{1cm} (7.8)

Note that the $queuing\_delay$ for each job is propagated based on its position in the queue. If the computed $TQDP$ varies significantly between the two queues, then, even if both the job queues are non-empty, a job from the heavy queue is executed on the non-optimal resource, until it reaches an acceptable imbalance threshold. The threshold for such imbalance can
be selected by the system. After this, the scheduler again resumes normal operation, where each resource is assigned a job only from the queue corresponding to its resource type. The critical advantage over the uncoordinated schemes is that such a fine-grained mechanism provides quicker attention to the rapidly decaying jobs, and consequently better user satisfaction.

*TQDP* can only indicate if there is an imbalance between the two queues. However, we still need to decide which job should be moved from the optimal queue to the non-optimal queue. For this purpose, we use the *Least Penalty* heuristic, i.e., we select the job that will have the least *Sufferage Factor* on the non-optimal resource, as calculated through Equation 7.5. Overall, we refer to this coordinated scheme as the **Coordinated Least Penalty (CORLP)** scheme.

### 7.2.4 Heuristics for Sharing

In this subsection, we describe our heuristic for sharing and the logic for determining when sharing will be valuable.

Our scheduler considers the possibility of having up to two jobs share a single resource (multi-core CPU or GPU) at any given time. In particular, we only allow *space-sharing* and do not let the jobs time-share the processor cycles. For instance, on a CPU with 8 cores, two jobs may each use 4 cores from the same CPU. Similarly, two jobs may each use 7 of the 14 SMs in a Tesla C2050/C2070 GPU. Our decision is motivated by the observation that space-sharing does not create a significant resource contention, whereas time-sharing may in some situations adversely impact the performances due to increased resource contentions [76].
The tradeoffs associated with sharing a resource is as follows. First, from the viewpoint of a single application, as it gets half of the resources, it is likely to incur a slowdown, which will impact the yield it obtains. Applications, as we can expect, have varying degree of scalability. We assume that through analytical models [57, 58] and/or profiling tools [39], it is possible to have a reasonable prediction of the scalability of a given application on a multi-core or many-core architecture. Thus, we assume that applications are characterized into three groups, i.e., those with low, medium, and high scalability.

Now, a job with a very low scalability beyond half the number of cores will have a low penalty due to sharing, whereas, a job with a good scalability beyond half the number of cores will have a high penalty. Thus, from the viewpoint of the aggregate yield, it is not preferable to share jobs for which yield decays rapidly, since yield gain for the other job sharing the resource will at most result in the same aggregate yield. In addition, it is also necessary to consider the overall load in the system. If the load is high, it will be desirable to have two low decay and/or low scalability jobs share one resource, so that another resource is available for a job that is highly scalable and/or has high priority.

Based on the above observations, our scheme for deciding which jobs should share a resource is as follows. Whenever a new batch of jobs arrives, after its initial placement (as described in section 7.2.2) into the queue, we calculate the predicted yield of the jobs based on its current position in the pending queue. If there are jobs with a very low or even negative yield, we take this as an indication of heavy load and enable sharing. Next, in each queue containing jobs with such low yield, we find the candidate jobs and mark them for sharing. However, in order to find the right candidates for sharing we need to consider both scalability factor and the decay rate of the job. Thus, we propose the concept of Scalability-Decay Factor. We first group the jobs under consideration in the order of
low, medium and highly scalable jobs. Next, within each group, the jobs are sorted in the ascending order of their decay rates. Such an ordering of jobs ensures that sharing is preferred for low scalable jobs and low decay jobs compared to highly scalable and high decay jobs respectively. However, in order to decide *how many* jobs to share, we parametrize the heuristic with a *Sharing Factor*. This factor dictates the fraction of jobs from the top of the pending queue that will be marked for sharing.

7.3 **Software Framework Design**

In this section, we provide details on the design and implementation of our scheduler.

![Scheduler Framework Design and its Components](image_url)

**Figure 7.2: Scheduler Framework Design and its Components**

(a) High-level Scheduler Framework Design

(b) Runtime Consolidation Framework For GPU Sharing
7.3.1 High-Level Scheduler Framework Design

The high-level framework design of our scheduling system is shown in Figure 7.2(a). Our scheduler follows the classic master-slave architecture, and consists of two major components: Cluster Level Scheduler (CLS) and Node Level Schedulers (NLS). The functionalities of these components are detailed below. We assume a shared file system across all nodes in the cluster. This is consistent with the configuration of most clusters and supercomputers in use today.

Cluster Level Scheduler (CLS): The CLS is installed on the master node, and has three responsibilities: (i) management and monitoring of the job queues, (ii) generation of scheduling decisions according to the heuristics described in Section 7.2, and (iii) communication of the scheduling decisions to the compute nodes.

The CLS maintains four types of job queues: Submission, Pending, Execution and Finished queues. All the jobs that are submitted by the users are placed into the submission queue. Since each job can be executed on a GPU or on a multi-core CPU, we maintain separate queues for these two types of resources. The order of scheduling of the jobs will depend on the pending queue. The execution and the finished queues are used for maintaining the status of the job, and for later analysis of the jobs’ execution. While maintaining two separate queues, we have two design choices. One possible design is to enqueue all the jobs in both the CPU and the GPU queues. This allows the flexibility of choosing a job from either queue. However, this implementation leads to the challenge of maintaining consistency among the queues, which can be expensive, and will not scale with the size of the cluster, the number of resource types, and the number of submitted jobs. Alternate design is to place a job into a single queue (either the CPU or the GPU queue), thereby eliminating the need for maintaining consistency. This implementation requires dynamic
schemes to move jobs between queues, which can be expensive in case of uneven load distribution (as described in sections 7.2.3 and 7.2.3). We chose this design as the frequency of such dynamic reassignments can be small.

The core functionality of the CLS is to perform scheduling decisions through the heuristics described in Section 7.2. Once the scheduling decisions are made, they will be communicated to the appropriate compute node by sending job-specific metadata over a TCP communication channel. To allow the CLS to maintain an up-to-date view of the resource utilization within the cluster, each compute node will notify the master node upon job completion.

Node Level Scheduler (NLS): The NLS is installed on all compute nodes in the cluster. Each compute node, whenever added to the cluster, registers itself with the master node. The NLS will fork a new process for each job assignment received by the CLS. In case of sharing, multiple CPU jobs will concurrently run on the compute node using the scheduling mechanisms provided by the underlying OS, and will thereby share the multi-core CPU. On the other hand, GPU sharing requires an additional software mechanism. To this end, we use the GPU runtime consolidation framework described below.

### 7.3.2 GPU Sharing Framework

We now briefly describe the GPU runtime consolidation framework used to enable GPU sharing within the NLS. The interested reader may refer to our recent work [76] for more details on GPU sharing mechanisms.

Until recently, task parallelism was not supported on NVIDIA GPUs (unless hard-coded within a single GPU kernel). It is only starting from GPUs with compute capability 2.0 (Fermi cards and later) that NVIDIA introduced a *concurrent kernel execution* feature, thus
enabling task parallelism across GPU kernels. This feature, offered by CUDA, is limited to kernels launched within a single process context, and therefore does not allow GPU sharing across distinct applications. To overcome this limitation, our software consolidation framework introduces the concept of virtual context, that represents an interface between the applications and the CUDA runtime. By mapping multiple applications onto the same virtual context, it is possible to enable the concurrent kernel execution feature offered by CUDA across application boundaries.

Virtual contexts provide a mechanism to enable GPU sharing across applications. However, GPU sharing is not always beneficial, and policies to dynamically determine which applications are suitable for sharing are required. To this end, we note that, similarly to CPUs, GPUs allow two kinds of sharing, namely space- and time-sharing. In the former case, co-scheduled kernels will be allocated a subset of the available streaming multiprocessors (SMs); in the latter, co-scheduled kernels will use the same computational resources during distinct compute cycles. Intuitively, space-sharing leads to less interference between applications (conflicts are mostly limited to the use of the available memory bandwidth), and is therefore preferable compared to time-sharing. Therefore, in this paper we restrict ourselves to space-sharing.

Our proposed GPU consolidation framework is represented in Figure 7.2(b). The framework relies on two software mechanisms: CUDA calls’ interception and redirection and virtual context management. CUDA calls’ interception allows handling applications at runtime, without any source code modifications. In particular, our framework leverages the interception mechanism provided by the open source project gVirtuS [41]. Following the split-driver model, CUDA calls are intercepted by a frontend library, and redirected to a backend demon (or server) running in the same OS space as the CUDA runtime. The
core functionalities of the GPU consolidation framework are implemented within two backend components: the Dispatcher and the Workload Consolidator. The Dispatcher inspects CUDA calls originated within the applications and redirected by the frontend library to the backend server. In the original design [76], the Dispatcher makes consolidation decisions based on the analysis of the execution configuration (number of thread blocks, number of threads, and shared memory usage) of the kernels. However, in our current proposal all scheduling (and consolidation) decisions are made by the CLS. Therefore, the Dispatcher will simply modify the execution configuration of the kernels to be consolidated in order to force space-sharing. In particular, it will set the number of thread blocks to half the number of available SMs. In addition, if the per-thread-block register and shared memory usage allows it, it will increase the number of threads per block so to have a total number of threads as close as possible to that in the original execution configuration. Next, the Workload Consolidator uses virtual contexts to allow concurrent kernel execution among different applications. In particular, the Workload Consolidator runs each kernel within a different CUDA stream, and defers synchronization calls until all consolidated kernels have completed execution. The virtual contexts created by the Workload Consolidator issue CUDA calls associated to them to the underlying CUDA Runtime.

7.4 Experimental Results

We first describe our evaluation methodology, and then present the results.

7.4.1 Evaluation Methodology

In this subsection, we provide details on our cluster hardware, benchmark applications, experimental setup, and the metrics we have considered in our evaluation.
**Cluster Hardware:** Our experiments are conducted on a cluster consisting of 16 heterogeneous nodes. Specifically, each machine is equipped with two Intel Quad-core Xeon E5520 CPUs (total of 8 cores, each running at 2.27 GHz), 48 GB main memory and an Nvidia Tesla C2050 (Fermi) GPU card. Each Tesla C2050 card has 14 streaming multiprocessors (SMs), each containing 32 cores (448 cores overall) running at 1.15 GHz, and 3 GB device memory. The machines are connected through an Infiniband network.

**Benchmarks:** In our evaluation, we use ten popular benchmark applications taken from different domains like scientific computations, financial analysis, data mining, and machine learning. These applications are listed in Figure 7.3. For each of them, we consider three different execution configurations: small dataset, large dataset, and large number of iterations (on large dataset). In Figure 7.3, we show the results obtained by running each benchmark program for each of the three configurations. Specifically, we report the sequential execution time (on CPU), as well as the GPU and CPU speedups relative to sequential execution. We now analyze the performance trends in these benchmarks. When small datasets are used, the majority of the benchmarks (7 out of 10) perform better on the multi-core CPU as compared to GPU. The trend changes when large datasets are used; in this case, the majority of the benchmarks (6 out of 10) run faster on the GPU. Two observations can be made from these results: first, the use of small datasets leads to a low GPU utilization; second, the initial GPU overheads are not well amortized over short execution times. In addition, when the number of iterations increases (third configuration), the ratio of data transfer overhead to the computation time decreases, leading to increased performance improvement on GPU (e.g. BlackScholes and MonteCarlo) as compared to CPU.
Experimental Setup, Baselines, and Metrics: Unless specified otherwise, our experiments have been conducted using a total of 256 jobs scheduled on our 16 node cluster. To the best of our knowledge, job traces submitted on heterogeneous (CPU-GPU) clusters are not available publicly. Thus, for our experiments, we generated a number of different workloads, by randomly selecting each job from the pool of 30 jobs described earlier. For the purpose of our experiments, we assume that the maximum value of each job is equivalent to its optimal walltime. In addition, the value of the job starts decaying beyond the optimal walltime. We have used a bimodal distribution of decay rates [22, 50]. Note that each of the data point has been collected after running the corresponding experiment for three times. In all cases the observed variation is less than 2.5% We compare our proposed scheduling schemes with two existing policies: an existing scheme used by current distribution of TORQUE [9], a popular scheduler for today’s heterogeneous clusters, and a published scheme, called Minimum Completion Time [69]. We refer to these two schemes
as TORQUE and MCT. TORQUE schedules by always assigning the job to the optimal resource. If the optimal resource is not available, then it waits until the requested resource is available. MCT assigns a job to the resource where it will complete the earliest depending on the waiting times on each resource. Our evaluation is based on both Value-based metric, Average Yield as well as TORQUE-based metric, Completion Time.

### 7.4.2 Comparison with TORQUE and MCT

**Comparison of Scheduling Schemes Based on Completion Time:** We first compare all the scheduling schemes based on completion time (referred as Comp. Time) and average latency (referred as Ave. Lat), which are the TORQUE-based metrics. The basic idea is to see how the TORQUE-based metrics affect our value-based scheduling schemes. The results are shown in the Figure 7.4. In this experiment, we use two different job mixes: Uniform Mix (UM) and Biased Mix (BM). In UM, the jobs are uniformly distributed across CPU and GPU queues, whereas in BM, the jobs are distributed with a bias (25%-75%) to the particular resource queue.
First, consider \textit{Comp. Time} metric. For the \textit{UM} case, we can see that \textit{LOR} performs only as good as \textit{TORQUE} as it is a very simple scheme and does not consider the penalty due to non-optimal assignment. However, more careful schemes like \textit{FNOR} and \textit{LNORP} performs about 10\% better than \textit{TORQUE}. This is because, the benefit from efficiently using the resources (zero idle time) outweighs the penalty due to the non-optimal assignment. In fact, the inefficiency due to resource idleness is further aggravated in the \textit{BM} case for \textit{TORQUE}. In that case, we can see that \textit{FNOR} and \textit{LNORP} can outperform \textit{TORQUE} by about 22\%. For both \textit{UM} and \textit{BM}, on the average, \textit{TORQUE} leaves each resource (16 nodes each containing a multi-core CPU and GPU) idle for 5\% and 14\% respectively. Next, consider \textit{Ave. Lat} metric. For both \textit{UM} and \textit{BM}, our schemes outperform the baselines by about 20\%. There are two reasons that explain the observation. First, our schemes may order the job queue with a preference for short jobs (see section 7.2.2). Next, we do not leave any resource idle and also can minimize the non-optimal penalty using the schemes like \textit{LNORP}.

Although, the schemes proposed in this paper are designed to improve yield, these results show that they can still outperform \textit{TORQUE} and \textit{MCT} using the \textit{TORQUE}-based metrics. Unless otherwise specified, for the remainder of this section, we use the value-based metric, \textit{Average Yield} with linear decay function.

\textbf{Impact of Varying CPU-GPU Job Mix on Yield:} We now study the effect of varying the ratio of CPU/GPU jobs in the workload (Figure 7.5). Specifically, we create three job mixes: GPU skewed (25C/75G), Uniform (50C/50G), and CPU skewed (75C/25G). Note that GPU skewed and CPU skewed job mixes will create imbalance in the resource queues. The results are shown in the Figure 7.5. Note that on Y-axis, we use the \textit{Average Yield} as the metric. From the results, we can see that our schemes outperform (up to a factor
of 8.8) both \textit{TORQUE} and \textit{MCT} for all job mixes. It is interesting to note that even with uniform mix our schemes outperform by a factor of 2.3 times. There are multiple reasons as to why our schemes give significant yield improvement over the baselines. First, the baseline schemes do not have the notion of value. Next, \textit{TORQUE} does a static mapping of jobs to the resources and leaves some resources idle even when they are available due to its inability to exploit flexible scheduling. On the other hand, our proposed schemes, use the notion of value to prioritize the jobs, flexibly schedule, and eliminate the idle times to improve the overall yield. Note that the improvement in yield is higher for skewed job mixes compared to the uniform job mix as a result of larger resource idleness with \textit{TORQUE}.

\textbf{Impact of Value Decay Functions on Yield:} In this experiment, we study the effect of two different value decay functions, namely, the linear and step decay functions respectively. The linear decay value function we use is described in Section 7.1.1. We next describe our step decay function. The job will achieve maximum value if the job is completed within a specific time limit, as expected, but beyond that, for every step (which is a delay by half the specified optimal walltime for that job), the value decreases by a certain fraction. The
results from the experiments are shown in the Figure 7.6. We can see that our proposed schemes perform significantly better compared to the two baseline schemes (TORQUE and MCT). Particularly, for linear and step decay functions, our schemes outperform by nearly a factor of 3.9 and 7 times respectively. Yield improvement is better with step decay compared to linear decay as the penalty due to delay in step decay function is more coarse-grained. This shows that our scheduling schemes can be quite adaptive to different value functions.

**Impact of Increasing System Load on Yield:** We now study the effect of increasing load (and the corresponding scalability in yield) on these scheduling schemes. The results are
shown in Figure 7.7. In this experiment, we vary the number of jobs from 128 to 512. As the load increases, the gap in the relative yield achieved by the baseline schemes and our proposed schemes increases linearly (a factor of 8.2 at 512 jobs). In particular, beyond 128 jobs, the average yield of TORQUE and MCT starts to decrease linearly, while, on the other hand, it is observed that our schemes achieve a sustained average yield, despite heavy load. This can be explained by observing that the uncoordinated and coordinated schemes take into account decay rate and value while minimizing the imbalance among resources, and hence, can sustain heavy loads. In most cases, our proposed schemes are comparable, and at 512 jobs, CORLP leads to the best yield.

### 7.4.3 Benefit from Enabling Sharing

In this section, we study the effect of resource sharing on the yield. From the benchmarks analysis reported in Figure 7.3, we found that approximately 30% of the jobs have sub-linear scalability (low and medium) on both CPU and GPU. Thus, in the following experiments, we fixed the fraction of jobs that exhibit low and medium scalability to 10% and 20% (a total of 30%), respectively. As explained in section 7.2.4, the sharing heuristic is guided by two parameters: (i) the tunable Sharing K Factor, and the (ii) Scalability-Decay Factor. The former determines how many jobs will execute on a shared mode, and the latter decides the candidates for sharing.

We first study the impact of sharing due to various K factors. The results are shown in Figure 7.8, where we report the improvement obtained from sharing compared to the best of the four uncoordinated and coordinated schemes. For this experiment, we show the impact of sharing on CPU only, GPU only, and both CPU and GPU. At lower values of K (0.1 and 0.2), the gains from sharing appear marginal. However, as we increase the K
Figure 7.8: Impact of Varying K-Factor on Sharing (Higher is Better)

factor, the yield improves until K=0.5. At K=0.6, more sharing leads to increasing penalty and hence, benefit starts decreasing. This show that in both cases, when the amount of sharing is too less or too much, the penalty due to sharing is significant. This experiment emphasizes a careful selection of $K$ value, and show that at K=0.5, we can achieve an yield improvement of about 22.1% due to sharing.

Another potential advantage of sharing is an added tolerance towards heavy loads. We verify this benefit with increasing loads, and the results are shown in Figure 7.9. Thus, as we increase the load from 128 to 512 jobs, more jobs are subject to high queuing delays, which impact the yield of the jobs. Allowing certain jobs to share a resource ensures that more resources will be available for high priority and urgent jobs. Moreover, using our heuristic, only the jobs with low or medium scalability, and/or jobs with low decay rate are preferred for sharing. Overall, we can see that sharing allows a sustained improvement in the yield (up to 23%), as compared to the best of our uncoordinated and coordinated schemes.
7.4.4 Overhead of Sharing a CPU Core

When we schedule a job on the multi-core CPU and a job on the GPU simultaneously within the same node, one CPU core will be shared by these two jobs. In this subsection, we evaluate the overhead of sharing a CPU core. The results are presented in the Figure 7.10. For various job mixes, where each mix containing 256 jobs are scheduled on 16 nodes, we measure the slowdown caused due to sharing one CPU core. Note that this does not include other slowdowns due to queuing delay or scheduling on a non-optimal resource. The overheads are averaged over all the jobs. We observed that for 6 different job mixes, the overhead varies between 3.7% to 8.8% with a geometric mean of 5.5%. The variation in overhead is attributed to the nature of the GPU jobs and its data transfer pattern with the CPU (or host). The overhead increases with the frequency of data transfer between GPU and CPU. Overall, low overheads observed from our experiment indicates that indeed, it is desirable to schedule the CPU and GPU jobs simultaneously on the same node. Moreover, it also increases the utilization without significantly affecting the overall aggregate yield.

7.4.5 Summary of Results

To summarize, using TORQUE-based metrics, we show that our proposed value-based schemes can still outperform TORQUE by about 22%. Using Average Yield (value-based) as the metric, we demonstrate significant gain in the yield (up to a factor of 8.8 times) using
the four proposed schemes when compared to the baseline schemes, TORQUE and MCT, depending on various factors like workload mix, value decay functions, and system load. Finally, we also show that by judiciously enabling sharing of resources we get a sustained improvement in the yield of up to 23% even at high system load.

### 7.5 Conclusions

A wide adoption of heterogeneous computing has led to the evolution of programming models like OpenCL that offer portability across multi-core CPUs and GPUs. Batch schedulers for heterogeneous clusters that are in practice today neither exploit such portability nor pay adequate attention to user satisfaction or profitability in their scheduling considerations.

In this paper, we have revisited the problem of scheduling independent jobs on heterogeneous computing clusters. Particularly, our formulation uses a notion of value function or yield that captures both the *urgency* and the *relative priority* of the jobs.
We propose four novel scheduling schemes to dynamically map the jobs on to the heterogeneous resources. In addition, we also propose a novel heuristic that enables efficient resource sharing on multi-core CPUs and GPUs. We have extensively evaluated our schemes using our scheduler framework. Our comparisons against two baseline schemes, TORQUE and MCT, show that, a factor of up to 8.8 times yield improvement can be achieved. Finally, we show that by judiciously sharing the resources, we achieve an yield improvement of up to 23%.
Chapter 8: Conclusions and Future Work

A trend that has materialized, and has given rise to much attention, is of the increasingly heterogeneous computing platforms. Such heterogeneous platforms consists of multi-core CPUs and accelerators like GPUs integrated into a single machine. Today, the CPU-GPU heterogeneous architecture is widely used starting from desktops and notebooks to fastest supercomputers. While these machines demonstrate huge potential for computational power, most of the application development is still unable to effectively utilize the aggregate computation power of CPU and GPU simultaneously.

Driven by the emergence of GPUs as a major player in high performance computing and the rapidly growing popularity of cloud environments, GPU instances are being offered by cloud providers. In a cloud environment, the important goal is the overall throughput for a set of applications as opposed to single application performance. The challenge of making GPU a true shared resource in the cloud has not yet been addressed. Moreover, the problem of optimizing the overall throughput for a set of applications deployed on a cluster of CPU-GPU nodes has not been well studied.

We have formulated and revisited the problem of scheduling to optimize the overall throughput of a set of applications deployed on a cluster of heterogeneous nodes comprising multi-core CPUs as well as many-core GPUs. Specifically, we have addressed two scheduling problems: one targeting single-node jobs and the other targeting multi-node
A wide adoption of heterogeneous computing has led to the evolution of programming models like OpenCL that offer portability across multi-core CPUs and GPUs. Batch schedulers for heterogeneous clusters that are in practice today neither exploit such portability nor pay adequate attention to user satisfaction or profitability in their scheduling considerations.

- We first developed a framework and runtime support targeting generalized reductions for heterogeneous configurations. Our runtime system automatically maps the computations to the heterogeneous processing elements and can utilize the aggregate computation power of CPU and GPU. We also developed dynamic work distribution scheme that can effectively realize the power of such platforms. The proposed work distribution scheme minimizes the overheads associated with dynamic scheduling in the heterogeneous setup.

- Next, we described a general dynamic scheduling framework for data parallel loops geared towards CPU-GPU heterogeneous machines. Our approach involves analysis of architectural trade-offs and communication patterns in the loop. Based on the analysis, we have developed a cost model for predicting the optimal chunk size, by balancing the performance critical factors. In addition to the general framework, we derive optimized instances of dynamic work distribution schemes for applications involving two distinct communication patterns, and show that, these schemes incur low overhead and can adapt to different heterogeneous configurations.

- Then, we presented a framework for transparently enabling one or more GPUs to be shared by applications in cloud environments. We have both extended an existing open source software to provide mechanisms for enabling consolidation, as well as
provide solutions to the conceptual problem of consolidation. We have developed
a method for computing the affinity score between two or more kernels, indicating
the improvement in throughput when they are consolidated. We have also introduced
two new molding methods, where a kernel’s execution configuration can be modified. Based on the affinity score and considering the possibility of molding, we have developed an algorithm for mapping a set of given kernels on GPUs.

- Next, we have developed a number of novel scheduling algorithms to enable concurrent job scheduling to improve global throughput on CPU-GPU clusters. Our proposed schemes are based on different amount of information expected from the user, and focus on different tradeoffs between application wait and completion times. Our scheduling solutions also consider molding of jobs along two dimensions, which are the resource type and the number of requested nodes.

- Finally, we have proposed four novel value-based scheduling schemes to dynamically map the jobs on to the heterogeneous resources, broadly categorized as Uncoordinated (three schemes) and Coordinated (one scheme). These schemes use resource-specific queues and operate by minimizing the imbalance between them. In our coordinated scheme, we allow periodic monitoring of the state of each queue to dynamically readjust the mapping and scheduling at a finer granularity. Uncoordinated schemes lack this periodic monitoring; however, they can dynamically readjust the mapping and scheduling at a coarser-granularity. In addition, we also propose a novel heuristic that enables efficient resource sharing on multi-core CPUs and GPUs.

While we have addressed some of the key problems in the emerging heterogeneous architectures, many more challenges remain open. First, in the future, we would like to
build upon our current proposed solutions to address more sophisticated formulations. For instance, in chapter 7, our scheduling schemes are limited to the allocation of one resource (either CPU or GPU) within a node. These scheduling schemes can be extended to consider non-trivial formulations like allocation of both CPU and GPU (within a node) and across multiple nodes. Next, while discrete CPU-GPU architectures have been the most popular form of heterogeneous architectures, other architectures in the form of integrated CPU-GPU architectures are emerging. The single most attractive feature of this architecture is to eliminate the data transfer over high-cost PCI express. However, such an architecture will mark the emergence of machines with both integrated and discrete GPUs within the same node, requiring more sophisticated work distribution and scheduling strategies.

Another approach to extend our work is by considering different goals for our existing scheduling formulations. For example, in all of the problems addressed in this thesis we focused on improving the performance of either a single or a set of applications. While performance is critical, energy considerations are also becoming important with exascale computing. We believe that these heterogeneous architectures (with multi-core CPU, discrete and integrate GPUs) provide a lot of flexibility and opportunity in developing energy-efficient work distribution and scheduling schemes.
Bibliography


[10] Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU, 2010.


