Accelerating Applications with Pattern-specific Optimizations on Accelerators and Coprocessors

Dissertation

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

By

Linchuan Chen, B.S., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2015

Dissertation Committee:

Dr. Gagan Agrawal, Advisor
Dr. P. Sadayappan
Dr. Feng Qin
Abstract

Because of the bottleneck in the increase of clock frequency, multi-cores emerged as a way of improving the overall performance of CPUs. In the recent decade, many-cores begin to play a more and more important role in scientific computing. The highly cost-effective nature of many-cores makes them extremely suitable for data-intensive computations. Specifically, many-cores are in the forms of GPUs (e.g., NVIDIA or AMD GPUs) and more recently, coprocessers (Intel MIC). Even though these highly parallel architectures offer significant amount of computation power, it is very hard to program them, and harder to fully exploit the computation power of them. Combing the power of multi-cores and many-cores, i.e., making use of the heterogeneous cores is extremely complicated.

Our efforts have been made on performing optimizations to important sets of applications on such parallel systems. We address this issue from the perspective of communication patterns. Scientific applications can be classified based on the properties (communication patterns), which have been specified in the Berkeley Dwarfs many years ago. By investigating the characteristics of each class, we are able to derive efficient execution strategies, across different levels of the parallelism. We design a high-level programming API, as well as implement an efficient runtime system with pattern-specific optimizations, considering the characteristics of the hardware platform. Thus, instead of providing a general programming model, we provide separate APIs for each communication pattern.
We have worked on a selected subset of the communication patterns, including MapReduce, generalized reductions, irregular reductions, stencil computations and graph processing. Our targeted platforms are single GPUs, coupled CPU-GPUs, heterogeneous clusters, and Intel Xeon Phis. Our work not only focuses on efficiently executing a communication pattern on a single multi-core or many-core, but also considers inter-device and inter-node task scheduling. While implementing a specific communication pattern, we consider aspects including lock-reducing, data locality, and load balancing.

Our work starts with the optimization of the MapReduce on a single GPU, specifically aiming to efficiently utilize the shared memory. We design a reduction based approach, which is able to keep the memory consumption low by avoiding the storage of intermediate key-value pairs. To support such an approach, we design a general data structure, referred to as the reduction object, which is placed in the memory hierarchy of the GPU. The limited memory requirement of the reduction object allows us to extensively utilize the small but fast shared memory. Our approach performs well for a popular set of MapReduce applications, especially the reduction intensive ones. The comparison with former state-of-art accelerator based approaches shows that our approach is much more efficient at utilizing the shared memory.

Even though MapReduce significantly reduces the complexity of parallel programming, it is not easy to achieve efficient execution, for complicated applications, on heterogeneous clusters with multi-core and multiple GPUs within each node. In view of this, we design a programming framework, which aims to reduce the programming difficulty, as well as provide automatic optimizations to applications. Our approach is to classify applications based on communication patterns. The patterns we study include Generalized Reductions.
Irregular Reductions and Stencil Computations, which are important ones that are frequently used in scientific and data intensive computations. For each pattern, we design a simple API, as well as a runtime with pattern-specific optimizations at different parallelism levels.

Besides, we also investigate graph applications. We design a graph processing system over the Intel Xeon Phi and CPU. We design a vertex-centric programming API, and a novel condensed static message buffer that supports less memory consumption and SIMD message reduction. We also use a pipelining scheme to avoid frequent locking. The hybrid graph partitioning is able to achieve load balance between CPU and Xeon Phi, as well as to reduce the communication overhead.

Executing irregular applications on SIMD architectures is always challenging. The irregularity leads to problems including poor data access locality, data dependency, as well as inefficient utilization of SIMD lanes. We propose a general optimization methodology for irregular applications, including irregular reductions, graph algorithms and sparse matrix matrix multiplications. The key observation of our approach is that the major data structures accessed by irregular applications can be treated as sparse matrices. The steps of our methodology include: matrix tiling, data access pattern identification, and conflict removal. As a consequence, our approach is able to efficiently utilize both SIMD and MIMD parallelism on the Intel Xeon Phi.
This is dedicated to the ones I love: my grandmother, my parents and my sister.
Acknowledgments

There are so many people whom I want to extend my deepest appreciation to. It would not have been possible for me to complete my dissertation, without their advise, help, and support. To only some of them, it is possible to give particular mention here.

I would first thank my advisor, Prof. Gagan Agrawal for his guidance. This dissertation would not have been possible without his direction, encouragement and patience. His broad knowledge and extraordinary foresight in the area of high performance computing, as well as his insightful advice, not only established my interest in my research area, but also helped me to develop the capability of identifying and solving problems, which would benefit my future career. At the same time, I sincerely appreciate all his supports, both financial and spiritual, without which it would have been hard for me to pass through the ups and downs during my Ph.D. study. He is not only a successful educator and computer scientist, but also an excellent example for my life. I would also like to thank my thesis committee members, Drs. P. Sadayappan, Feng Qin, and Christopher Charles Stewart. Their incisive and critical analysis, as well as intensive interest have significantly helped me with my research.

Thanks to my host family, Woody and Dorothy, the nice couple who provided warm and friendly help when I first arrived in Columbus five years ago. It was their hospitality that helped me to quickly adapt to the new place.

I also would like to thank Dr. Kalluri Eswar and Dr. Zhaohui Fu, who helped me to significantly improve my programming skills while I was doing my internship at Google.
I also appreciate the support from National Science Foundation, Ohio State, as well as Department of Computer Science and Engineering, especially the head of our department, Dr. Xiaodong Zhang, and all friendly staffs in our department, especially Kathryn M. Reeves, Catrena Collins and Lynn Lyons.

I also would like to thank the colleagues in our group, Data-Intensive and High Performance Computing Research Group. My Ph.D. life would not have been so wonderful without the friendship from them. Former and present members of our lab have provided me with valuable support and help, in both everyday life and research, and they are: Wei Jiang, Vignesh Trichy Ravi, Tantan Liu, Bin Ren, Xin Huo, Tekin Bicer, Yu Su, Yi Wang, Mehmet Can Kurt, Mucahid Kutlu, Sameh Shohdy, Jiaqi Liu, Roee Ebenstein, Surabhi Jain, Gangyi Zhu, Peng Jiang and Jiankai Sun.

Finally, special thanks are extended to my parents and my sister, who have been always supportive. I am also thankful to my friends out of our department, as well as the friends who are in China.
Vita

December 5th, 1988  . . . . . . . . . . . . . . . . . . . . . . . . . Born - Ganyu, China

2010  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B.S. Software Engineering,
Nanjing University, Nanjing, China

2014  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . M.S. Computer Science & Engineering,
Ohio State University, Columbus, OH

2014  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Software Engineering Intern, Google,
Mountain View, CA

Publications

Research 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>v</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vi</td>
</tr>
<tr>
<td>Vita</td>
<td>viii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xvi</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xix</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Parallel Architectures</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Communication Patterns</td>
<td>7</td>
</tr>
<tr>
<td>1.2.1 MapReduce</td>
<td>7</td>
</tr>
<tr>
<td>1.2.2 Generalized Reductions</td>
<td>8</td>
</tr>
<tr>
<td>1.2.3 Irregular Reductions</td>
<td>8</td>
</tr>
<tr>
<td>1.2.4 Stencil Computations</td>
<td>9</td>
</tr>
<tr>
<td>1.2.5 Graph Processing</td>
<td>10</td>
</tr>
<tr>
<td>1.3 Dissertation Contributions</td>
<td>12</td>
</tr>
<tr>
<td>1.3.1 MapReduce for GPUs with Effective Shared Memory Usage</td>
<td>12</td>
</tr>
<tr>
<td>1.3.2 MapReduce for a Coupled CPU-GPU Architecture</td>
<td>13</td>
</tr>
<tr>
<td>1.3.3 Scheduling Applications on Architectures with Heterogeneous Cores</td>
<td>14</td>
</tr>
<tr>
<td>1.3.4 Pattern Specification Framework for Accelerating Scientific Computations on Heterogeneous Clusters</td>
<td>15</td>
</tr>
</tbody>
</table>
4.1 Problem Definition .......................................................... 80
4.2 Dynamic Scheduling Methods .......................................... 82
   4.2.1 Master-worker Scheduling Method .............................. 82
   4.2.2 Core-level Token Passing Method ............................... 84
   4.2.3 Device-level Token Passing Method ............................ 86
4.3 Experimental Results .................................................. 88
   4.3.1 Experimental Setup ................................................. 89
   4.3.2 Applications Used .................................................. 89
   4.3.3 Overall Performance Improvements and Comparison of Different Scheduling Methods ................................. 91
   4.3.4 Detailed Comparison of Core-level and Device-level Token Passing Methods ........................................... 97
   4.3.5 Scaling on Multiple GPUs ......................................... 98
   4.3.6 Comparison with StarPU and OmpSs ................................ 100
4.4 Summary ................................................................. 103

5. A Pattern Specification and Optimizations Framework for Accelerating Scientific Computations on Heterogeneous Clusters ........................................ 104
   5.1 Programming Model .................................................. 104
      5.1.1 Communication Patterns and APIs ............................ 105
      5.1.2 Case Study: A Comprehensive Example Involving Multiple Communication Patterns ...................................... 109
      5.1.3 Discussion .......................................................... 110
   5.2 Framework Implementation and Pattern Specific Optimizations .......................................................... 112
      5.2.1 Code Organization and Compilation ............................. 112
      5.2.2 Runtime Execution Flow ......................................... 113
      5.2.3 Distributed Memory Execution ................................... 114
      5.2.4 CPU-GPU Workload Partitioning ................................. 118
      5.2.5 GPU and CPU Execution .......................................... 120
   5.3 Experimental Results ................................................ 123
      5.3.1 Applications Used ................................................ 123
      5.3.2 Code Sizes ......................................................... 125
      5.3.3 Overall Performance from the Framework ................. 126
      5.3.4 Benefits from Optimizations ..................................... 129
      5.3.5 Comparison with GPU Benchmarks ......................... 129
   5.4 Summary ................................................................. 130

6. Efficient and Simplified Parallel Graph Processing over CPU and MIC .......... 132
   6.1 Programming API ..................................................... 132
   6.2 Framework Implementation .......................................... 136

xii
6.2.1 System Components and Workflow ........................................ 136
6.2.2 Message Buffer Design .................................................... 138
6.2.3 Processing of Messages Using CSB ..................................... 140
6.2.4 Intra-device Load Balancing ............................................. 144
6.2.5 Graph Partitioning between CPU and MIC .......................... 145
6.3 Experimental Results ....................................................... 147
  6.3.1 Evaluation Environment ............................................... 147
  6.3.2 Applications Used ..................................................... 148
  6.3.3 Overall Performance ................................................ 151
  6.3.4 Benefits of SIMD Execution for Message Processing ............ 154
  6.3.5 Effect of Hybrid Graph Partitioning for CPU-MIC Execution .. 155
  6.3.6 Overall Performance Gained from Our Framework .............. 156
6.4 Summary ........................................................................ 157

7. Exploiting Recent SIMD Architectural Advances for Irregular Applications .... 159
  7.1 Exploiting New SIMD Architectural Features for Irregular Applications . 159
    7.1.1 Irregular Reductions .................................................. 159
    7.1.2 Graph Algorithms ................................................... 163
    7.1.3 Sparse Matrix-Matrix Multiplication .............................. 164
    7.1.4 Matrix Reordering for Irregular Reductions and Graph Algorithms 168
    7.1.5 Discussion ............................................................ 168
  7.2 Evaluation Results ......................................................... 169
    7.2.1 Evaluation Environment ............................................. 169
    7.2.2 Applications Used ................................................... 170
    7.2.3 SIMD Utilization Ratio ............................................. 172
    7.2.4 Results from Irregular Reductions ................................. 173
    7.2.5 Results from Graph Algorithms ................................... 177
    7.2.6 Results from SpMM .................................................. 179
  7.3 Summary ........................................................................ 182

8. Related Work .................................................................... 183
  8.1 MapReduce Frameworks for Heterogeneous Architectures ............ 183
  8.2 Task Scheduling Frameworks for Parallel Systems ................... 184
  8.3 Parallel Programming Systems and Languages for Communication Patterns We Have Worked on ................................................. 186
  8.4 Optimizing Irregular Applications on Diverse Parallel Platforms .... 189

9. Conclusions ..................................................................... 192
  9.1 Contributions ................................................................ 192
9.2 Future Work ........................................... 194
  9.2.1 Supporting Irregular Applications with Adaptive Data Structures on SIMD Architectures .................................. 194
  9.2.2 Supporting the New MIC Architectures ...................... 195

Bibliography ....................................................... 197
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Applications and Datasets Used in our Experiments</td>
<td>42</td>
</tr>
<tr>
<td>3.1</td>
<td>Worker Info Element</td>
<td>62</td>
</tr>
<tr>
<td>3.2</td>
<td>Applications and Datasets Used in our Experiments</td>
<td>70</td>
</tr>
<tr>
<td>3.3</td>
<td>Parallel Efficiency Obtained from Our Framework</td>
<td>77</td>
</tr>
<tr>
<td>5.1</td>
<td>User-defined Functions for Communication Patterns</td>
<td>106</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparison of Intra-node Scalability: Perfect (Assuming No Scheduling Overheads) and Actual</td>
<td>128</td>
</tr>
<tr>
<td>6.1</td>
<td>Messages Being Sent in the Example Graph</td>
<td>140</td>
</tr>
<tr>
<td>6.2</td>
<td>Parallel Efficiency Obtained from Our Framework</td>
<td>157</td>
</tr>
<tr>
<td>7.1</td>
<td>Applications and Datasets Used in the Experiments</td>
<td>170</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>An Example Graph and Its Representation in CSR Format</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>Memory Hierarchy</td>
<td>26</td>
</tr>
<tr>
<td>2.2</td>
<td>Reduction Object</td>
<td>29</td>
</tr>
<tr>
<td>2.3</td>
<td>Performance Under Different Group Configurations</td>
<td>43</td>
</tr>
<tr>
<td>2.4</td>
<td>Speedup over Sequential Implementations</td>
<td>46</td>
</tr>
<tr>
<td>2.5</td>
<td>Speedup over MapCG for Reduction-intensive Applications</td>
<td>47</td>
</tr>
<tr>
<td>2.6</td>
<td>Speedup over MapCG for Other Applications</td>
<td>48</td>
</tr>
<tr>
<td>2.7</td>
<td>Speedup over Ji et al.’s Work</td>
<td>50</td>
</tr>
<tr>
<td>2.8</td>
<td>Swap Frequency under Different Number of Distinct Keys</td>
<td>51</td>
</tr>
<tr>
<td>2.9</td>
<td>Comparison with Other Implementations for WC</td>
<td>52</td>
</tr>
<tr>
<td>2.10</td>
<td>Comparison with MapCG for PVC</td>
<td>52</td>
</tr>
<tr>
<td>3.1</td>
<td>Map-dividing Scheme</td>
<td>59</td>
</tr>
<tr>
<td>3.2</td>
<td>Pipelining Scheme</td>
<td>64</td>
</tr>
<tr>
<td>3.3</td>
<td>Load Imbalance under Different Task Block Sizes</td>
<td>71</td>
</tr>
<tr>
<td>3.4</td>
<td>Computation Time under Different Task Block Sizes</td>
<td>72</td>
</tr>
</tbody>
</table>
Section 3.5: Execution Time of Different Schemes for Five Applications (KM, WC, NBC, MM, and kNN) .................................................. 73

Section 4.1: Master-worker Scheduling Method ........................................ 83

Section 4.2: Core-level Token Passing Method ........................................ 85

Section 4.3: Device-level Token Passing Method ..................................... 87

Section 4.4: Execution Time of Different Versions on Coupled CPU-GPU ....... 92

Section 4.5: Execution Time of Different Versions on Decoupled CPU-GPU .... 94

Section 4.6: Computation Time for Core-level Token Passing Method and Device-level Token Passing Method while Increasing the Number of Cores/SMs .................................................... 96

Section 4.7: Scaling to Multiple GPUs .................................................. 98

Section 4.8: Comparison with StarPU and OmpSs ..................................... 101

Section 5.1: An Irregular Reduction Loop ............................................. 107

Section 5.2: Code-level Structure of the Framework ................................ 112

Section 5.3: Arrangement of Nodes for Irregular Reductions. Solid lines: local edges, Dashed lines: cross edges ........................................... 114

Section 5.4: Time Sequence of the Stencil Runtime for Heterogeneous Executions ................................................................. 116

Section 5.5: Intra-node and Inter-node Scalability and Comparison Against Hand-written MPI Applications ............................................. 122

Section 5.6: Comparison of Code Sizes between Handwritten (MPI) Benchmarks and Versions Using our Framework .................................. 125

Section 5.7: Effect of Optimizations for Irregular Reductions and Stencil Computations ................................................................. 129

Section 5.8: Comparison with Benchmarks on a Fermi GPU ......................... 130

Section 6.1: System Components .......................................................... 137
6.2 Condensed Static Buffer (CSB) and Message Insertion Methods ........................................ 141
6.3 Message Generation Pipelining Using Worker and Mover Threads ..................................... 144
6.4 (a) - (e): Comparison of Different Versions of Five Applications. (f): Impact of SIMD Processing. ................................................................. 148
6.5 Impact of Different Graph Partitioning Methods ................................................................. 155
7.1 Access Patterns of Irregular Reductions and Graph Algorithms ........................................ 160
7.2 SIMDized Multiplication between Two $2 \times 2$ Tiles ......................................................... 166
7.3 SIMD Utilization Ratios of Datasets (Used by Irregular Reductions and Graph Algorithms) under Different Tile Sizes ............................................. 172
7.4 Single Thread SIMD Execution Times of Moldyn and Euler under Different Tile Sizes ........ 174
7.5 Single Thread Performance of Moldyn and Euler with Different Approaches .................... 174
7.6 SIMD+MIMD Speedups of Irregular Reductions over Serial (Single Thread Scalar) Execution ................................................................. 176
7.7 Single Thread SIMD Execution Times of Bellman-Ford and PageRank under Different Tile Sizes ................................................................. 177
7.8 Single Thread Performance with Different Approaches. ..................................................... 178
7.9 SIMD+MIMD Speedups of Bellman-Ford and PageRank over Serial (Single Thread Scalar) Execution ................................................................. 179
7.10 Single Thread Performance of SpMM with Clustered Datasets using Different Data Types. ................................................................. 180
7.11 Speedups of SpMM with MIMD+SIMD Parallelism over Single Thread Scalar Execution .... 181
List of Algorithms

1. map\( (input, offset) \) .................................................. 24
2. reduce\( (value_1, value_1\_size, value_2, value_2\_size) \) .................. 24
3. insert\( (key, key\_size, val, val\_size) \) .................................. 31
4. Overflow Handling and Swapping Mechanism ............................... 33
5. Task Scheduling in the Map-dividing Scheme ................................. 63
6. Task Scheduling in the Pipelining Scheme with Dynamic Load Balancing . 66
7. Task Scheduling in the Core-level Token Passing Method for GPU Side . 84
8. Task Scheduling in the Device-level Token Passing Method for GPU Side . 87
9. Transforming a Tile to Conflict-free Groups .................................. 161
Chapter 1: Introduction

Our overall research work has been motivated by three trends in recent years.

First, in the recent decade, pure graph processing units have evolved the ability to support general purpose computing. These highly parallel architectures are able to conduct data-intensive computing using SIMD parallelism. Modern GPUs are promising hardware platforms in that they are suitable for large scale and cost effective high performance computing. Second, GPUs need to be launched by a host. The combination of GPU(s) and the host CPU forms a heterogeneous hardware platform. GPUs and CPUs are different in that their computing properties and memory access patterns are distinct from each other. These differences lead to a series of challenges in fully utilizing both CPU and GPU for a single application. The emergence of integrated CPU-GPUs, represented by the AMD Fusion A-PU and the Intel Ivy Bridge, provides a new way of exploiting both CPU and GPU using a shared memory architecture. Lastly, Intel MIC is a relatively newer accelerator (coprocessor). It provides large scale shared memory parallelism in the form of a large amount of cores, with each supporting wide SIMD lanes.

Even though the newly emerging architectures provide to us tremendous amount of computing power, compared to the traditional HPC platforms, nowadays programming such architectures still needs more considerations on both programmer productivity and performance efficiency. Efficient execution on GPUs needs to consider reducing locking
overhead, as well as efficient utilization of memory hierarchy. Heterogeneous execution using both CPU and GPU requires judicious task scheduling between the CPU and the GPU. While programming the Intel MIC, it is very hard to avoid contention, inefficient memory access and inefficient utilization of SIMD, especially for irregular applications such as irregular reductions and graph processing.

Based on these observations, we believe that improving the programmability on these new architectures is a crucial issue to solve. Our goal is not only to offer high-level APIs, but also offering high performance underlying mechanisms such as efficient runtime and optimizations. We approach this by investigating applications based on communication patterns. For each communication pattern, we work on programming API, memory efficiency, locking control, and task scheduling (either on a single processing unit or in heterogeneous environments).

In this chapter, we would like to introduce the background knowledge about the parallel architectures, the communication patterns, and some famous parallel programming models such as MapReduce and Pregel. Then we will take an overview of the current work. The work we will introduce here include: two MapReduce frameworks on GPUs, and on a coupled CPU-GPU, an extended MapReduce-like framework supporting pattern-specific optimizations for heterogeneous clusters, as well as a graph processing framework for the Intel Xeon Phi. We will also introduce a work for efficient task scheduling on heterogeneous chips.

1.1 Parallel Architectures

This section introduces common parallel systems, including multi-core CPUs, discrete GPUs, coupled CPU-GPU, and Intel MIC.
**Multi-core CPUs:** Multi-core CPUs are chips integrating multiple X86 processing cores. Each core is a complicated processing component embedded with hardware optimizations such as instruction reordering. Recent multi-cores from Intel support hyperthreading, i.e., each core supports two or more hardware threads. Also, SSE processing is also being supported by both Intel and AMD multi-core CPUs. Multi-cores are an important processing power even compared with many-core GPUs in that multi-core systems are typically not limited by the memory size, and that they are efficient in handling loops or conditional instructions because of the hardware optimizations, also, each core has a much higher clock rate than a core on the many-core systems. Multi-core systems usually have multi-level cache, typically in the form of private L1 cache on each core and shared L2 and L3 caches.

**Discrete GPUs:** within the last 5-6 years, GPUs have emerged as the means for achieving extreme-scale, cost-effective, and power-efficient high performance computing. On one hand, some of the fastest machines in the world today are based on NVIDIA GPUs. At the same time, the very favorable price to performance ratio offered by the GPUs is bringing supercomputing to the masses. It is common for the desktops and laptops today to have a GPU, which can be used for accelerating a compute-intensive application. 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 Gigaflop. Yet another key advantage of GPUs is the very favorable power to performance ratio.

A modern Graphical Processing Unit (GPU) architecture consists of two major components, i.e., the processing component and the memory component. The processing component in a typical GPU is composed of a certain number of streaming multiprocessors. Each streaming multiprocessor, in turn, contains a set of simple cores that perform in-order processing of the instructions. To achieve high performance, a large number of threads,
typically a few tens of thousands, are launched. These threads execute the same operation
on different sets of data. A block of threads are mapped to and executed on a streaming
multiprocessor. Furthermore, threads within a block are divided into multiple groups,
termed as warp. Each warp of threads are co-scheduled on the streaming multiprocessor
and execute the same instruction in a given clock cycle (SIMD execution).

The memory component of a modern GPU-based computing system typically contains
several layers. One is the host memory, which is available on the CPU main memory.
This is essential as any general purpose GPU computation can only be launched from the
CPU. The second layer is the device memory, which resides on the GPU card. This repre-
sents the global memory on a GPU and is accessible across all streaming multiprocessors.
The device memory is interconnected with the host through a PCI-Express card (version
can vary depending upon the card). This interconnectivity enables DMA transfer of data
between host and device memory. From the origin of CUDA-based GPUs, a scratch-pad
memory, which is programmable and supports high-speed access, has been available pri-
ivate to a streaming multiprocessor. The scratch-pad memory is termed as shared memory
on NVIDIA cards. Till recently, the size of this shared memory was 64 KB.

**Coupled CPU-GPU:** Even though GPUs are a major component of the high performance
computing landscape today, computer architectures have already taken another step towards
more effective utilization of the GPUs, in the form of integrated chips. In these chips, a
multi-core CPU and a GPU are integrated in silicon. Both the two devices share the same
physical memory, which makes it possible to copy data between devices at high speeds.
The AMD Fusion architecture integrates a GPU that is OpenCL programmable on the CPU
chip, which has been available for some time. Intel released their latest generation *Ivy
Bridge* processor in late April, 2012; this chip also integrates a GPU that supports OpenCL.
NVIDIA has announced their on-going project Denver, which will integrate their ARM-based CPUs with the NVIDIA GPUs on the same chip. These newly emerged or upcoming heterogeneous architectures will efficiently accelerate computations on PCs or embedded systems.

We focus on the AMD Fusion chip, a representative integrated CPU-GPU chip we used to develop our framework.

The Fusion architecture integrates an AMD Opteron CPU and a GPU on the same silicon chip. The processing component of the on-chip GPU is similar to the discrete GPUs, such as the NVIDIA GPUs and the AMD Radeon GPUs. The processing component of the GPU is composed of a certain number of streaming multiprocessors. Each streaming multiprocessor, in turn, contains a set of simple cores that perform in-order processing of the instructions, and use SIMD parallelism. Furthermore, threads running on a streaming multiprocessor are divided into wavefronts, which are similar to warps in NVIDIA GPUs. Each wavefront of threads are co-scheduled on the streaming multiprocessor and execute the same instruction in a given lock-step.

The GPU does not have its own device memory. Instead, it has access to the system memory with the CPU. Unlike a GPU which is connected to the host through a PCIe bus, the data transfers between memory and both processing units are through the same high speed bus, and are controlled by a unified memory controller. The system memory is however divided into two parts: the device memory and the host memory. The device memory and host memory are designed for the GPU and the CPU, respectively, though both of them can be accessed by either device. Such accesses are possibly through a type of memory buffer called the zero copy buffer.
Starting with the CUDA-based (NVIDIA) GPUs, a scratch-pad memory, which is programmable and supports high-speed accesses, has been available private to each streaming multiprocessor. The scratch-pad memory is termed as the shared memory. On the Fusion architecture, the size of the shared memory is 32 KB.

In the OpenCL programming standard, the CPU and the GPU can be viewed to have the same logical architecture. Every processing core on the CPU is similar to one streaming multiprocessor on the GPU, and corresponds to one workgroup (thread block) in the OpenCL programming model. Each workgroup contains a number of threads, which has been discussed before. The same OpenCL code can run on both devices without modification.

Intel MIC Architecture: Over the last 6-7 years, high-end computing systems have changed significantly with respect to the intra-node architectures, with popularity of coprocessors. Over the last 3 years, as many as three of the five fastest supercomputers (at any time, based on the bi-annual top 500 list [1]) in the world involved coprocessors on each node, as they offered excellent performance-price and performance-power ratios. A recent development along the same lines has been the emergence of Xeon Phi chips, based on the Intel MIC architecture. Xeon Phi is a promising system, because it allows x86 compatible software to be used. However, there are many challenges in effectively using this hardware. Xeon Phi is built on top of the long-existing Intel SSE (Streaming SIMD Instructions), and particularly, supports IMCI (Initial Many Core Instructions) instruction set for use of SIMD. The SIMD width has been extended to 512 bits (16 floats), potentially offering large benefits for applications. Overall, effectively exploiting the power of a coprocessor like Xeon Phi requires that we exploit both MIMD and SIMD parallelism. While the former can be done through Pthreads or OpenMP, it is much harder to extract SIMD performance. This is because the restrictions on the model make hand-parallelization very
hard. At the same time, productions compilers are unable to exploit SIMD parallelism for many of the cases.

The x86-compatible Intel Xeon Phi coprocessor, which is a latest commercial release of the Intel Many Integrated Core (MIC) architecture, has already been incorporated in 9 of the top 100 supercomputers at the time of writing this paper [1]. MIC is designed to leverage existing x86 experience and benefit from traditional multi-core parallelization programming models, libraries, and tools.

In the available MIC systems, there are 60 or 61 x86 cores organized with shared memory. These cores are low frequency in-order ones, and each supports as many as 4 hardware threads. Additionally, there are 32 512-bit vector registers on each core for SIMD operations. The main memory sizes vary from 8 GB to 16 GB, and the memory is shared by all cores. The L1 cache is 32 KB, entirely local to each core, whereas each core has a coherent L2 cache, 512 KB, where cache for different cores are interconnected in a ring.

1.2 Communication Patterns

In this section, we introduce the communication patterns that we have studied and will continue to study. The communication patterns we will introduce include: MapReduce, Generalized Reductions, Irregular Reductions, Stencil Computations, and Graph Processing.

1.2.1 MapReduce

MapReduce [31] was proposed by Google for application development on data-centers with thousands of computing nodes. It can be viewed as a middleware system that enables easy development of applications that process vast amounts of data on large clusters. Through a simple interface of two functions, map and reduce, this model facilitates parallel
implementations of many real-world tasks, ranging from data processing for search engine support to machine learning [25, 41].

MapReduce expresses the computation as two user-defined functions: *map* and *reduce*. The *map* function takes a set of input instances and generates a set of corresponding intermediate output *(key, value)* pairs. The MapReduce library groups together all of the intermediate values associated with the same key and shuffles them to the *reduce* function. The *reduce* function, also written by the users, accepts a key and a set of values associated with that key. It merges together these values to form a possibly smaller set of values. Typically, just zero or one output value is produced per *reduce* invocation.

The main benefits of this model are in its simplicity and robustness. MapReduce allows programmers to write functional style code that is easily parallelized and scheduled in a cluster environment. One can view MapReduce as offering two important components [87]: a *practical programming model* that allows users to develop applications at a high level and an *efficient runtime system* that deals with the low-level details. Parallelization, concurrency control, resource management, fault tolerance, and other related issues are handled by the MapReduce runtime.

### 1.2.2 Generalized Reductions

This operation involves steps of generating key-value pairs and combining these pairs using associative and commutative operations. Such functions have been supported by our HPC programming systems, e.g., OpenMP [28] provides a reduction clause for combination of computation results from different threads, although the reduction is only applicable to individual scalar variables for most compilers.

### 1.2.3 Irregular Reductions
Real X(numNodes), Y(numEdges); ! Node&Edge Data arrays
Integer IA(numEdges, 2); ! Indirection array
Real RA(numNodes); ! Reduction array
for (i = 1; i < numEdges; i++) {
    RA(IA(i,1)) = RA(IA(i,1)) op (Y(i) op X(IA(i,1)) op X(IA(i,2)));
    RA(IA(i,2)) = RA(IA(i,2)) op (Y(i) op X(IA(i,1)) op X(IA(i,2)));
}

Irregular reductions can also be referred to as unstructured grids in Berkeley dwarfs. Unlike structured grids, nodes in unstructured grids are connected by edges explicitly, since the connectivity of nodes cannot be determined by node positions (coordinates). The code above shows a typical irregular reduction loop. Each loop updates two elements in reduction array RA using commutative and associative operations op, through an indirection array (edges) IA. Because each node may be connected by multiple edges, the update of the elements in RA involves reductions. The related data in an irregular reduction loop could be categorized as read only input data and reduction result. In the above example, the input data are arrays X, Y and IA, while the reduction result is RA. Our API for irregular reductions will be based on these elements.

1.2.4 Stencil Computations

Stencil applications involve computations of updating each element in the input based on the values of its neighboring elements and itself. The input for stencil applications is usually a structured grid or matrix with two or higher dimensions. Stencil applications are usually quite simple and easy to implement in serial code, however, optimizations to the parallel code are complicated for programmers, and accelerating it using a heterogeneous cluster is even harder.
1.2.5 Graph Processing

Graph algorithms have been widely used, and for a long time – for example, \textit{route problems} such as the \textit{shortest path problem} and others are important algorithms, used in some of the common and recent applications, such as route suggestions for navigation and online maps. \textit{Graph coloring problems} are used in resource scheduling or color arrangement for maps. With rapid development of social network websites, graphs have become the most important data structure for representing relationships among people or other entities for social network websites such as Facebook, Twitter, Wikipedia, and others. A variety of graph mining algorithms have been proposed for discovering the relationships among people for social network graphs.

The most commonly used graph representations for storing a graph are the \textit{adjacency matrices} or \textit{adjacency lists}. \textit{Compressed Sparse Row (CSR) format} is a well known format for efficient storage of sparse matrices, which is also widely used for storing sparse graphs. Figure 1.1 shows an example graph and its storage in CSR format (the associated vertex data or edge data are not shown).

Figure 1.1: An Example Graph and Its Representation in CSR Format
Despite a significant variation in the underlying logic across different graph algorithms, they share certain similarities. First, most graph algorithms are vertex oriented, i.e., the values and the processing is centered around vertices. These algorithms usually gradually update the values associated with vertices, using edges as the auxiliary information to perform the computations. The output of these algorithms is typically a set of vertex values. Second, edges are usually used for communication among vertices. These common characteristics have inspired the design of many graph processing programming models that have targeted different hardware platforms, including Pregel [75], GPS [89], and Medusa [123], and also form the basis for the work presented in this paper.

**Google Pregel:** There have been a number of studies on graph programming models for different hardware platforms. Many of these models are based on the Bulk Synchronous Parallel (BSP) model [110]. BSP involves three important steps in each phase, or what they refer to as a super-step, which are, concurrent local processing, communication, and, finally, synchronization. The use of this model has been proved to be successful to simplify parallel computing for many classes of applications, e.g., the MapReduce programming model [31] and its implementations on a variety of parallel platforms [45, 47, 73] can be viewed as following the BSP model.

The distributed graph processing framework, Pregel [75], is the first to apply BSP model for graph processing. Pregel is also the first to propose a parallel graph programming model with simplified APIs. Pregel extracts from graph applications the common properties and represents a graph processing application with message passing. The message passing typically involves processing edges for each vertex of the graph, passing values along edges to other vertices, and using received messages to update values associated with the vertex. A message is a data structure composing the destination vertex ID and the value
being sent. More concretely, Pregel exposes to users a user-defined \textit{vertex-oriented} function \texttt{compute()}, in which users can express sequential logic for processing associated with each vertex - including \textit{message processing}, \textit{vertex updating} and \textit{message generation}.

1.3 Dissertation Contributions

In this section, we give a brief introduction to the current contributions of our research work, including six components.

1.3.1 MapReduce for GPUs with Effective Shared Memory Usage

The MapReduce abstraction has been found to be suitable for specifying a number of applications that perform significant amount of computation (e.g. machine learning and data mining algorithms). These applications can be accelerated using GPUs or other similar heterogeneous computing devices. As a result, there have been several efforts on supporting MapReduce on a GPU \cite{11, 45, 47}.

A GPU is a complex architecture and often significant effort is needed in tuning the performance of a particular application or framework on this architecture. Effective utilization of \textit{shared memory}, a small programmable cache on each multi-processor on the GPU has been an important factor for performance for almost all applications. In comparison, there has only been a limited amount of work in tuning MapReduce implementations on a GPU to effectively utilize shared memory \cite{54}.

This work describes a new implementation of MapReduce for GPU, which is very effective in utilizing shared memory. The main idea is to perform a \textit{reduction-based} processing of a MapReduce application. In this approach, a key-value pair that is generated is immediately merged with the current copy of the \textit{output results}. For this purpose, a \textit{reduction object} is used. Since the memory requirements for the output results or the reduction object
is much smaller than the requirements for storing all key-value pairs for most applications, we reduce the runtime memory requirements very significantly. This, in turn, allows us to use the shared memory effectively. For many applications, the reduction object can be entirely stored in the shared memory.

Many challenges have been addressed to create a general and efficient implementation of our approach. A mechanism for dynamic memory allocation in the reduction object, maintaining a memory hierarchy for the reduction object, use of a multi-group strategy, and correct swap between shared and device memory for portions of the reduction object are some of the challenges we addressed.

1.3.2 MapReduce for a Coupled CPU-GPU Architecture

Based on the previous success of MapReduce framework on clusters as well as many-core and multi-core architectures, one can expect MapReduce to help application development and performance for the emerging fused CPU-GPU architectures as well. However, it is not clear as to how the MapReduce framework should exploit integrated CPU-GPU architectures.

This paper focuses on the design, implementation, and performance evaluation of a MapReduce framework on the integrated architectures, using an AMD Fusion chip as a representative example. We focus on the parallelism within a single chip, though the work can be extended to clusters of such nodes using the same mechanisms as the current cluster-based MapReduce implementations. We consider two different schemes for partitioning the work between CPU cores and GPU cores within a chip. The first scheme is the map-dividing scheme, which dynamically distributes the input to each scheduling unit on both devices, and each device conducts map and reduce simultaneously. The other scheme is
the pipelining scheme, which runs the map and reduce stages on different devices, i.e., each device only executes one stage of MapReduce. In our design, we use one processing core on the CPU to be the scheduler, and use each of the remaining CPU cores and each streaming multiprocessor on the GPU as one scheduling unit, in order to achieve flexible load balancing. With the goal of avoiding runtime data transfer between the CPU and the GPU, we put the input data and the scheduling information in the zero copy buffer. To support self-adaptive load balancing for the map-dividing scheme, our system uses a runtime tuning method to adjust the task block size for each scheduling unit. To lower the memory overhead, we use an alternative design of MapReduce based on continuous reduction, which pipelines the intermediate key-value pairs from the map stage to the reduce stage directly.

For the pipelining scheme, we try both dynamic load balancing and static load balancing based implementations.

### 1.3.3 Scheduling Applications on Architectures with Heterogeneous Cores

Based on our previous work, i.e., scheduling MapReduce on a coupled CPU-GPU, we propose several general scheduling methods for applications that are executed on heterogeneous chips, including both discrete CPU-GPUs and coupled CPU-GPUs. By examining the features of CPU-GPU architecture memory system, we observe that we need to address scheduling in the present of a non-coherent, non-uniform access shared memory. Traditional methods used in shared memory systems, such as work stealing, normally require locking support. We have developed several locking-free scheduling methods for our scenario, including: 1) a master-worker method, which dedicates a CPU core as the master for scheduling, and the remaining CPU cores and SMs on the GPU(s) are the workers, 2)
A core-level token passing model, which uses a token to pass the permission for retrieving tasks among all the cores/SMs, and 3) A device-level token passing model, which is a coarse-grained token passing scheme that only passes tokens between the CPU and the GPU(s), and thus reduces the token passing overhead. We have also implemented these methods on two different architectures, i.e., a decoupled CPU-GPU node with 2 GPUs, and a fused CPU-GPU node (from AMD). Thus, we also offer a portable runtime system to applications.

1.3.4 Pattern Specification Framework for Accelerating Scientific Computations on Heterogeneous Clusters

The heterogeneity in current systems is creating a major obstacle in programmability and portability of high-performance applications. Particularly, programming clusters of heterogeneous nodes is very challenging, and portability across different existing and emerging heterogeneous architectures is also a very difficult problem.

Though several ongoing efforts are addressing this problem, improving programmer productivity for scientific computations continues to be an ongoing challenge. Broadly, creating a high-level, general, and efficient parallel programming system remains the ‘holy grail’ of parallel computing. As compared to the various more general parallel programming efforts that are now considering CPU-GPU clusters [4, 9, 27, 43, 60, 109], we focus on a different problem, which is of creating a high-level and efficient framework that can cover a reasonable variety of, but not all, scientific applications.

In addressing this problem, our inspiration arises from MapReduce [31], initiated by Google, which is an example of a high-level model that has been extremely popular for its target class of applications. One of the remarkable aspects of MapReduce is that it has been applied on a variety of platforms, including a single GPU [47, 16] and even clusters.
of GPUs [97]. However, MapReduce is certainly not suitable for scientific applications that have complex communication patterns. Moreover, its popular open-source implementation, Hadoop, has been shown to be significantly slower than popular parallel programming models [55]. Yet, the popularity of MapReduce certainly leads to the question “can MapReduce-like APIs be developed for several classes of popular scientific applications, to ease application development, while achieving high performance on clusters with accelerators?”.

This work addresses this question. We start by individually considering popular patterns that arise in scientific computations. For our current work, we have focused on generalized reductions, irregular reductions, and stencil computations, which can cover scientific applications involving structured and unstructured grids and particle simulations. Together, these three patterns are sufficient to cover a reasonable subset of scientific applications (e.g. 16 out of 23 Rodinia [14] benchmark applications), and moreover, our current system can be extended to cover more patterns in the future. We create variants of the popular map and reduce functions in MapReduce for computations associated with these patterns, and implement them efficiently.

### 1.3.5 Graph Processing over CPU and the Intel Xeon Phi

This work describes a system for graph processing over the CPU and the Intel Xeon Phi architecture. We support a high-level API that is a variant of Pregel [75]. Pregel, which can be viewed as a specialization of MapReduce for graph processing, also originated from Google, just like MapReduce. Pregel-like systems have also been supported on distributed memory systems [75, 89] and graph processing units [123].
This work describes a system for graph processing utilizing both the CPU and the MIC chip. We support a vertex-centric high-level API that can express a graph application easily. The system enables the users to write portable code that is simultaneously executed on both a Xeon Phi and a multi-core CPU. Overall, we address the following four challenges in our work:

1) Minimizing random memory accesses and exploiting the wide SIMD lanes: this is achieved through an innovative design of the message buffer.
2) Reducing contention overhead from concurrent vertex update: we create a pipelining implementation for the message generation step, which is suitable for operating with a large number of threads.
3) Load balancing among the large number of cores: we support a novel dynamic load balancing scheme for execution within a device.
4) Workload partitioning between CPU and MIC: we develop an effective hybrid graph partitioning scheme which achieves balanced workload and minimized communication. Even though the first three challenges arise from the MIC execution, and the optimizations are specifically designed considering its particular properties, the same code and optimizations are used for CPU execution.

1.3.6 Exploiting SIMD Architectures for Irregular Applications

SIMD parallelism has been commonly available in popular processors for at least 15 years now. However, till recently, this type of parallelism has considered suitable only for regular applications. Recent SIMD architectural trends are changing this, and are creating new opportunities as well as challenges for irregular applications. The developments include wider SIMD lanes (up to 512 bits), combination of SIMD parallelism with massive shared memory MIMD parallelism (e.g. in Intel Xeon Phi, which has 61 cores), and more flexible programming APIs, which include gather and scatter instructions. These trends
have the following implications. First, with increasing SIMD width, there is an increas-
ing performance penalty associated with not exploiting such parallelism – for applications
that use integers and/or single precision floating point data, a potential speedup of up to 16
will not be achieved if SIMD potential is ignored. Second, effectively combining SIMD
parallelism with massive MIMD parallelism is important, which requires that issues like
limits on memory bandwidth and write conflicts must be addressed. Finally, with gather
and scatter operations, applications with non-contiguous accesses can be mapped to SIMD
hardware, but performance characteristics of the hardware must be understood and exploit-
ed.

This work investigates mapping of applications involving data-dependent memory ac-
cesses to modern SIMD features. Though different class of applications that fit this pattern
(i.e, unstructured grid computations, molecular dynamics, graph applications and sparse
solvers) have different characteristics, we develop a general optimization methodology ap-
licable to each of them. We observe how different types of irregular applications can be
viewed as sparse matrix computations – a representation that captures both the data access
and the computation patterns, and makes the optimization opportunities apparent. The op-
timization method we propose involves three important sub-steps: 1) locality enhancement
through tiling, 2) data access pattern identification, and 3) write conflict removal at both
SIMD and MIMD levels. This optimization method significantly improves data access lo-
cality and avoids inter-lane write conflicts, while eliminating the need for locking across
cores.

We show the generality and effectiveness of our optimization methodology with three
important irregular application subsets: irregular reductions, graph algorithms and sparse
matrix matrix multiplication (SpMM). These three subsets cover a broad range of cases,
including the cases that access only one sparse matrix, as well as the case that accesses multiple sparse matrices. The data access pattern of each subset is also distinct. Despite the differences, our optimization methodology effectively accelerates them.

1.4 Outline

Rest of this dissertation is organized as follows: Chapter 2 introduces a MapReduce framework for GPUs with effective shared memory usage. Chapter 3 presents a MapReduce system for a coupled CPU-GPU. Chapter 4 introduces several locking-free scheduling methods for applications running on parallel architectures with heterogeneous cores. Chapter 5 introduces a MapReduce like framework for programming applications with different communication patterns on heterogeneous clusters. Chapter 6 presents a graph processing system over the CPU and Intel Xeon Phi. Chapter 7 describes a general method for optimizing irregular applications on the Intel Xeon Phi. Chapter 8 compares our work with related work from other researchers, and Chapter 9 concludes this dissertation.
Chapter 2: Optimizing MapReduce for GPUs with Effective Shared Memory Usage

MapReduce was proposed with the increased need of data intensive processing in the big data era. The processing structure of MapReduce is very suitable for parallel platforms with very large scale parallelism, such as clusters with multi-cores on each node. The GPU architecture, i.e., a processing unit with a large amount of sequential processing cores supporting thousands of threads, is a very feasible hardware platform for running MapReduce. However, a GPU is a very complex architecture with a large number of cores and limited memory bandwidth in terms of the large number of threads running on it. Though there are several prior efforts on implementing MapReduce for GPUs, the performance of these systems are usually limited by the I/O overhead, especially for the reduction-intensive applications. In this work, we optimize MapReduce framework for GPUs with consideration of the memory hierarchy on modern GPUs. We intensively utilize the shared memory at the top of this memory hierarchy, and significantly improve the performance of the execution, especially for reduction intensive applications. The following of this chapter introduces how we designed our system.
2.1 System Design

This section describes the implementation of our system. The main emphasis in our approach is using the shared memory of a GPU effectively.

Initially, we describe why it is both important and challenging to effectively utilize shared memory while executing a MapReduce application. As we stated previously, shared memory supports much faster read and write operations. In addition, execution of an application with the MapReduce not only needs read/write operations, but also a large number of atomic operations, which are used to synchronize data accesses and updates by different threads. There is an even greater difference between the performance of atomic operations on shared memory and their performance on device memory. Thus, a large number of atomic operations on device memory can greatly undermine the performance of a MapReduce application.

The key challenge, however, in effectively utilizing shared memory is that the capacity of shared memory is very limited. Most MapReduce applications generate a large number of intermediate key-value pairs at the end of the Map phase. For these applications, it is impossible to keep all the key-value pairs in shared memory.

A recent implementation of MapReduce has demonstrated one mechanism for using shared memory. Particularly, the work from Ji et al. involves an innovative approach to using shared memory as a buffer to stage the input and output of both the map and reduce stages [54]. They copy the input from the device memory in a coalesced pattern into the shared memory. During the processing from a particular stage, when the shared memory is full, they copy the output from shared memory to device memory, again using coalesced writes. In this way, they achieve a speedup over the implementation that uses
device memory only. However, the key-value pairs from the map stage still need to be stored in device memory, and shuffling is also required.

In our framework, we use a distinct approach for exploiting shared memory. We implement a MapReduce framework in a reduction-based manner, which is an alternative to the traditional implementation methods. Our focus is on exploiting shared memory effectively for reduction-intensive applications. By reduction-intensive application, we imply an application that generates a large number of key-value pairs that share the same key. Thus, there is a significant amount of work performed during the reduce phase. Moreover, the reduction operation is associative and commutative, which gives us flexibility in executing the reduction function.

Before describing our system in details, we first discuss the idea of the reduction-based MapReduce.

2.1.1 Basic Idea: Reduction Based Implementation

In reduction-based MapReduce, the map function, which is defined by the users, works in the same way as in the traditional MapReduce. It takes one or more input units and generates one or more key-value pairs. However, the key-value pairs are handled in a different way. In the traditional MapReduce, the key-value pairs are first stored, and after all the input units have been processed, the MapReduce library groups together all of the intermediate values associated with the same key and passes them to the reduce function. The reduce function each time accepts a key and a set of values associated with that key. It merges these values in the way defined by the users. In some cases, a user may define a combine function, which can combine key-value pairs generated on the same node, and
reduce interprocess communication. However, the memory requirement on each node is typically not reduced by the combine function.

The reduction-based method, which can only be used if the reduction function is associative and commutative, inserts each key-value pair to the reduction object at the end of each map operation. Thus, every key-value pair is merged to the output results immediately after it is generated. A data structure storing these intermediate values of output is referred to as the reduction object. Every time a key-value pair arrives, the reduction object locates the index corresponding to the key, and reduces this key-value pair to that index in the reduction object, exploiting the associative and commutative property. The reduction object is transparent to the users, who write the same map and reduce functions as in the original specification.

We use k-means clustering as a running example to illustrate our concepts. K-means clustering is one of the most popular data mining algorithms [53]. The clustering problem is as follows. We consider data instances as representing points in a high-dimensional space. Proximity within this space is used as the criterion for classifying the points into clusters. Four steps in the sequential version of k-means clustering algorithm are as follows: 1) start with \( k \) given centers for clusters; 2) scan the data instances, for each data instance (point), find the center closest to it and assign this point to the corresponding cluster; 3) determine the \( k \) centroids from the points assigned to the corresponding centers, and 4) repeat this process until the assignment of points to clusters does not change.

Algorithm 1 and Algorithm 2 show the code of map and reduce functions in k-means. The map function processes one input point each time. It uses the identifier of the closest cluster center as the key, and the coordinates of this point, together with the number of points (1) and the distance to the cluster center are combined together to form a key-value
Algorithm 1: map(input, offset)

\[
point \leftarrow \text{get\_point}(input, offset);
\]
\[
\text{for } i \leftarrow 0 \text{ to } K \text{ do}
\]
\[
\begin{aligned}
\text{dis} & \leftarrow \text{distance}(point, \text{clusters}[i]); \\
\text{if } \text{dis} < \text{min} \text{ then} \\
\text{min} & \leftarrow \text{dis}; \\
\text{min\_idx} & \leftarrow i;
\end{aligned}
\]

\[
K\text{means\_value value};
\]
\[
\text{value.num\_points} \leftarrow 1;
\]
\[
\text{value.dim}0 \leftarrow \text{point}[0];
\]
\[
\text{value.dim}1 \leftarrow \text{point}[1];
\]
\[
\text{value.dim}2 \leftarrow \text{point}[2];
\]
\[
\text{value.dist} \leftarrow \text{min};
\]
\[
\text{reduction\_object.insert}(&\text{min\_idx}, \text{sizeof}(&\text{min\_idx}), &\text{value}, \text{sizeof}(&\text{value}));
\]

Algorithm 2: reduce(value1, value1\_size, value2, value2\_size)

\[
\text{km\_value1} \leftarrow (K\text{means\_value})\text{value1};
\]
\[
\text{km\_value2} \leftarrow (K\text{means\_value})\text{value2};
\]
\[
\text{K\text{means\_value} tmp};
\]
\[
\text{tmp.num\_points} \leftarrow \text{km\_value1.num\_points} + \text{km\_value2.num\_points};
\]
\[
\text{tmp.dim}0 \leftarrow \text{km\_value1.dim}0 + \text{km\_value2.dim}0;
\]
\[
\text{tmp.dim}1 \leftarrow \text{km\_value1.dim}1 + \text{km\_value2.dim}1;
\]
\[
\text{tmp.dim}2 \leftarrow \text{km\_value1.dim}2 + \text{km\_value2.dim}2;
\]
\[
\text{tmp.dist} \leftarrow \text{km\_value1.dist} + \text{km\_value2.dist};
\]
\[
\text{copy}(value1, &\text{tmp}, \text{sizeof}(&\text{tmp}));
\]

pair. As part of runtime processing, this key-value pair is inserted in the reduction object. A reduction object is a data structure that maintains the current value of the final output results from the reduce stage of the application. For k-means clustering, the output of the reduce stage is the number of points associated with each cluster center, together with the total aggregated coordinates and distance of these points from the cluster center. At any point in the processing of the application, the reduction object contains the same information, but only for the set of points that have been processed so far.

The insert function works as follows. If the key already exists in the reduction object, the reduce function is invoked to merge the new key-value pair to the existing key. If it is the first time that this key is inserted, a new space for this key-value pair is created. Thus,
elements within the reduction object are dynamically allocated. The reduce function in this example computes the new value for the existing key, i.e., it accumulates the number of points, the coordinates, and the distance of the new value to the existing value associated with that key. We will discuss the insert function in details in Section 2.1.4.

As we stated earlier, the main advantage of this approach is that the memory overhead of a large number of key-value pairs is avoided. This, in turn, is likely to allow us to utilize the shared memory on a GPU. However, we still need to address several challenges to have an efficient implementation for MapReduce processing on a GPU.

To complete our design, we have the following features, which will be explained in the rest of this section.

• A memory hierarchy for maintaining the reduction object.

• A multi-group scheme in shared memory to trade off space requirements and locking overheads.

• A general and efficient data structure for the reduction object.

• An efficient overflow handling and swapping mechanism.

2.1.2 Memory Hierarchy

The processing structure we support reduces the memory requirements, by replacing the need for managing key-value pairs by a reduction object. For many applications, the reduction object tends to be small and can be kept in shared memory throughout the computation. However, for some other applications, it is possible that the number of distinct keys is very large, and shared memory may not be sufficient to hold the reduction object.
To effectively use shared memory and yet have a general implementation, a memory hierarchy is maintained in our framework. As shown in Figure 2.1, reduction objects exist in both shared memory and device memory.

Elements of the reduction object are kept in shared memory as long as there is space. If an element corresponding to a key value that did not exist in the reduction is to be added, and there is no space, our framework swaps the data out to the device memory reduction object. The swapping mechanism will be discussed in details in Section 2.1.5.

There is only one copy of the reduction object in the device memory, which is used to collect the reduction result from shared memory reduction objects. The number of shared memory reduction objects depends upon the number of thread blocks and a design decision related to the number of thread groups that are created within each block. Each thread block is divided into one or more groups, and each group has one copy of the reduction object.
object. Further details of the reasons for creating these thread groups will be discussed in the next subsection.

After all the threads in one thread block finish processing, they merge the data from the shared memory reduction object(s) into the device memory reduction object. To improve efficiency, this step is also performed in parallel. Merging threads read data as key-value pairs from shared memory reduction object(s), and insert them into the device memory reduction object. After all threads on GPU finish processing, they cooperate to copy data from the device memory reduction object to the result array, which is finally copied to host memory at last.

2.1.3 Multi-group Scheme in Shared Memory

A large number of threads (typically up to 512) are used in each thread block to exploit available parallelism in a streaming multiprocessor. Suppose there is one copy of the reduction object for all threads in the block. If all threads try to update the object, race conditions can arise. To avoid race conditions, we use synchronization mechanisms, such as the atomic operations that are supported on GPUs. However, when the number of distinct keys is small, the contention between the threads for updating the values associated with one key can be large, leading to significant waiting times.

One method for avoiding these costs could be to use full replication, i.e., create one copy of the reduction object for each thread. However, this method is not feasible because the number of threads is large, and if every thread owns one copy of the reduction object, the memory space needed will be large, leading to poor utilization of shared memory. In addition, such a large number of copies of the shared memory reduction object can lead to a very high overhead when they are merged to the device memory reduction object.
In order to address this problem, we introduce a *multi-group scheme*. Here, each thread block is partitioned into multiple groups, and each group owns its own copy of the reduction object. Let the number of threads in one block be $N$, and the number of groups in one block be $M$. Then the number of threads in one group is $N/M$ and these threads share one copy of the reduction object. Locking operations are still required to update the copy of the reduction object, but the contention among the threads is reduced, since the number of threads competing for the same copy of the reduction object is now smaller. At the same time, the number of copies of the reduction object is still reasonable, which keeps the memory requirements and the overhead of combination modest.

The number of groups of threads that are used is a parameter in the system, and its optimal value can depend upon the application. Up to a certain level, increasing the number of thread groups improves performance, since the contention among threads is reduced. However, after a certain point, the performance can be reduced. One reason is that the size of shared memory is limited, so if the total number of keys is large, then the shared memory may not be able to hold many copies of the reduction object. This, in turn, increases the frequency with which swap mechanism needs to be invoked. Also, when the computation finishes, reduction objects in shared memory need to be merged into the device memory reduction object, and the data copying and synchronization to device memory can be time consuming. Clearly, the optimal choice can depend upon the size of one copy of the reduction object. When the reduction object is small and/or involves a smaller number of distinct keys, larger number of groups (or a smaller number of threads in each group) will be preferable. Similarly, when the reduction object is large and/or involves a large number of distinct keys, a smaller number of copies of the reduction object should be used.
An important parameter in our approach is the number of groups, $g$. The optimal choice for this parameter can be determined as follows. The overall execution time $T$ for a specific application can be represented as:

$$T = C + \frac{T_{con}}{g} + T_{com} \times g$$  \hspace{1cm} (2.1)$$

where, $C$ is the total time spent on initialization and computation, which is a constant for a given application and dataset. $T_{con}$ is the contention overhead if only one group is used, and $T_{com}$ is the combination overhead for one reduction object.

The minimum value of $T$ is achieved when $g$ equals $\sqrt{\frac{T_{con}}{T_{com}}}$, and the resulting optimal value of $T$ is $C + 2\sqrt{T_{con} \times T_{com}}$.

### 2.1.4 Reduction Object Implementation

The structure of the reduction object is shown in Figure 2.2. It consists of two main parts: an index array and a memory pool. The index array is implemented as a hash table. Each bucket in the index array is composed of two indices: a key index and a value index.
The key index refers to the position where the key data is stored in the memory pool. The value index is the position where the value data is stored.

Every bucket corresponds to a unique key, and the bucket is initialized when the first key-value pair with that particular key is inserted. The key data area contains not only the key data, but also the size information of key and value. In order to save space, only two bytes are used for storing the size information: one byte for the key size and the other for the value size. The memory pool is also an array, which is allocated before the computation starts. A memory allocator is used to allocate space for new keys and values.

Besides the index array and memory pool, one lock array is used in every copy of the reduction object. For each bucket in the index array, one lock is used to synchronize the updates to the key and value associated with that bucket.

Updating the Reduction Object: Algorithm 3 shows the logic of updating a reduction object. When a key-value pair is to be inserted into the reduction object, the hash value of the key is first calculated by the hash function, and then an index is calculated by using the hash value. The framework uses this index to find the bucket corresponding to the key.

The update involves the following two steps:

1. The thread which is performing the computation acquires the corresponding lock on the bucket. If the bucket is empty, new space for the key data and the value data is allocated by the memory allocator. After that, the size information and the key data is stored into the newly allocated key data space, and the value data is stored into the value data space. Finally, the index information of the key and the value is stored in the bucket. The computing thread releases the lock.

2. If the bucket already contains data, the key size and key address associated with this bucket are first retrieved, and then used to compare with the new key. If the two
Algorithm 3: insert(key, key_size, val, val_size)

index ← hash(key, key_size) % NUM_BUCKETS;
while finish ≠ 1 do
    DoWork ← 1;
    /*wait on the lock*/
    while DoWork do
        if get_lock(index) then
            if buckets[index] = 0 then
                k ← alloc(2 + key_size);
                v ← alloc(val_size);
                /*store size information*/
                key_size_addr ← get_addr(k);
                val_size_addr ← key_size_addr + 1;
                *key_size_addr ← key_size;
                *val_size_addr ← val_size;
                /*store key and value data*/
                key_data_addr ← key_size_addr + 2;
                val_data_addr ← get_addr(v);
                copy(key_data_addr, key, key_size);
                copy(val_data_addr, val, val_size);
                /*store key and value indices*/
                buckets[index][0] ← k;
                buckets[index][1] ← v;
                release_lock(index);
                DoWork ← 0;
                finish ← 1;
            else
                key_r ← get_key_addr(index);
                key_size_r ← get_key_size(index);
                if equal(key_r, key_size_r, key, key_size) then
                    val_r ← get_val_addr(index);
                    val_size_r ← get_val_size(index);
                    reduce(val_r, val_size_r, val, val_size);
                    release_lock(index);
                    DoWork ← 0;
                    finish ← 1;
                else
                    release_lock(index);
                    DoWork ← 0;
                    index ← new_index(index);
            endif
        endif
    endwhile
endif
endwhile

keys are the same, the user-defined reduce operation is performed, and the value associated with the key is updated. After the reduce operation, the computing thread releases the lock on the current bucket. If the keys are not the same, the computing thread releases the lock and calculates a new index, which is then used to repeat the steps starting from the Step 1.
Memory Allocation in Reduction Object: Beginning with CUDA 3.2, dynamic device memory allocation has been supported on NVIDIA GPUs. However, the default mechanism tends to be expensive. Hong et al. developed a light-weight dynamic device memory allocator as part of the MapCG implementation [47]. Our method for memory allocation works in a similar way as MapCG, but has its own characteristics.

In MapCG, a large memory pool in the device memory is reserved before the computation starts. Every time additional space is needed, the light-weight memory allocator allocates a memory space and returns the memory index. We associate a memory allocator with both the device memory reduction object and the shared memory reduction object. These two memory allocators are somewhat different from each other. In every shared memory reduction object, only one offset is used to indicate the next available memory space. However, in the device memory reduction object, multiple offsets are used. We divide the threads in each block into many groups, which is similar to the multi-group scheme in the shared memory. The reason why we use multiple offsets for the device memory reduction object is that all the threads across all the blocks can be updating the same device memory reduction object. To avoid race conditions, the memory offset is updated by using atomic operations. However, the contention of the lock can be high with a single copy, which is why separate offsets are used. Also, we store the offsets in the shared memory, which further lowers the overhead of memory allocation.

2.1.5 Handling Shared Memory Overflow

For many MapReduce applications, it is possible that during the process of computation, the shared memory reduction object gets full. More specifically, there can be two situations in which a reduction object is full. The first is when the buckets in the index
array are all used up, and the second situation is when the memory pool does not have any more available space to allocate.

In such cases, we need to perform overflow handling. Our framework supports two ways of overflow handling. The first way is to swap the data out to device memory, and the second is to sort the reduction object and delete unnecessary data. For most applications, the swapping mechanism is used when the reduction object is full. Data from the shared
memory reduction object is retrieved and inserted into the device memory reduction object. After such a swap is completed, the shared memory reduction object in one block is re-initialized, which is necessary for the remaining computations.

The reduction object also supports *in-object parallel sorting*. For applications like kNN (k-nearest neighbor search), when the reduction object in shared memory is full, bitonic sort [42] is performed on the reduction object. In such application, newly inserted data can make earlier data redundant. After sorting, only a small part of the data is kept in the reduction object, and the unnecessary data is deleted, which makes room for the remaining computation.

Algorithm 4 shows the logic of the swapping mechanism in overflow handling. In-object sorting logic is similar and is not shown here. CUDA provides a synchronization barrier routine for the threads within each block. A full flag shared among one block is used to indicate whether overflow handling is needed. Every time before a thread performs the *map* operation, it checks this flag. If this flag is not set, it carries out the regular update. Whenever a thread encounters an insertion failure, it sets this flag. All other threads within this block will see that the flag has been set, and know that they need to cooperate to conduct overflow handling. All threads in the block stop computations and merge the shared memory reduction object into the device memory reduction object. This operation is performed in parallel by all threads. If multiple thread groups, i.e., multiple copies of the reduction object are being used, the following logic is used. Irrespective of which group’s reduction object is full, all threads from all groups within the block need to suspend all computations and perform overflow handling. Every thread only participates in the work associated with its own group’s reduction object. We require all groups to perform overflow handling even when only one group’s reduction object is full because the synchronization
barrier, \_syncthreads(), is used for all the threads in one block. There is no synchronization mechanism for a subset of threads in each block, except for the automatic synchronization within a warp. Although other groups’ reduction objects may not yet be completely full, the rate of space utilization within the reduction objects is almost the same, because the load has been evenly distributed.

2.1.6 Application Examples

In this section, we use two representative examples to show how our framework supports different parallel applications. These two applications are k-means clustering (k-means) and k-nearest neighbor search (kNN). We use k-means to demonstrate the use of swapping mechanism, and use kNN to show how the in-object sorting mechanism helps.

**K-means Clustering:** The steps in the k-means clustering algorithm, along with a MapReduce implementation, were described earlier in Section 2.1.1. Now we show how our framework implements this algorithm.

As the implementation of the map function shows, we use the cluster center identifier as the key, and a structure containing the coordinates, the number of points and the distance from the cluster center to the current point as the value. Thus, at most \( k \) distinct keys are generated.

Immediately after a key-value pair is generated, it is inserted into the reduction object. As described in Section 2.1.4, when a key-value pair is inserted, the reduce function is used to update the newly generated value to the existing value corresponding to the same key. If \( k \) is small, for example, when \( k \) is 10 or 20, then the number of distinct keys is also small. We can use more groups in each block to reduce the contention among threads. If \( k \) is larger, such as 50 or 100, then we should use fewer groups to ensure that the shared
memory can hold all the copies of the reduction object. For even larger \( k \), we can choose the number of groups to be 1, though even in this case, shared memory may overflow and the swapping mechanism will be required.

**K-nearest Neighbor Search:** The k-nearest neighbor search (kNN) is a type of instance-based learning method [3]. This method is used to classify an object in an n-dimensional space based on the closest training samples. Given \( n \) training samples and one unknown sample, the k-nearest neighbor classifier searches the training samples by euclidean distance and finds the \( k \) nearest points to the unknown sample. Then it classifies the unknown sample according to the majority vote of the \( k \) nearest samples.

When implementing this algorithm by using our framework, we can take advantage of the in-object sorting mechanism we have supported, though it requires an extra function to be provided by the users. For every training sample, we use its point identifier as the key, and the distance from the unknown sample as the value. The framework keeps generating key-value pairs. When the reduction object corresponding to one group is full, the framework does not swap the reduction objects in the shared memory, but instead it conducts bitonic sort on the reduction objects. Users only need to define a single additional function, which will be comparing two buckets. This function is invoked by the sorting algorithm. After sorting, the framework deletes unnecessary data and only keeps \( k \) points that are the closest to the unknown sample. These points are kept in the shared memory reduction object, whereas the other space is made available for the remaining computations. When all threads finish processing data points and have combined the shared memory reduction objects to the device memory reduction object, an in-object sort on the device memory reduction object is performed.
2.1.7 Discussion

As we mentioned before, our framework is very suitable for reduction-intensive applications. These applications generate a large number of key-value pairs for any given key-value, and the key-value pairs are reduced in an associative and commutative way. For these applications, we can reduce the memory overhead, and effectively utilize the shared memory.

Consider an application with no reduction, i.e. where only one key-value pair is generated for a given key, or the reduction operation is non-associative-and-commutative. Matrix Multiplication, if written using a MapReduce framework, has this property. In such cases, our method has no advantage over other GPU based MapReduce frameworks. In fact, it may be better not to try and use shared memory, since what is written in shared memory is never updated, and then has to be copied to device memory.

While using our framework, it is desirable to investigate the specific properties of the application. Our implementation is advantageous (over other GPU map-reduce implementations) only for applications with associative and commutative reduction operations, and within those, applications that are reduction-intensive. Also, certain parameters in our system, including the number of buckets, the memory pool size in each reduction object, as well as the number of groups in each thread block, currently need to be determined by the users. The users can make these decisions based on the potential number of key-value pairs that can be generated and the estimated memory requirements of the key-value pairs. This is similar to the more popular map-reduce implementations like Hadoop, where parameters like chunk size, replication factors, and others need to be explicitly chosen by users. The next step in our research will be developing techniques to automatically choose the values of some or all of these parameters.
The architecture of modern GPUs are changing rapidly. Over time, GPUs have seen an increase in shared memory size and device memory access speed. However, memory hierarchy still exists, and will continue to exist. Our method of utilizing shared memory for MapReduce on a GPU is, therefore, applicable to any of the existing or upcoming architectures. In fact, because larger shared memory can hold larger reduction objects, and thus reduce the frequency of swapping operations, our approach may be even more beneficial.

2.1.8 Extension for Multi-GPU Systems

Even though a single GPU is a highly parallel architecture, high throughput processing may require use of a multi-GPU node, and/or a cluster with GPU(s) on each node. We now argue why our reduction object based approach is suitable for multi-GPU environments.

For applications that involve associative and commutative reduction operations and that are reduction-intensive, the following approach can be used. First, the data to be processed can be evenly divided between the nodes or GPUs. Second, processing as described in this paper can be directly applied on each GPU. The result of this processing will be the reduction object. Next, the reduction objects from each node or GPU can be combined using the associative and commutative reduction function to obtain the final results. The approach here will have two advantages. First, the processing on every GPU will be faster than other approaches. Second, the communication overheads will be lower than other approaches that involve shuffling of key-value pairs. If the reduction object is small, the final global reduction step will be very inexpensive.
For applications that do not involve associative and commutative reduction operation or are not reduction-intensive, we can default to the approaches used by other multi-GPU implementations, for example, GPMR [97].

2.2 Experimental Results

We have evaluated the performance of our framework with many different applications, covering various types of applications that can be implemented using the MapReduce framework. We compare the performance of the applications against sequential (1 CPU thread) implementations, which were chosen or implemented to have the same logic/algorithm as the corresponding MapReduce implementations. We also compare the performance against an earlier MapReduce framework for GPU, MapCG [47], which does not use shared memory intensively. Finally, we also compare the performance of our system against Ji et al.’s work [54], a recent implementation of MapReduce that does use shared memory.

We have also conducted a number of experiments to evaluate some of the design decisions we made. For the reduction-intensive applications we have used, we consider different configurations of thread groups in each block and test the impact of the multi-group scheme. To investigate the performance of the swapping mechanism, we evaluate two applications with specific datasets.

2.2.1 Experimental Setup and Applications

Our experiments were conducted using an NVIDIA Quadro FX 5800 GPU with 30 multiprocessors. Each multiprocessor has 8 processor cores working at a clock rate of 1.3 GHz. Thus the GPU we used has 240 processor cores in total. The total device memory size is 4 GB and the shared memory size on each multiprocessor is 16 KB. The GPU is
connected to a machine with 8 AMD 2.6 GHz dual-core Opteron CPUs. The total main memory on this machine is 24 GB.

We selected seven commonly used MapReduce applications to evaluate the performance of our framework. These examples cover both reduction-intensive and map computation-intensive applications. As we stated earlier, the former represents applications which have a large number of key-value pairs for each unique key, and the reduction operation is associative and commutative, and thus, reduction computation time is significant. In contrast, the latter represents applications that spend almost all their time in map stage, i.e., they have one or a very small number of key-value pairs for each unique key.

The applications we selected were as follows. K-means clustering (k-means) is one of the most popular data mining algorithms, which has been described previously. An important parameter here is the number of cluster (centers), \( k \), which impacts memory requirements, and thus the multi-group scheme. K-nearest neighbor classifier (kNN) is a simple but popular machine learning algorithm, which was also described earlier.

Word Count (WC) is a very commonly used application for evaluating MapReduce frameworks. It calculates the total number of occurrences for each distinct word. In the key-value pairs generated by this application, the key is the character sequence of a word, and the value is the integer 1. WC can be reduction-intensive if the number of distinct words is small, and then the reduction objects can be kept in shared memory until the end of the computation. If the number of distinct words is very large, the reduction object cannot be held in shared memory, and then the swapping mechanism is needed.

Naive Bayes Classifier (NBC) is a simple classification algorithm based on observed probabilities. Given two sets of observations, one with classifications, one without, the
algorithm classifies the second set. NBC selects the most likely classification $V_{nb}$ given the attribute values $a_1, a_2, \ldots a_n$. $V_{nb}$ is computed as

$$V_{nb} = \text{argmax}_{v_j \in V} P(v_j) \prod P(a_i|v_j)$$

(2.2)

The particular invocation of this application in our experiments is given a set of data which includes the attributes color, type, origin, transmission type, and whether or not stolen. It judges the age of this vehicle based on the given information. When implemented using MapReduce, the total number of distinct keys is 30. The value generated by each map function is 1.

Our next application, Page View Count (PVC), obtains the number of distinct page views from web logs. Each entry in the web log is represented as $\langle \text{URL, IP} \rangle$, where URL is the URL of the accessed page and IP is the IP address that accesses the page. This application involves two passes in MapReduce. The first one removes the duplicate entries in the web logs, and the second one counts the number of views for each URL. In the first pass, the pair of the entry is used as the key and the size of the entry is used as the value. The second pass uses the URL as the key, and integer 1 as the value. Typically, the datasets have a lot of repeated entries, so the application is reduction-intensive in nature.

Another application we used is Matrix Multiplication (MM), which is a map computation-intensive application. Given two input matrices $A$ and $B$, the map stage computes multiplication for a row $i$ from $A$ and a column $j$ from $B$. It outputs the pair $\langle \{i, j\} \rangle$ as the key and the corresponding result as the value. If $A$ is an $m_a \times n_a$ matrix, and $B$ is an $m_b \times n_b$ matrix, then the total number of distinct keys is $m_a \times n_b$, which is a very large number if the dimensions of the matrices are large. Thus for this application, we use the
Table 2.1: Applications and Datasets Used in our Experiments

<table>
<thead>
<tr>
<th>Application</th>
<th>Dataset Size (Small, Medium, Large)</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-means, K = 20 (KM20)</td>
<td>10K points, 100K points, 1M points</td>
</tr>
<tr>
<td>K-means, K = 40 (KM40)</td>
<td>10K points, 100K points, 1M points</td>
</tr>
<tr>
<td>K-means, K = 100 (KM100)</td>
<td>10K points, 100K points, 1M points</td>
</tr>
<tr>
<td>KNN, K = 20 (KNN)</td>
<td>10K points, 100K points, 1M points</td>
</tr>
<tr>
<td>Word Count, 90 Distinct Words (WC)</td>
<td>5.46MB, 10.94MB, 87.5MB</td>
</tr>
<tr>
<td>Naive Bayes Classifier (NBC)</td>
<td>26MB, 52MB, 104MB</td>
</tr>
<tr>
<td>Page View Count (PVC)</td>
<td>44MB, 88MB, 176MB</td>
</tr>
<tr>
<td>Matrix Multiplication (MM)</td>
<td>512<em>512, 1024</em>1024, 2048*2048</td>
</tr>
<tr>
<td>Principle Component Analysis (PCA)</td>
<td>1048576<em>8, 1048576</em>16, 1048576*32</td>
</tr>
</tbody>
</table>

device memory reduction object only. The *reduce* stage is not needed in this application, so there is no advantage to using shared memory reduction objects.

Our last application is Principle Component Analysis (PCA), which is also a map computation-intensive application. It aims to find a new (smaller) set of dimensions (attributes) that better captures the variability of data. Because it includes several steps that are not compute-intensive, we only perform the calculation of the correlation matrix on the GPU. In this step, given an $n \times m$ data matrix $D$ with $n$ rows and $m$ columns, we compute the matrix $S$, which is an $m \times m$ covariance matrix of $D$. The reduction object for this application stores the elements of the covariance matrix. This application also does not have the *reduce* phase, and it uses device memory only.

For every application example in our experiment, we use datasets of different sizes (small, medium, and large). The information about these datasets is shown in Table 2.1.
2.2.2 Evaluation of the Multi-group Scheme

To understand the behavior of our multi-group scheme, we executed the five reduction-intensive applications from our set of applications, varying the group numbers. Three different numbers of groups for each block were considered: 1, 2, and 4. Among our set of seven applications, MM and PCA are not included in these experiments, because they do not require reductions, and do not involve a shared memory reduction. kNN is not strictly a reduction-intensive application, but uses an in-object sort, and thus, the multi-group scheme is suitable for it. To focus on the impact of the number of groups, the results are reported from executions where the shared memory reduction objects are small and no overflow occurs during the entire computation. Finally, for k-means clustering, three different versions, KM20, KM40, and KM100 were created, by varying the number of clusters (centers), $k$. Recall that the size of the reduction object increases with increasing value of $k$.

Figure 2.3 shows the comparison of performance using different number of groups. These applications were tested with the large datasets. For KM20, the highest performance is achieved when the number of groups is 2. The total number of distinct keys in KM20 is
20, and thus the size of reduction object is very small. This implies a low combination overhead at the end of the computation. However, if the number of groups is 1, all the threads in one thread block update the same copy of the reduction object, and the contention can be very high. When the number of groups increases, the combination overhead increases, but the contention for each reduction object decreases. When we change the group number from 2 to 4, the increase of combination overhead is more significant as compared to the decrease in the contention overhead, so that the execution time increases.

KM40 has a larger reduction object. The combination overhead is now higher, but the contention during the updates to the reduction object is less severe. When we change the number of groups from 1 to 2, the benefit gained from the lower contention is almost the same with the performance loss due to the combination overhead, thus the execution time stays unchanged. When we increase the number of groups to 4, the performance gets worse due to the high combination overhead. The performance of KM100 follows a similar pattern with KM40 since it also has a large reduction object. The best performance is achieved with the group number of 1.

As we have discussed earlier, kNN is implemented by using the in-object sorting mechanism. Our experiments use 20 as the value of $k$. The reduction objects in this application fill up very frequently and the threads in each block need to perform overflow handling frequently, which, in turn, requires a bitonic sort. The bitonic sort incurs a large amount of data movement and is very time consuming. The combination overhead is relatively small compared with the sorting overhead. It turns out that in our implementation, as the number of groups increases, more threads take part in sorting. Thus the performance keeps improving when we increase the number of groups.
In the case of WC, the dataset used contains 90 distinct words, which makes it a reduction-intensive application. The best performance is achieved with the number of groups being 4. Although WC has a larger number of distinct keys than KM20, the contention on its reduction objects is more severe. This is because the time used to compare the newly generated keys with the keys in the reduction objects is long, and the hashing process and locating the key in the reduction object also take a substantial time. Thus, when the number of groups is increased, the contention decrease outweighs the combination overhead increase.

Although NBC has a small number of distinct keys (30), its execution time remains relatively unchanged with an increase in the number of groups. There are two reasons for this. On one hand, this application spends a relatively long time on processing each entry and generating key-value pairs, which makes the relative frequency of updating the reduction objects low, and the contention on the reduction objects also low. On the other hand, the hash function is able to locate the right buckets directly for this application, which makes the insertion to each bucket very fast.

PVC performs the best when the number of groups is 4. The PVC dataset we used involves 103 distinct entries, and thus, the total number of distinct keys in the first pass (which takes almost all the execution time) is also 103. The map function only computes the hash value of each entry and then emits it to the reduction object. Thus, this application has less computation, and the relative frequency of updates to each reduction object is very high. In addition, in this application, the hashing process and locating the key in the reduction object also take a relatively long time. Thus the contention on the reduction objects is quite high. Increasing the number of groups reduces such contention, resulting in better performance.
2.2.3 Comparison with Sequential Implementations

To show the overall benefits from our GPU-based framework, we compare the performance achieved on the GPU against the performance on a single thread application executing on 1 core of a CPU.

The results for the seven applications, and using three datasets for each application, are shown in Figure 2.4. The speedups vary considerably, depending upon the nature of the computation and the ratio between data transfer costs and execution times.

MM achieves the best speedup among all the applications. The reason is that MM does not involve any reduction. Moreover, the ratio between computation and amount of data to be processed is very high ($O(N^3)$ computation on $O(N^2)$ data). Thus, the GPU execution is not bound by data transfer costs or synchronization overheads.

The three versions of k-means, KM20, KM40, and KM100, along with WC, NBC, and PVC are reduction-intensive applications. Our framework is effective in speeding up these applications, especially for large datasets, but not to the extent of MM. This is because of a less favorable ratio between computation and data transfers, and because the reduction
Figure 2.5: Speedup over MapCG for Reduction-intensive Applications

stage takes a considerable amount of time. Still, the speedup of KM100 on the large dataset is nearly 180, the speedups of WC on all datasets are around 200, and the speedups of NBC on all datasets are close to 50. In the case of k-means, the speedups increase with increasing value of $k$, because the amount of computation performed on each record also increases.

kNN only achieves a small speedup of 3.6x with the large dataset. The reason is the negligible amount of computation in the map phase and the high data movement cost. PCA also has a relatively small speedup, though it does become 24x on the large dataset. The reason is that the total number of data units to be processed is small, and thus the degree of parallelism is low.

All the applications achieve higher performance as the datasets become larger. The reason is that when the datasets are small, the initialization and synchronization overheads are dominant compared to the processing time.
2.2.4 Comparison with MapCG

We now compare the performance of our framework against one of recent implementations of MapReduce on GPUs [47]. This framework had been shown to outperform one of the earlier implementations of MapReduce for GPUs, Mars [45].

MapCG does not take advantage of shared memory intensively. Instead, it stores a lot of intermediate key-value pairs to the hash table in the device memory. This not only leads to significant overheads of device memory access and synchronization, but also high costs of execution of the reduction phase. So, we will expect that its performance will not be good for reduction-intensive applications, where our framework can exploit shared memory very effectively.

Figure 2.5 shows the speedup of our implementations over the MapCG framework for four applications that we consider as the reduction-intensive applications. Each application is tested with small, medium, and large datasets, and three different values of $k$ for k-means are used. We can see that we achieve significant speedups over MapCG for all datasets, and all applications in this set. KM20 achieves the highest speedup over MapCG.
This is because it has a very small number of distinct keys, which leads to very high synchronization overheads (device memory based atomic operations) in the hash table used by MapCG. As the number of distinct keys becomes larger, MapCG performs somewhat better, though the relative speedups for large datasets are still very high. Similarly, WC and NBC both achieve very high relative speedups (about 90x on all datasets) due to the significant synchronization overheads in the map phase of MapCG’s implementations. PVC has a speedup of 5x, which is smaller compared with the other 5 applications. One reason is that PVC has a large number of distinct keys, which makes the synchronization overhead with MapCG’s hash table relatively low.

Figure 2.6 shows the speedup over MapCG for other three applications, which either spend a lot of time on map computations, or involve sorting. MM and PCA both do not use shared memory reduction objects. So the performance of our framework is almost the same as MapCG. For kNN, MapCG needs to sort the intermediate key-value pairs in device memory, which is time consuming. In comparison, our framework does most of the sorting in shared memory, although it also needs to sort the device memory reduction object before the computation finishes. Thus, our framework is more efficient, and as we can see from Figure 2.6, we have a factor of 3 speedup on the average over MapCG.

Overall, we can see that our method for exploiting shared memory leads to much better performance as compared with MapCG, for reduction-intensive applications. By comparing our speedups over MapCG and the speedups over sequential versions, we can see that for KM20 and NBC, MapCG even has a slowdown over sequential implementations, defeating the purpose in using a GPU. Thus, effective utilization of shared memory is crucial in benefiting from GPUs for MapReduce applications.
2.2.5 Comparison with Another Approach for Using Shared Memory

As we stated earlier in the paper, one recent implementation of MapReduce for GPUs does use shared memory [54]. This system, from Ji et al., is based on a more traditional approach for executing MapReduce, i.e., it has a shuffle or sort phase between map and reduce phases. As described earlier, their system uses shared memory as a buffer to stage input and output.

We obtained the implementation from Ji et al. and have compared it against our implementation. Among the applications we have used, implementations of KM, WC, and MM were available as part of their distribution. Because their implementation is based on a somewhat different API, it was not possible to execute other four applications on their framework. Thus, we have used these three applications.

The results from KM20, KM40, KM100, WC, and MM with large datasets are shown in Figure 2.7. Our system is always faster, though the exact speedup varies. Our framework achieves a very high speedup (56x) for WC. The reason is the large proportion of time spent on shuffling the key-value pairs in Ji et al.’s implementation. Also, for all versions
of k-means, our framework achieves a speedup of more than 6, as we get an effective use of shared memory and avoid shuffling overhead. MM only has the map phase, i.e., neither framework needs shuffling. We still achieve about a factor of 2 speedup for this application.

Overall, we can see that our approach for implementing MapReduce, including the way we use shared memory and avoid shuffling is significantly more efficient, and clearly outperforms the previous implementation.

### 2.2.6 Evaluation of the Swapping Mechanism

In some reduction-intensive applications, the number of distinct keys is large and the reduction objects in the shared memory can become full. The swapping mechanism implemented as part of our framework is invoked in such cases. Thus, while our framework can clearly correctly handle such applications, it is useful to explore the performance of our framework for these applications.

For this purpose, we experiment with WC and PVC, where the number of keys can vary (and can be quite large), depending upon the dataset. Thus, we examine the performance of these two applications by increasing the number of distinct keys, which leads to the increase.

![Figure 2.8: Swap Frequency under Different Number of Distinct Keys](image-url)
Figure 2.9: Comparison with Other Implementations for WC

Figure 2.10: Comparison with MapCG for PVC

of swaps. For each application, we use 5 datasets to evaluate the swapping mechanism. The numbers of distinct keys in the datasets vary from 300 to 4000. The size of each dataset is 100 MB. The number of buckets in each shared memory reduction object is 600. Thus for datasets with 300 distinct keys, no swap is involved.
We first investigate the change of swap frequency with the number of distinct keys increasing. The swap frequency \( f \) is defined as:

\[
f = \frac{N_{\text{swap}}}{N_{\text{task}}} \tag{2.3}
\]

where \( N_{\text{swap}} \) represents the total number of swaps in one execution, and \( N_{\text{task}} \) represents the total number of tasks in the workload. Larger values of \( f \) imply larger percentages of time spent on swapping.

The relationship between the number of distinct keys and the swap frequency is illustrated in Figure 2.8. We can see that as the number of distinct keys increases, the swap frequency increases, as we would expect. However, for both WC and PVC, the overall swap frequencies are quite low, although the swap frequency of PVC reaches 0.2% with the number of distinct keys being 4000. The results imply that in most cases, our system still performs much of the reduction work in shared memory.

We have also compared our system with MapCG and Ji et al.’s work when swaps are involved. Figure 2.9 shows the execution times of different frameworks for WC. Although our system involves swaps for number of distinct keys of 800, 2000, and 4000, it still achieves better performance than other two systems. MapCG has very high overhead because of the frequent accesses to global memory. Ji et al.’s framework spends considerable time shuffling. Figure 2.10 shows the comparison between our framework and MapCG for PVC (As was mentioned in Section 2.2.5, PVC is not included in Ji et al.’s distribution, and doesn’t seem straight-forward to implement because of their specialized API). Our system performs better than MapCG for most cases, though MapCG has a shorter execution time as the distinct key number increases to 4000.

Thus, we can see from the results that even when the frequency of swaps increases, our framework still performs well. This is because that our system keeps a large proportion
of the reduction object in shared memory, which results in lower device memory access frequency.

2.3 Summary

This chapter focuses on utilizing shared memory to accelerate MapReduce implementations on GPUs. In order to reduce the high overhead of data copying and synchronization on the device memory, we use a reduction-based approach, where we avoid storing the intermediate data (key-value pairs) in device memory. Instead, we perform reduction on the shared memory. We have also designed an approach for storing the reduction object on the memory hierarchy of the GPU. Finally, to further improve the performance, we have developed a multi-group scheme which allows us to balance the memory overhead and locking costs for the reduction objects.

We have evaluated the performance of our framework using seven applications. We have shown that our implementation delivers significantly higher performance than MapCG, a MapReduce framework which does not utilize shared memory intensively, and Ji et al.’s implementation, which takes a different approach for using shared memory.
Chapter 3: Accelerating MapReduce on a Coupled CPU-GPU

With the development of the hardware, CPU and GPU are integrated on the same chip, with the CPU and the GPU sharing the same physical memory space. Because the GPU is no longer connected through the PCIe bus, data transfer between devices is much more efficient.

Even though GPU is performing much faster for highly parallel and computation intensive applications, a multi-core CPU is still and will be a non-negligible computing power, and is faster than a GPU for certain types of computations such as conditional branches and data retrieval.

Accelerating MapReduce using a single GPU has been well studied by both other research groups or by ourselves, and also, accelerating applications using a CPU and a discrete GPU has also been studied. It still remains to be seen how we should efficiently accelerate applications, and MapReduce using the CPU and the GPU together, on a coupled CPU-GPU architecture supporting memory space sharing between the CPU and the GPU.

This chapter introduces the approaches we use to explore the new features of such a new architecture for MapReduce, using both the CPU and the GPU on an AMD coupled CPU-GPU. We will introduce several execution schemes and analyze each of them.
3.1 System Design

This section describes the main challenges involved, the design decisions we made, and the scheduling schemes we developed.

3.1.1 Challenges and Our Approach

There are several challenges that need to be addressed to utilize both the CPU and the GPU on a coupled architecture.

**Memory Requirements of the MapReduce Framework:** The execution of a MapReduce application tends to generate a large number of key-value pairs. These key-value pairs not only take up considerable memory, but also need frequent read/write operations to the device memory, leading to a low compute-to-device-memory-access ratio. Furthermore, shuffling the intermediate key-value pairs requires frequent synchronization and data movement.

**Efficiently Utilizing the Memory Hierarchy on the GPU:** Modern GPU architectures incorporate a memory hierarchy. At the top of the memory hierarchy is a small but fast *shared memory*, which is private to each streaming multiprocessor. Shared memory supports high-speed access and fast atomic operations. However, it is challenging to utilize it for the large number of key-value pairs that most MapReduce applications produce.

**Utilizing CPU and GPU for a Single Application:** We need to investigate how to effectively utilize both the CPU and the GPU for accelerating a single MapReduce application. Both CPU and GPU have significant computing power, though both of the them are better suited for certain task over others. At the same time, different MapReduce applications spend a different fraction of their time on map and reduce stages.
Task Scheduling across the CPU and the GPU: Dynamic load balancing across the CPU and the GPU is challenging, because of the overheads of certain operations on these architectures.

We now briefly outline how the above challenges were addressed. To lower the memory requirement by the MapReduce framework, we base our design of MapReduce on the distinct continuous reduction model. In this design model, an implicit data structure, named reduction object, is used. Key-value pairs generated by the map stage can be inserted and reduced to the reduction object directly, if the reduction operation in the MapReduce application is associative and commutative. This reduces the memory overhead of storing key-value pairs, and makes it possible to take advantage of the limited-sized shared memory.

To utilize CPU and GPU together for a single application, we propose two different approaches. The first is called map-dividing scheme, where map operations are divided between the CPU and the GPU. The second is called the pipelining scheme, where one device is used for map operations and the other is used for reduce operations.

To address the problem of dynamic workload scheduling on the CPU and the GPU, we develop novel master-worker load balancing models. We use one thread on the CPU device as the scheduler, which runs outside of the OpenCL kernels. The remaining CPU cores, as well as the streaming multiprocessors on the GPU, are the workers executing the tasks scheduled by the scheduler. Our proposed methods avoids the need of using atomic operations in task scheduling.

We now elaborate on the above ideas.
3.1.2 MapReduce based on Continuous Reduction

In the continuous reduction based implementation model, the map function, which is defined by the users, works in the same way as in the traditional MapReduce. It takes one or more input units and generates one or more key-value pairs. However, each key-value pair is inserted to the reduction object at the end of each map operation. Thus, every key-value pair is merged to the output results immediately after it is generated. A data structure storing these intermediate values of output is referred to as the reduction object. Every time a key-value pair arrives, the reduction object locates the index corresponding to the key, and reduces this key-value pair to that index in the reduction object. The reduction object is transparent to the users, who write the same map and reduce functions as in the original specification.

The method described above is based on the assumption that the operation in the reduce stage is associative and commutative, which covers most of the MapReduce applications, with some exceptions like a sorting application. If the reduce operation is non-associative-and-commutative, all the intermediate key-value pairs still need to be stored in the reduction object, and a shuffling (sort) is needed after all the key-value pairs are generated by the map stage. Our framework supports the parallel in-object sort, which is based on the bitonic sort [42].

The main advantage of this approach is that the memory overhead of a large number of key-value pairs is avoided since key-value pairs are reduced immediately after they are generated. This, in turn, is likely to allow us to utilize the shared memory on a GPU, especially if the number of distinct keys is not too large. In such case, continuous reductions can be first applied in the shared memory.
However, due to the limited size of shared memory (32 KB on the Fusion architecture) and the possibly large number of distinct keys, we need to address one potential problem of shared memory overflow. For such cases, we have support for swapping shared memory reduction objects to the device memory reduction object.

### 3.1.3 Map-Dividing Scheme

![Map-Dividing Scheme Diagram]

*Figure 3.1: Map-dividing Scheme*

The first implementation we created is based on the map-dividing scheme. As shown in Figure 3.1, all the workers follow the same execution steps. The scheduler dispatches the input tasks to each worker through the worker info array. Every worker corresponds to
one workgroup, which in turn consists of a number of working threads. After a task block is scheduled to a workgroup, the workload is evenly distributed to all the threads within that workgroup. Every thread processes the input tasks based on continuous reduction. It first invokes the user-defined map function, and generates the key-value pair(s), which are immediately inserted into the reduction object.

Note that besides a device reduction object in each device, we keep a private copy of the reduction object for each worker. This design is based on two considerations. First, for the GPU, each workgroup places its private copy of the reduction object in the shared memory. The shared memory supports faster read/write speed and faster synchronization. Thus the reduction in the shared memory can greatly improve the reduction performance. Second, for the CPU, as well as for the GPU, using private copies can reduce the contention overhead. If all the threads within one device update the same copy of the reduction object, contention on the buckets of the reduction object will be high.

After each worker receives the end message from the scheduler and finishes all its remaining tasks, all the threads belonging to this worker will merge the private reduction object to the device reduction object within each device. The merge of two reduction objects is to retrieve the key-value pairs from one reduction object and insert them to another reduction object in parallel. The merge operation may also be conducted during the process of computation. As we discussed in Section 3.1.2, the size of the shared memory on each streaming multiprocess of the GPU is limited, thus if the private copy of the reduction object is full, overflow handling is needed.

After the end of the computation on each device, the two device memory reduction objects from the CPU and the GPU need to be merged together. The merge process is done in parallel by the GPU, rather than by the CPU, since GPUs are good at processing highly
parallel operations. This is efficient for applications with large reduction objects, such as matrix multiplication.

The key challenge in this approach is dividing the map operations between the CPU and GPU cores. The relative speed of operations on the two devices can vary significantly depending upon the application. Therefore, we need dynamic scheduling. However, there are many challenges in supporting an efficient and effective scheme. An intuitive dynamic load balancing method between a CPU and a GPU will involve exiting the kernel and get a task block when a device is idle, as is also used in MapCG [47]. This method is not efficient since each task block assignment requires a kernel launch. A possible way of avoiding frequent kernel launch is to utilize the zero copy memory buffer, so that each device can access a global task offset from within the kernel. Because different devices access the same data structure in an atomic way, this turns out to be very expensive on current architectures.

Thus, we have developed this new dynamic scheduling method based on the master-worker model. The thread running on the first core of the CPU works as the scheduler. The remaining CPU cores and streaming multiprocessors on the GPU are workers. The entire workload is placed in the zero copy buffer, where data can be accessed by both the scheduler and the workers. The communication between the scheduler and the workers is through a data structure referred to as the worker info array, which is also placed a zero copy buffer, since the worker info must be accessed by both the scheduler and the workers from both devices. Task assignments and requests between the scheduler and the workers are sent through the worker info array. Each element in the worker info array is used independently between the scheduler and one worker.
Table 3.1: Worker Info Element

<table>
<thead>
<tr>
<th>Scheduling Message</th>
<th>Task Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>has_task</td>
<td>task_index</td>
</tr>
<tr>
<td></td>
<td>task_size</td>
</tr>
</tbody>
</table>

Each worker info element contains two types of information: the scheduling message and the task info. As shown in Table 3.1, the scheduling message only includes a flag: has_task, and the task info includes task_index and task_size. Algorithm 5 shows the major steps of this scheduling scheme. After the execution starts, the scheduler iterates on the has_task flags for all the workers. If the scheduler finds one worker’s flag to be 0, it assigns one task block to this worker. The task assignment includes setting the task_index and task_size fields in this worker’s worker info element. After that, it sets has_task flag to 1 to inform the worker that a new task block has been assigned to it. The first thread (thread 0) of each worker (workgroup) keeps on polling the has_task flag in the corresponding worker info element, and other threads just wait until thread 0 detects a newly scheduled task block. When thread 0 sees the value of this flag to be 1, it gets the task information, which includes task_index and task_size, and then sets the value of the flag as 0, notifying the scheduler that a new task block can be scheduled to this worker again. After that, the task block is processed by all the threads of this worker. As we said earlier, the global input is stored in the zero copy buffer. The value of task_index indicates the starting position of a particular task block in the global input. task_size represents the size of the task block that it is going to process. The flag has_task is also used to send the end message to each worker. When all the workload has been dispatched, the scheduler
sets the value of this flag in each worker info element as -1, notifying the workers to jump out of the kernel after processing their remaining tasks.

**Algorithm 5: Task Scheduling in the Map-dividing Scheme**

```plaintext
function Task_Scheduling(task_offset)

// the main steps of the task scheduling
while true do
    foreach worker i do
        if worker_info[i].has_task = 0 then
            /* assign a map task block to map worker i */
            worker_info[i].task_index ← task_offset;
            worker_info[i].task_size ← TASK_BLOCK_SIZE;
            worker_info[i].has_task ← 1;
            task_offset += TASK_BLOCK_SIZE;
        end
    end
    /* end the loop when all map tasks are scheduled */
    if task_offset ≥ total_num_tasks then
        break;
    end
end
/* the remaining step of the task scheduling */
Send end messages to all the workers;
```

### 3.1.4 Pipelining Scheme

Observing the fact that GPUs tend to be better at compute-intensive applications, while CPUs are more effective in handling control flow and data retrieval, we propose another scheme, which we refer to as the *pipelining scheme*. In this scheme, each device is only responsible for doing one stage of the MapReduce, for example, the GPU might perform all *map* operations and the CPU might perform all the *reduce* operations, or vice-versa. However, since the *map* stage involves a larger amount of computation, and the *reduce* stage tends to involve more branch operations and data retrieval, we expect the version where GPU performs map and CPU performs reduce to perform better.
Figure 3.2: Pipelining Scheme

This scheme involves a producer-consumer model. The key-value pairs are produced by the map workers, and are consumed by the reduce workers. Thus, buffers need to be used between the two different kinds of workers. For this scheme, we consider both the dynamic load balancing and the static load balancing.
Subgraph (a) in Figure 3.2 shows the processing structure of the pipelining scheme with dynamic load balancing. The dynamic load balancing used here is similar to what is used in the map-dividing scheme. However, there are several differences also. For each map worker, one circular buffer is used. Every circular buffer consists of a number of key-value blocks. Input tasks are only scheduled to the map device. The key-value pairs generated by a map worker are not inserted to the reduction object, but instead, they are stored into one key-value block in the circular buffer. When one key-value block is full, the next empty block in the circular buffer is used. Each key-value block is a reduce task block, and each time one key-value block is scheduled to one reduce worker. All the key-value blocks in all the circular buffers form a universal reduce task pool, and are scheduled by the scheduler to reduce workers dynamically. Since the circular buffers are accessed by both devices, therefore they are placed in zero copy buffers.

The scheduling procedure is shown in Algorithm 6. In each scheduling cycle, the scheduler not only assigns map tasks to map workers, but also assigns reduce tasks to reduce workers. The map tasks are scheduled in the same way as that in the map-dividing scheme. If an idle reduce worker is found, the scheduler will search the key-value blocks in all the circular buffers. The first full key-value block to be found will be scheduled to the idle reduce worker. If no full buffer is found in one searching iteration, the scheduler will skip this worker and continue with next worker. This prevents the scheduler from spinning on the empty key-value blocks.

Each key-value block not only stores key-value data, but also maintains status information. The full flag is set by the map workers and the reduce workers. Value 1 means this block is full or is ready to be scheduled to a reduce worker, while value 0 means this block is not full and can be used by a map worker to store key-value pairs. Whenever a
Algorithm 6: Task Scheduling in the Pipelining Scheme with Dynamic Load Balancing

```
map_task_offset ← 0;
block_index ← 0;
/*the main steps of the task scheduling*/
1 while true do
   /*schedule map tasks*/
   2 foreach map worker i do
      3 if worker_info[i].has_task = 0 then
         4 Assign a map task block to map worker i;
         5 map_task_offset+ = TASK_BLOCK_SIZE;
      end
   end
   /*schedule reduce tasks*/
   8 foreach reduce worker j do
      9 if worker_info[j].has_task = 0 then
         10 i ← block_index;
         11 foreach key-value block blocks[i] do
            12 if blocks[i].full&&blocks[i].scheduled then
               /*assign blocks[i] to worker j*/
               worker_info[j].task_index ← i;
               worker_info[j].task_size ← blocks[i].size;
               worker_info[j].has_task ← 1;
               blocks[i].scheduled ← 1;
               block_index ← (block_index + 1)%total_num_blocks;
               i ← block_index;
            end
         end
      end
      /*end the loop when all map tasks are scheduled*/
      23 if map_task_offset ≥ total_num_tasks then
         24 break;
      end
   end
   /*the remaining steps of the task scheduling*/
   27 Traverse the key-value blocks, assign full blocks to reduce workers;
   28 Send end messages to all the map workers;
   29 Wait for all map workers to finish execution;
   30 Schedule the key-value blocks which contain data to reduce workers;
   31 Send end messages to all the reduce workers;
```

map worker fills up a key-value block, it sets its full flag to 1, and finds next empty block to use. Whenever a reduce worker consumes a block, it sets its full flag to 0.

The scheduled flag indicates whether a key-value block has already been scheduled to a reduce worker; this flag helps to avoid repeatedly scheduling the same block to different workers. When a reduce worker finishes processing the key-value pairs in a block, it sets its scheduled flag back to 0.
When all the input tasks have been processed, the main scheduling cycle ends. Before exiting the entire scheduling process, the scheduler has some remaining work to do. First, it traverses all the key-value blocks on the map device and assigns full blocks to reduce workers. This step guarantees that the map workers which are still emitting key-value pairs have available key-value blocks to use. Next, the scheduler sends end messages to all the map workers since there are no longer any map tasks to be scheduled. After all the map workers finish execution, the scheduler iterates on all the key-value blocks, and the blocks which contain data are scheduled to idle reduce workers. When all the blocks have been scheduled, the scheduler sends end messages to the reduce workers.

Although dynamic load balancing provides better load balance, it brings scheduling overheads, and furthermore, one CPU core is only used exclusively for scheduling. In the pipelining scheme, each device is only responsible for one type of tasks, and all the processors on one device have the same processing speed, thus load imbalance may not be a significant problem. To investigate the possible benefit of avoiding dynamic scheduling, we also exploit the static load balancing to this scheme.

Figure 3.2 (b) shows the static load balancing for the pipelining scheme. We can see that the scheduler is no longer used. All the processors on the CPU participate in computation. Input tasks are evenly and statically distributed to each map worker. Like in the dynamic load balancing, each map worker has one circular buffer, which contains a number of key-value blocks. All the key-value blocks in all the circular buffers form a universal reduce task pool. This time, the key-value blocks are not assigned to the reduce workers dynamically, instead, they are evenly mapped to the reduce workers. map workers find empty key-value blocks to store key-value pairs, and reduce workers find full key-value blocks to consume. Synchronization is guaranteed by reading and setting the full flag in each key-value block,
in the same way as is described in the dynamic load balancing version. When all the
map workers finish their jobs, they set the \textit{finish} flag in each key-value block to be true,
notifying the reduce workers the end of the execution.

3.1.5 Load Imbalance and Runtime Tuning

So far in our discussion, the task block size for the dynamic scheduling in our frame-
work has been assumed to be fixed. In practice, however, determining the appropriate size
is a non-trivial problem. As we can expect, a large size can cause load imbalance, whereas
a small size can result in high scheduling overheads.

To overcome the difficulty of choosing optimal task block size, which can vary depend-
ing upon the application, we have developed a runtime tuning method. The method works
in the following way: At the beginning, a fixed and relatively small block size is chosen
for all the workers. We use these blocks to \textit{probe} the processing speed of each worker. We
count the total number of task blocks consumed by each worker, till a certain percentage of
the work is performed. After this point, the scheduler adjusts the task block size for each
worker by using the profiling information.

Let \( C_k \) be the total number of task blocks scheduled on the worker \( k \), and let \( n \) be the
total number of workers. A large block size is chosen as the \textit{initial size} (200,000 in our
implementation), and then the block size for each worker is calculated in the the following
way:

\[
tuned\_size_k = \frac{C_k}{(\sum_{i=1}^{n} C_i) / n} \times \text{initial\_size} \tag{3.1}
\]

After this tuning, the job scheduling frequency for each worker will approximate the
same, which reduces both the likelihood of load imbalance and the overhead of scheduling.
When the scheduling process is reaching the end, the runtime system further reduces the block size of each worker in order to reduce the difference among the finish times of the workers. This idea of reducing the block sizes at the end of the scheduling process is also used in the pipelining scheme. Overall, by maintaining large block sizes during the middle of the scheduling, the runtime tuning mechanism reduces the scheduling overhead. By tuning the task block size for each worker according to its computing ability, it improves the load balance among workers.

3.2 Experimental Results

We have extensively evaluated our framework with different applications, covering various types of applications for which MapReduce is used. We compare the performance of different approaches we have described in this paper against CPU-only and GPU-only implementations. We also analyze the influence of the task block size on the performance and show the benefits of our runtime tuning method.

3.2.1 Experimental Setup and Applications

Our experiments were conducted using an AMD Fusion APU (A8-3850) which integrates a 2.9 GHz quad-core CPU and a HD6550D GPU. The GPU has 5 streaming multiprocessors (SM), and each SM has 80 Radeon cores on it. Thus there are 400 Radeon cores on this GPU, with a clock rate of 600 MHz. Each streaming multiprocessor has a shared memory of size 32 KB. The system memory used in this machine is an 8 GB DDR3 memory with 1600 MHz clock frequency. The system memory is partitioned into a 512 MB device memory and a 7680 MB host memory.

We selected five commonly used MapReduce applications. These applications cover both reduction-intensive and map computation-intensive applications, where the former
Table 3.2: Applications and Datasets Used in our Experiments

<table>
<thead>
<tr>
<th>Application</th>
<th>Dataset Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-means, K = 40 (KM)</td>
<td>20,000,000 points</td>
</tr>
<tr>
<td>Word Count, 500 Distinct Words (WC)</td>
<td>100 MB text file</td>
</tr>
<tr>
<td>Naive Bayes Classifier (NBC)</td>
<td>207 MB vehicle dataset</td>
</tr>
<tr>
<td>Matrix Multiplication (MM)</td>
<td>2000*2000 matrices</td>
</tr>
<tr>
<td>KNN, K = 40 (kNN)</td>
<td>20,000,000 points</td>
</tr>
</tbody>
</table>

represents applications which have a large number of key-value pairs for each unique key, and thus, reduction computation time is significant. In contrast, the latter represents applications that spend most of their time in map stage. Among our applications, K-means clustering (KM) is one of the most popular data mining algorithms [53], which is a reduction-intensive application. Word Count (WC) is our second application, and can also be reduction-intensive if the number of distinct words is small. If the number of distinct words is very large, however, the reduction object cannot be held in shared memory, and then the overflow handling is needed. Naive Bayes Classifier (NBC) is a simple classification algorithm based on observed probabilities. When implemented using MapReduce, the total number of distinct keys is 30, i.e., it is also reduction intensive and has a small reduction object.

The fourth application is Matrix Multiplication (MM), which is a map computation-intensive application. If the two matrices being multiplied are of size \( x \times y \) and \( y \times z \) matrix, then the total number of distinct keys is \( x \times z \). Thus, for this application, we use the device memory reduction object only. The reduce stage is not needed in this application, and as a result, the pipelining scheme is also not feasible. Our last application is the k-nearest neighbor search (kNN) [3].
3.2.2 Impact of Task Block Size (Map-Dividing Scheme)

To investigate the influence of the task block size to the load imbalance between the CPU and the GPU, we execute each of the applications using the map-dividing scheme with different task block sizes. To quantify the load imbalance, we use the following expression:

$$\text{load imbalance} = \frac{|T_{CPU} - T_{GPU}|}{\max(T_{CPU}, T_{GPU})}$$  \hspace{1cm} (3.2)

where, $T_{CPU}$ represents the computation time by the CPU, and $T_{GPU}$ represents the computation time by the GPU.

The change of the load imbalance with the task block size increasing is illustrated in Figure 3.3. We can see that for the applications we used, the overall trend is that the load imbalance increases as the task block size increases, as we expect. MM has the highest load imbalance among all the applications, because a GPU workgroup is much faster than a
CPU workgroup. KM has the lowest overall load imbalance, as the relative speed difference between the CPU and the GPU is relatively small.

We will also like to note that if our runtime tuning method is used, the load imbalance is below 0.03 for all applications.

Figure 3.4 shows the normalized computation time of each application under different task block sizes. As we would expect, initially the computation time decreases with increasing task block size, as the scheduling overhead decreases. After a certain point, the computation time increases due to the increase in load imbalance. Different applications have different optimal values for the task block size. The computation times of KM and kNN do not change much from the task block size of 2000. As we indicated earlier, KM is not likely to have a load imbalance, as the processing speeds of the CPU and the GPU
are almost identical. WC, NBC and MM favor smaller task block sizes since the GPU processing speed differs very much with the CPU. Thus, a large block size can lead to an idle workgroup.

The runtime tuning does help to achieve near optimal performance for each application, which can be seen from Figure 3.5.

### 3.2.3 Comparison of the Performance from Different Approaches

To see the efficiency of our proposed schemes, we compare the total execution time of each application using different executing schemes. The total execution time includes the computation time, the data copy time, and the combination time. Note that for CPU-only versions, as well as for the versions where map tasks are only executed by the CPU, no data
copy is required since input data is originally in the host memory. For *map-dividing scheme* versions, because the input is stored in the *zero copy buffer*, which is also located in the host memory, no data copy is involved. Combination time is the time spent on combining the reduction objects from both devices, and does not apply to the pipelining scheme.

The schemes we compare here include:

- **CPU**: CPU-only version.
- **GPU**: GPU-only version.
- **MDO**: *map-dividing scheme* with a manually chosen optimal task block size.
- **TUNED**: *map-dividing scheme* using *runtime tuning method*.
- **GMCR**: *pipelining scheme*, the GPU does the *map* stage and the CPU does the *reduce* stage. We further create two different versions of these, which are **GMCRD** (with dynamic load balancing) and **GMCRS** (with static load balancing).
- **CMGR**: *pipelining scheme*, the CPU does the *map* stage and the GPU does the *reduce* stage. Again, **CMGRD** and **CMGRS** are the versions with dynamic and static partitioning, respectively.

Note that for CPU-only and GPU-only versions, we have implemented both static and dynamic workload partitionings. For each application, we choose the better version to compare with our multi-device versions.

The test results for all the applications are shown in Figure 3.5. For KM, GMCRS achieves the best performance (2.1 times faster than the CPU-only version), and is even faster than GMCRD. Static load balancing eliminates the scheduling overhead and does not require a CPU processing core to be dedicated to scheduling. Both dynamic and static
versions of GMCR are faster than all the other schemes. Although GMCR is a pipelining scheme, and therefore incurs I/O overheads to store the intermediate key-value pairs, this scheme works very well for an application like KM, which has a computation heavy map stage and a reduction intensive reduce stage. This allows effective utilization of GPU cores for floating point calculations in map stage, and use of CPU cores for reductions, which involves more control flow and data movement. For KM, the map-dividing scheme and its runtime-tuning version also achieve a speedup of 1.31 over the CPU-only version. GMCR is clearly faster than the CMGR, since executing map on CPUs and reductions on GPUs leads to ineffective utilization of both resources.

For WC, the CPU-only version is 2.7 times faster than the GPU-only version, as the hashing process and locating the keys in the reduction object is not executed efficiently on a GPU. The map-dividing scheme version has the maximum speedup of 1.24 over the CPU-only version. GMCRS and GMCRD do not perform better than the CPU-only version due to the I/O overhead. Similar as in KM, GMCR performs much better than CMGR, and static load balancing for pipelining scheme is more efficient than dynamic load balancing.

The speed gap between the CPU-only and the GPU-only for NBC is also large, i.e., CPU is 4.2 times faster. NBC has a small number of distinct keys (30 in total), which makes the reduction object very small, and leads to high contention among the threads on the GPU side. The best performance is achieved by the map-dividing scheme. MDO version has a speedup of 1.21 and the TUNED version has a speedup of 1.13 over the CPU-only version.

MM turns out to be very different than other applications. First, note that the pipelining scheme is not feasible for MM because there is no reduce stage in this application.
Second, because GPUs are good at processing highly parallel and compute-intensive applications, the CPU-only version takes more than 10 times as long as the GPU-only version. Furthermore, because of the regularity of the application, static load balancing used in the GPU-only version already evenly distributes the tasks to all the workgroups, ensuring high load balance. The dynamic load balancing schemes cannot further improve the load balance. On the contrary, in the map-dividing version, the scheduling overheads and the combination costs slow down the application. Thus, the dynamic scheduling schemes are even slower than the GPU-only version.

We observe that the GPU-only version for kNN is 1.57 times faster than the CPU-only version. This speed difference is relatively small, which makes the combination of the two devices efficient. MDO and its runtime tuning version achieve 1.5 speedup over the GPU-only version. GMCRS also has a speedup of 1.27 over the GPU-only version. Similarly, for the pipelining scheme, the GMCR versions are faster than the CMGR versions, and the static load balancing is faster than the dynamic load balancing. Note that in this application, the GPU-only and GMCR versions also have their input data in the zero copy buffer, thus no data copy time is involved. The reason is that this application involves less computation, and if the input data is copied from the host memory to the device memory, the data copy time would take a large fraction of the total execution time. For other applications, since their data copy time only takes a small percentage, we put their input in the device memory, which supports faster access speeds.

Note that for WC and NBC, the pipelining scheme performs worse than the single device versions. This is due to the high I/O overhead incurred by storing the key-value pairs in buffers. For both of these applications, the map stage is not very compute-intensive and the total execution time is mainly bound by the reduce stage. In the future, if coupled
architectures support faster I/O, the pipelining schemes may perform better. Also, for the pipelining scheme, static load balancing is more efficient than dynamic load balancing for most cases, implying the benefit of avoiding the scheduling overhead.

Overall, by integrating both the CPU and the GPU, our proposed scheduling schemes across different devices achieve considerable speedups over the single device versions in most applications. Data copy and the combination times take only a small or modest percentage of the overall execution time (below 15% and 0.5%, respectively). At the same time, analyzing the results for MM, we also note that for applications where the execution times on different devices vary significantly, and who already have very good load balance within each device, it is better to use only the faster device.

### 3.2.4 Overall Speedups from Our Framework

To see the efficiency and the scalability of our framework, we examine the relative and absolute speedups we are able to obtain from the use of the coupled CPU-GPU architecture. For this purpose, we compare the best CPU-GPU version with a *sequential* version (written in C/C++ and executed by a single core of the CPU) and a single CPU core MapReduce version (implemented using our MapReduce framework but executed on a single CPU core). Different versions of each application follow the same logic/algorithm.

<table>
<thead>
<tr>
<th></th>
<th>KM</th>
<th>WC</th>
<th>NBC</th>
<th>MM</th>
<th>kNN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequential</strong></td>
<td>7042</td>
<td>2017</td>
<td>2655</td>
<td>98810</td>
<td>1004</td>
</tr>
<tr>
<td><strong>MapReduce (1 core)</strong></td>
<td>7804</td>
<td>2057</td>
<td>2712</td>
<td>93647</td>
<td>1154</td>
</tr>
<tr>
<td><strong>Best CPU-GPU</strong></td>
<td>959</td>
<td>516</td>
<td>818</td>
<td>3445</td>
<td>112</td>
</tr>
<tr>
<td>(Relative Speedup)</td>
<td>(7.34x)</td>
<td>(3.91x)</td>
<td>(3.25x)</td>
<td>(28.68x)</td>
<td>(8.96x)</td>
</tr>
</tbody>
</table>

**Table 3.3: Parallel Efficiency Obtained from Our Framework**
The execution times are shown in Table 3.3. For all the applications, the MapReduce single CPU core version spends no more than 15% extra time compared with the sequential version. Note that for MM, the MapReduce single core version is even slightly faster than the sequential version. This is likely due to the different compiler optimizations used in OpenCL and C.

The speedup of the best CPU-GPU version for each application varies. MM has the maximum speedup, as the GPU is much faster than the CPU for this application. KM has very good load balance between the CPU and the GPU, and thus, integrating the 4 CPU cores and 5 GPU SMs achieves 7.34 speedup over the sequential implementation. In the case of kNN, we also achieve a good load balance and the GPU is faster than the CPU. The overall speedup is 8.96, which is also large. WC and NBC have significant load imbalance between the CPU and the GPU, and the CPU is much faster than the GPU. Thus, the speedups are limited by the parallel efficiency on 4 CPU cores.

3.3 Summary

This chapter focuses on scheduling MapReduce tasks across CPU and GPU on a coupled CPU-GPU chip. We propose two different scheduling schemes: the map-dividing scheme, which dynamically divides the map tasks to both devices; and the pipelining scheme, which executes the map and reduce stages on different devices. To automatically achieve high load balance while keeping scheduling costs low, we use a runtime tuning method for the map-dividing scheme. Our implementation of MapReduce is based on continuous reduction, an design alternative which avoids the memory overhead of storing key-value pairs and makes it suitable to utilize the small but fast shared memory.
We have evaluated the performance of our framework using five applications. For 4 of the applications, our system achieves 1.21 to 2.1 speedups over the better of the CPU-only and GPU-only versions. The speedups over a single CPU core execution range from 3.25 to 28.68. The runtime tuning method we have developed achieves very low load imbalance, while keeping scheduling overheads low.
Chapter 4: Scheduling Methods for Accelerating Applications on Architectures with Heterogeneous Cores

Last two chapters focused on implementing MapReduce for GPUs and integrated CPU-GPUs. We observe that some of the methods used in the scheduling of MapReduce tasks could be used also for other applications, which means, these scheduling methods are general enough for heterogeneous architectures for any kind of applications. Also, these scheduling policies are not limited to coupled CPU-GPUs, but all parallel architectures with heterogeneous cores, including multi-GPU architectures. In this chapter, we introduce several locking-free heterogeneous scheduling schemes that are general enough for different applications on different hardware platforms.

4.1 Problem Definition

Broadly, task scheduling methods can be divided into static scheduling and dynamic scheduling methods. A static scheduling method determines the task assignments before the execution starts. It tends to be simple and involves little runtime scheduling overhead. However, the optimal task division ratio is hard to predict for our problem, since the relative performance of CPU and GPU cores depends on the application and even the dataset. Thus, we clearly need to develop dynamic task scheduling methods for our problem.
The main challenge in developing a dynamic scheduling scheme for a CPU-GPU architecture arises because of the overheads of certain operations on different CPU-GPU architectures. A traditional and straightforward dynamic method suitable for CPU-GPU nodes, which we can also refer to as the kernel relaunch approach, can be as follows [88, 4, 91]: Input units can be split into task blocks and queued at the host side. The execution on each device will exit the kernel and get a new task block when it is idle. However, the kernel launches lead to a high overhead, which arises because of several reasons, such as the global synchronization cost (as each kernel is launched, a global fence or barrier is needed at the host side) and the additional CPU cycles which are needed to handle a kernel relaunch.

Outside of the specific context of CPU-GPU nodes, dynamic task scheduling has been studied over a wide variety of platforms, ranging from multi-core processors to large-scale clusters. Different solutions for dynamic workload distribution have been proposed. Cilk [7, 39] initiated the work stealing approach, first applying it among threads that share a common and coherent address space. This work has been extended in other contexts. For example, Dinan et al. have proposed a scalable work stealing framework for multicore clusters [34]. Both these two works rely on a locking mechanism, supported by either a coherent shared memory, or the PGAS model.

Now, return to the CPU-GPU nodes, it appears that expensive kernel relaunches can be avoided by using work stealing, based on either a physical or a virtual memory that can be accessed by both CPU and GPU cores. As we had discussed before, such a memory is available on configurations involving a discrete GPU and the Fusion CPU-GPU, in the form of zero-copy buffer. However, what is not currently supported are atomic or locking operations on such a memory. It turns out that work stealing has also been attempted on CPU-GPU nodes [40]. In this approach, at the beginning of the task scheduling, tasks can
be evenly distributed to both devices. The device that finishes processing its tasks before the other one, can steal tasks from the other device. Because of lack of atomic operations, such stealing requires kernel relaunches, which again can be expensive.

Thus, we can see that even an efficient implementation of work stealing will either require kernel relaunch, or atomic operations must be used, which are currently not always supported. Furthermore, what we have is a non-uniform access shared memory between CPU and GPU, with slower accesses by the GPU. Thus, in any mechanism we might use, access to this memory by the GPU should not be frequent.

In summary, we have a task distribution problem over non-coherent, non-uniform access shared memory, for which new solutions are needed. Our solutions are presented in the next section.

4.2 Dynamic Scheduling Methods

Based on the discussion in the previous section, we can see that locking-free dynamic scheduling methods are needed. We now describe several different candidate designs for meeting this requirement. For all these methods, we put the input tasks in the zero-copy memory buffer, where data could be directly accessed by all the processing units. The framework does not need to transfer data before the kernel is executed. The access to the input data is in a coalesced way, and utilizing this memory can overlap the computation with data copying.

4.2.1 Master-worker Scheduling Method

The first locking-free scheduling method we use is the master-worker method. As shown in Figure 4.1, in this method, the first core of the CPU is used as the master (or
scheduler), and the remaining CPU cores and the GPU SMs are the workers. (For simplicity, we only show one GPU, although our method is general for any number of GPUs, which is the same for the remaining two methods.) A worker information including the status (busy or idle), task assignment flag (for new task assignment notification), and task offset for each worker core is placed in the zero copy memory buffer, where data can be read and updated by both the workers and the master. The master core iterates on each worker’s status information. Every time it finds an idle worker, it retrieves a task block from the task pool and assigns this task block to the idle worker by setting task offset and task assignment flag. In this scheduling method, worker cores do not retrieve task blocks actively, instead, when any of them finishes processing its allocated tasks, it just sets its status to idle and waits there, and the master core will schedule a new task block to it. Therefore, the competition on the global task offset among workers no longer exists; only the master core will read and update its value. Obviously, this scheduling method avoids
the need of using locks to guarantee mutual exclusion to the global task offset, and only a single kernel is to be launched without being interrupted.

Despite the benefit of locking-free scheduling of the master-worker method, this method requires a CPU core to be used exclusively for task scheduling. For applications where CPUs outperform GPUs significantly, dedicating a CPU core to task scheduling is a significant waste of resources. This has motivated the design of the next two schemes, which are both based on token passing.

### 4.2.2 Core-level Token Passing Method

**Algorithm 7: Task Scheduling in the Core-level Token Passing Method for GPU Side**

```plaintext
while true do
  /* first thread in the SM retrieves tasks */
  if (local_id == 0) then
    status[worker_id].busy ← 0;
  /* wait on the token */
  while true do
    if (token == worker_id) then
      if (no tasks available) then
        /* notifies local threads to exit */
        finish ← 1;
      else
        task_index ← *global_offset;
        *global_offset ← *global_offset + BLOCK_SIZE
        status[worker_id].busy ← 1;
        passed ← pass_token();
        break;
    else
      barrier();
    if (finish) then
      break;
  /* barrier within one thread block */
  barrier();
  if (finish) then
    break;

  /* process the task block on the SM, with thread 0 in the thread block periodically passing the token */
```

84
Figure 4.2: Core-level Token Passing Method

Figure 4.2 shows the core-level token passing method. In this method, all CPU cores and GPU SMs are treated equally as workers, with unique worker ids, and they are connected by a token ring. An integer variable token is placed in the zero-copy memory buffer and can be accessed by workers from both the CPU and the GPU. A worker gets the token when the value of token equals its worker id. Only when a worker gets the token can it retrieve a new task block from the task pool. The status of each worker is placed in the zero-copy memory buffer. After a worker retrieves a task block, it checks to see the status of other workers. The token will be passed to next idle worker in the token ring. This scheme ensures that only one worker can be retrieving tasks at any given time. Thus, we achieve the goal of mutual exclusion, without using any locking operations.

Algorithm 7 shows how a GPU SM gets task blocks and passes tokens. The CPU cores follow the same logic, which is not shown here. Whenever an SM finishes processing a task block, the first thread (thread 0) in the thread block sets its status to idle and waits on the token. When it finds the token passed to this thread block, it knows that it is granted
the permission to obtain a new task block. The process of obtaining the task block involves reading and updating the global task offset. After the task block is retrieved, thread 0 sets its status to busy and tries to pass the token to an idle worker attached to the token ring. It is possible that no idle worker is found immediately after thread 0 retrieves a task block. Thus, when the newly retrieved task block is being processed, thread 0 periodically attempts to pass the token, which prevents other idle workers from waiting for too long.

Though core-level token passing method allows all CPU cores and all GPU SMs to focus on task processing, the first thread on each SM and every CPU thread need to spend a substantial number of computing cycles on checking all other workers’ status information whenever the token is being passed. Also, if a large number of workers are involved, the cost of checking the status and the frequency of token passing at the core level will be high. These issues motivate the next scheme, which is a combination of the token passing and the intra-device locking.

Compared with the core-level token passing method, the device-level token passing method is more coarse-grained. In this scheduling method, a token is passed between the CPU and the GPU (i.e., the devices), rather than the cores.

4.2.3 Device-level Token Passing Method

As shown in Figure 4.3, every time a device is idle, it checks whether it is currently holding the token; if so, the first thread on the device retrieves a large chunk of tasks, which is composed of a number of smaller task blocks. Within each device, locking method is used to dynamically distribute the set of smaller task blocks among the cores or SMs. Moreover, the first thread now tries to pass the token to the other device.
Algorithm 8: Task Scheduling in the Device-level Token Passing Method for GPU Side

```
while true do

  /*first thread in the device retrieves tasks*/
  if global_id == 0 then
    status[device_id].busy ← 0;
    *has_task ← 0;
    if *token == device_id then
      /*notify each device to exit*/
      if no tasks available then
        *has_task ←− 1;
      else
        device_offset ← *global_offset;
        *global_offset ← *global_offset + CHUNK_SIZE;
        *has_task ← 1;
        passed ← pass_token();
        status[device_id].busy ← 1;
      end if
    end if
  end if

  /*barrier for the whole device*/
  global_barrier();
  if *has_task == −1 then
    break;
  end if
  if *has_task == 1 then
    {/*process the chunk on the device with the first thread on the device
     periodically passing the token*/}
  end if

```

Figure 4.3: Device-level Token Passing Method

The logic for the GPU is illustrated by Algorithm 8 (CPU side follows the same overall idea and the details are omitted). Whenever a device finishes processing the task chunk it
retrieved, the first thread on each device sets its status to idle and checks the token, and if the
token is currently held by the device, it gets a new task chunk. Other threads on this device
just wait there until the task chunk is retrieved. Thus, device-level synchronization [119]
is required. After the task chunk has been retrieved, the first thread sets the status of the
device to busy and tries to pass the token to the other device when the other device is idle.
Similar to the core-level token passing, it is possible that immediately after a device gets a
task chunk, the other device is still busy and the token cannot be passed. In such a case, the
first thread on the device will keep periodically checking to be able to pass the token to the
other device.

By creating a hybrid scheme, i.e., combining device-level token passing with intra-
device locking, the status checking costs and token passing frequency of the core-level
token passing scheme are greatly reduced. At the same time, although atomic operations
are used to implement intra-device work distribution, the overhead is not high since tasks
are scheduled in relatively large blocks. Moreover, at the device level, this method is still
locking-free.

4.3 Experimental Results

We had the following goals in our experiments. First, we examine the overall effec-
tiveness of scheduling schemes by measuring the acceleration of the application achieved
through the use of CPU and GPU cores. In the process, we also compare different methods
for scheduling that we have introduced, and see the overheads they introduce. We would
also try to see the scalability of our scheduling methods while increasing the number of
devices. Finally, we also compare CPU-GPU execution against StarPU [4] and OmpSs [9],
two recent heterogeneous programming systems that can allow an application to use both CPU and GPU cores at the same time.

### 4.3.1 Experimental Setup

We have implemented our scheduling framework on both a coupled CPU-GPU architecture and a decoupled CPU-GPU architecture. The coupled CPU-GPU we used is an AMD Fusion APU (A3850), which integrates a 2.9 GHz quad-core CPU and a HD6550D GPU. The GPU has 5 streaming multiprocessors (SM), and each SM has 80 Radeon cores on it, with a clock rate of 600 MHz. Each streaming multiprocessor has a shared memory of size 32 KB. The system memory used in this machine is an 8 GB DDR3 memory with 1600 MHz clock frequency.

The decoupled CPU-GPU node we used contains a multicore CPU and two NVIDIA Fermi cards (M2070). It has 14 SMs and each SM has 32 cores running at a frequency of 1.15 GHz. The GPU is connected to 12 Intel Xeon 5650 CPU cores. Each CPU core works at a frequency of 2.67 GHz. The evaluation on the decoupled CPU-GPU includes both CPU+1GPU and CPU+2GPU scenarios.

### 4.3.2 Applications Used

We use 6 applications involving different complexity and properties to evaluate our framework. These applications vary in both communication pattern and input partitioning, and exhibit different load balancing in task partitions. We now introduce each of them.

**Kmeans:** Kmeans clustering is one of most popular data mining algorithms. It clusters each point to one of $K$ cluster centers based on euclidean distance. The clusters are then updated using the clustered points. Thus, this application follows a generalized reduction pattern. The partitioning of the input is straight forward, i.e., splitting the points into equal
partitions, and each partition will be scheduled as a task. The calculation of the new cluster centers are conducted in a reduction-based manner. Points assigned to one cluster is immediately reduced to the reduction result. The implementation on the GPU first conduct local reductions by each thread block, with the reduction objects stored in the on-chip shared memory. A final combination is conducted before the computation finishes. We use a three dimensional dataset (10M points) with 40 cluster centers.

**Gridding Kernel:** Gridding Kernel is a data intensive algorithm which is often used in astronomy software pipelines [111, 112]. It transforms visibilities sampled in the \((u, v, w)\) space from various frequency channels into a generalized grid. It also follows the pattern of generalized reduction, and the input partitioning is in the same way with Kmeans. We used 4 million visibilities as the input.

**Jacobi:** 3D Jacobi, one of the most commonly used *stencil* applications, updates each element in a 3D matrix with the arithmetic mean of its surrounding neighbors. The input matrix is tiled into equal 3D blocks, and each block is scheduled as a task. A \(768^3\) matrix was used.

**Sobel:** Sobel filter is a image processing application. It applies two \(3 \times 3\) convolution matrices to a 2D matrix. Similar with Jacobi, this is also a stencil application, and we also tile the input into blocks for scheduling. An image of size 16, 384\(^2\) was used.

**Moldyn:** Molecular dynamics [51] is an application simulating the interaction and motion of molecules in a period of time in a 3D space. The major computation is to compute the accumulated force to each molecule. An interaction list (edges) is built based on the position of all the molecules (nodes) before the force computation, after which force computation is applied to each interaction. The results of force computation are reduced into a reduction object, which is used for updating the speed of the molecules. The partitioning of
the input is based on the molecules (nodes), in which case nodes are evenly partitioned and the corresponding interaction list (edges) for a specific node partition is grouped. These edge groups are scheduled dynamically to the computation units (CPU cores or GPU SMs). The nodes are partitioned in a way that each node partition could fit into the GPU’s on-chip shared memory, which supports much faster reduction. Because of the randomness of the density of molecules in the 3D space, the edge groups exhibit obvious load imbalance. This application is categorized as *irregular reduction* due to the indirect access through the interaction list. The dataset used contains 1M nodes and 120M edges.

**Euler**: Similar with Moldyn, Euler [29] is also an irregular reduction application. It is a Computational Fluid Dynamics application which conducts computations to an unstructured mesh in a number of time steps. Each time step updates the velocity of the nodes in the mesh according to the interaction list. The workload partitioning is conducted in the same way as Moldyn. We used a dataset involving 1M nodes and 5M edges.

While implementing Moldyn and Euler on the decoupled CPU-GPU system, we store a full copy of the node data into the device memory on the GPU, since each node may be accessed multiple times. In this way, we avoid the repetitive access of the zero-copy buffer.

### 4.3.3 Overall Performance Improvements and Comparison of Different Scheduling Methods

In this section, we will compare the execution times of the applications using different scheduling methods, including single device execution, traditional kernel relaunch method, and the three locking-free scheduling methods we have proposed. The single device execution includes both CPU-only and GPU-only versions, each of which is implemented using intra-device dynamic load balancing among CPU cores or GPU SMs. We also include the
performance of a version based on optimal static partitioning between the CPU and GPU (hand-optimal). The hand-optimal version is implemented in a way that input workload is partitioned statically and thus there is no scheduling overhead involved. The partitioning ratio is determined from the execution of a dynamic scheduling: portion of work to a device is the same as the portion of work that was scheduled to a device in dynamic scheduling. All execution times we report include the data copy time, the computation time on different devices, and the combination time (i.e. the time for final combination of output from different devices). The results reported in this section will be limited to single GPU on the node, later, we will consider performance with multiple GPUs.

**Results from a Coupled CPU-GPU Chip:** The results for different versions on the coupled CPU-GPU are shown in Figure 4.4.

For Kmeans, all the locking-free scheduling methods we have developed outperform the traditional kernel relaunch method. Although core-level token passing method utilizes all CPU cores for computations, it does not have a visible advantage over the master-worker method. The device-level token passing method has the highest performance, which is 1.52
times faster than the CPU-only version, which, in turn, is slightly faster than the GPU-only version. The device-level token passing method has a 12% improvement over the core-level token passing, which clearly shows the benefit of decreasing the token passing overhead.

For Gridding Kernel, the CPU-only version is 2.8 times faster than the GPU-only version. As a result, utilizing both the CPU and the GPU has a relatively smaller benefit over the CPU-only version. Because of the high token passing overhead, the core-level token passing even has a small slowdown in performance. Master-worker method dedicates a CPU core to scheduling, which makes it less efficient compared with device-level token passing. Again, the device-level token passing results in best performance among the CPU-GPU versions, which is 1.25 times faster than the CPU-only version.

For Jacobi, the GPU-only version is 1.95 times faster than the CPU-only version. When both the CPU and the GPU are used together, the maximum speedup over the single device version (1.35x) is achieved by the device-level token passing method. Because Jacobi involves a relatively large amount of computation, the kernel relaunch overhead and core-level token passing overhead is not substantial compared with the time spent on computation. Also, since the CPU is much slower than the GPU, dedicating a CPU core on scheduling in the master-worker has only a very small influence on performance. The CPU-GPU versions with kernel relaunch, master-worker, and the core-level token passing method achieve speedups of 1.18, 1.20, and 1.28, respectively, over the GPU-only version.

Compared with Jacobi, Sobel Filter is less compute-intensive. Thus, the total execution time is more easily impacted by the scheduling overhead. We can see that the kernel relaunch scheduling method even has a slowdown compared with the single device versions. The CPU-GPU version with device-level token passing method has the highest speedup, 1.41x, over the better of single device versions.
The two irregular reductions share a similar execution property, which is that the GPU is much faster than the CPU (approximately 3x). This is due to the efficient utilization of the shared memory on GPU. Due to the overhead of scheduling, kernel relaunch, master-worker and core-level-token passing methods do not have substantial speedup over the GPU only-version. Device-level token passing method is still the most efficient, which achieves a speedup of 1.11 and 1.18 over the GPU-only for the two applications.

The results presented in Figure 4.4 can also help us quantify the scheduling overheads of different schemes. This is because we have included the hand-optimal versions, which involve the same work distribution as the dynamic scheme, but hard-coded in the application. By comparing dynamic schemes against the static ones, we see that overheads of scheduling vary across schemes. The device-level token passing method has the lowest overhead, which is less than 8% (on average).

**Results from a Decoupled CPU-GPU**: Figure 4.5 shows the results for different versions on a decoupled CPU-GPU architecture. Here we show the scenario where only one GPU is used together with the CPU. Because all three locking-free schemes are based on the
zero-copy buffer, and we have already seen that device-level token passing outperforms the core-level token passing method, our evaluation here does not include core-level token passing scheme. On the other hand, because kernel relaunch method involves different mechanisms, we have implemented and evaluated versions that use this method as a baseline.

From the results, we can see that on the whole, the device-level token passing method has a better performance than the kernel relaunch method, although the performance difference varies for different applications. For applications that are compute-intensive, i.e., Kmeans, Gridding Kernel, and Jacobi, the kernel launch overhead for the kernel relaunch method is less obvious, thus it is nearly as efficient as the device-level token passing method. For Kmeans, the device-level token passing method is the fastest, which is 1.32 times faster than the better of the single device versions. For Gridding Kernel, a speedup of 1.24 is achieved by device-level token passing method over the better of the two single device versions, i.e., the CPU-only version. Master-worker model has the lowest performance compared with the other two, which has an only 1.13x speedup over the CPU-only version. This is because for this application, the CPU is much faster than the GPU, and the master-worker method devotes one CPU core to scheduling, instead of computation.

For Jacobi, device-level token passing method is the fastest, and is 6% and 3% faster than kernel relaunch and master-worker respectively. The best speedup achieved for Jacobi over the better of the CPU-only and GPU-only version is 1.81.

For Sobel Filter, device-level token passing method has a more obvious advantage over the kernel relaunch method, as the total execution time is more sensitive to the scheduling overhead. The best speedup over the better of the CPU-only or GPU-only version is 1.30. For Moldyn and Euler, device-level token passing method for these two applications
achieves speedups of 1.88 and 1.35, respectively. Again, the device-level token passing method is achieving a close performance with hand-optimal version (less than 5% slowdown).

Figure 4.6: Computation Time for Core-level Token Passing Method and Device-level Token Passing Method while Increasing the Number of Cores/SMs
4.3.4 Detailed Comparison of Core-level and Device-level Token Passing Methods

As is described in Section 4.2.2, the motivation of proposing *device-level token passing method* is the potential poor scalability of the *core-level token passing method* with increasing number of cores. As we saw earlier, device-level scheme does turn out to be more efficient on both architectures. However, to better understand the scalability of both the schemes with increasing number of cores, we designed a new experiment. The results we report here are from the AMD Fusion APU, though similar trends are seen with a discrete GPU. The experiment has two components, which are as follows. In the first component, we start with a GPU-only execution, which only involves 5 GPU SMs on the APU and the work is evenly and statically distributed to all the SMs. We gradually increase the number of CPU cores, and investigate the change of the computation times for each application using both the core-level token passing method and the device-level token passing method.

The second component of the experiment starts with a CPU-only version, that is, only the 4 cores on the CPU are doing the computation, which is evenly and statically distributed among the four cores. By increasing the number of GPU SMs, we can see the change of the computation time of each application with both the scheduling methods.

The results from the first and the second components of this experiment are shown in Figure 4.6. From the first set of results (solid lines in the figure), we can see that the device-level token passing method has good scalability for each application. Core-level token passing method has a higher token passing overhead, thus its computation time is almost always longer than device-level token passing method. The overhead of core-level token passing is more obvious for Sobel Filter, which is less compute-intensive, and thus, its computation time is more easily influenced by the scheduling overhead. Due to the
increase of scheduling overhead, core-level token passing method is showing slow down with the number of CPU cores increasing for Moldyn and Euler.

Now, considering the second component, i.e., the dashed lines in the figure, for each application, device-level token passing method has an overall better performance. For Gridding Kernel, core-level token passing method even has a slight slowdown in performance when more SMs are involved. Thus, overall, we can see that core-level token passing has very high overheads, and device-level token passing can scale very well.

![Figure 4.7: Scaling to Multiple GPUs](image)

**4.3.5 Scaling on Multiple GPUs**

Increasingly it is becoming common to have multiple GPUs connected to the same node. We have also evaluated our system on a decoupled CPU-GPU system comprising of a multicore CPU and two NVIDIA GPUs. To see how our scheduling schemes scale on multi-devices, we plot the normalized execution times of different hardware configurations for each scheduling method, which is shown in Figure 4.7. Again, here we are only showing
the kernel-relaunch method, the master-worker method, and the device-level token passing method.

The performance improvement from CPU+1GPU co-processing to CPU+2GPU varies depending on the application. Data to be processed by a GPU has to be loaded to the device via the PCIe bus. This can be done in multiple ways, e.g. coping the data to the device memory before the computation starts, copying it asynchronously with the computation, or accessing the data through the zero-copy buffer. Our framework use the last approach, consistent with our scheduling scheme. However, as the number of GPUs increases, the bandwidth of the PCIe bus may become a bottleneck and the data copy time will dominate the overall time.

For applications which have high ratio between computation and amount of data access, e.g., Kmeans and Sobel, the scalability is higher.

However, for applications which have low computation to data access ration, such as Jacobi, combining another GPU contributes a smaller improvement in performance. Overall, the token passing method is showing a better performance compared with the traditional kernel relaunch method for the CPU+2GPU cases.

Specifically, for Kmeans, different scheduling methods result in very similar performance, and the CPU+2GPU version using device-level token passing achieves a speedup of 1.79 over the CPU+1GPU version. For Sobel Filter, which has a higher computation to memory access ratio than the other stencil computation we have used (Jacobi), the CPU+2GPU version for device level token passing is showing a speedup of 1.52 over the CPU+1GPU version. Because GPU is much slower than CPU for Gridding Kernel, utilizing an additional GPU brings a relatively smaller speedup of 1.36. CPU and GPU have a very similar performance for Jacobi, as we had seen earlier from Figure 4.5. When we
are scaling the computation from CPU+1GPU to CPU+2GPU, we only see a limited performance improvement (1.25x for device level token passing). This is because that Jacobi has a very low computation to memory access ratio (approximately 1:1). The input data is stored in the zero-copy buffer, which is accessed by the CPU and both GPUs directly. The main bottleneck of the data access is the PCIe path from the CPU to the two GPUs, which is shared by both GPUs. Thus when the second GPU is added, the bandwidth for copying data to each GPU is reduced to half of a single GPU case. Similarly, a speedup of 1.40 and 1.56 could be seen by the CPU+2GPU version over the CPU+1GPU version for Moldyn and Euler, respectively.

We can also see that, with the number of GPUs increasing, the relative performance of master-worker method is becoming closer to that of the device level token passing method. This is because that as the total computing power in the system increases, sacrificing a CPU core has a less influence to the overall performance.

### 4.3.6 Comparison with StarPU and OmpSs

StarPU [4] is one of the recent scheduling systems for heterogeneous architectures. It provides APIs for data partitioning, task submission, and scheduling. Similarly, OmpSs [9] supports a programming model where users can offload loops to multi-core CPUs and GPUs by adding OpenMP directives.

StarPU and OmpSs both support a number of scheduling policies, including greedy scheduling, random scheduling, work stealing, and others. These scheduling policies are all based on kernel relaunch, i.e., task scheduling is performed at the host side. They both support automatic overlapping of data transfer with computation using asynchronous
Our framework utilizes zero-copy, which can also overlap the PCIe transfer with computation, in addition to avoiding kernel relaunches.

For our comparison, we implemented three applications using StarPU and OmpSs, representing the three different computation patterns that are involved in the six applications we have used. While implementing them on StarPU and OmpSs, we use the same logic, algorithm and the data partitioning methods as the corresponding implementations using our scheduling framework. Since our goal was to compare the efficiency of different scheduling methods, we have carefully implemented applications on the different frameworks in a way that that single device executions take nearly the same amount of time as ours.

The experiments were conducted on a decoupled CPU-GPU node, which was used in our previous experiments as well. Existing scheduling schemes in StarPU and OmpSs are not optimized for coupled CPU-GPUs, and thus, to be fair, comparison was limited to a decoupled CPU-GPU node. The same datasets as in Section 4.3.3 were used. The experiments were conducted for both single- and multi-GPU scenarios.
As we noted earlier, both StarPU and OmpSs support different scheduling schemes. We tried different policies provided by these systems and report the results from the best version. Similarly, the execution times we report from our framework are with the device-level token passing method, which gave the best performance in our previous experiments.

Figure 4.8 shows the execution times of StarPU, OmpSs and our framework, comparing CPU-only, GPU-only, CPU+1GPU and CPU+2GPU executions. We can see that the three systems have very similar performance for CPU-only and GPU-only executions, establishing that comparable applications were used as the basis for evaluating performance of scheduling.

For all the three applications with CPU+1GPU configuration, our framework is faster than StarPU (1.12x, 1.08x, and 1.19x faster for Kmeans, Jacobi, and Moldyn, respectively) and OmpSs (1.13x, 1.10x, and 1.21x faster for Kmeans, Jacobi, and Moldyn, respectively). For the CPU+2GPU configuration, we also outperform StarPU (1.13x, 1.10x, and 1.19x faster for Kmeans, Jacobi, and Moldyn, respectively) and OmpSs (1.14x, 1.09x, and 1.20x faster for Kmeans, Jacobi and Moldyn, respectively). The likely reason for the performance difference is that the task scheduling method we have developed is more efficient. Kmeans and Jacobi are more computation intensive applications, and thus the scheduling overhead is less obvious than Moldyn. In comparison, StarPU and OmpSs’s task scheduling involves frequent kernel relaunches, which contributes to slower CPU-GPU performance.
4.4 Summary

This chapter has focused on the problem of exploiting parallelism across heterogeneous cores. We have developed a runtime framework for distributing the workload, which includes several locking-free and relaunch-free scheduling schemes. Particularly, the device-level token passing scheme we have developed has very low scheduling overheads.

We have implemented and evaluated our framework on both a coupled CPU-GPU architecture and a decoupled CPU-GPU architecture. We show significant performance improvements from the use of both CPU and GPU cores, over using only a single device. We have also inspected the scalability of our scheduling schemes over CPU plus multiple GPU configurations. The comparison with StarPU and OmpSs demonstrates that the performance of CPU-GPU versions using our framework is better.
Chapter 5: A Pattern Specification and Optimizations Framework for Accelerating Scientific Computations on Heterogeneous Clusters

The previous chapters focused on frameworks or scheduling policies for single node architectures. Even though the efforts we have made help to exploit the computation power of both CPU and GPU(s) on each node, it is far from enough without utilizing the distributed computing resource in cluster environments. This chapter introduces a programming model for easing the programming of scientific applications with different communication patterns on a heterogeneous cluster. Our idea is to provide pattern-specific APIs and optimizations. The pattern-specific optimizations are applied at different levels including CPU/GPU execution, inter-device workload scheduling, and inter-node communication. We aim to improve the programming productivity, as well as to maintain high performance with efficient runtime implementation.

5.1 Programming Model

Our programming model is designed with the following considerations. First, we want a high-level API that can capture most of scientific applications. Second, we want an API from which efficient execution can be achieved on accelerators, while also effectively partitioning the work across CPU and GPU cores, as well as across nodes, and moreover,
without involving restructuring compiler support. The last consideration is important for us because of the need for creating a robust implementation with a moderate effort.

Achieving the above goals while also providing a very general programming system (comparable to MPI or CUDA) does not appear feasible. To address the challenge of creating high-level APIs for scientific computations, we derive inspiration from summarization of scientific computing kernels as *dwarfs* by Phil Collela (and as further generalized in Berkeley’s landscape on parallel computing [36]). Their observation has been that scientific applications follow one of the seven templates or patterns: structured grid or stencil computations, unstructured grids, N-body simulations, generalized reductions, dense linear algebra, sparse linear algebra, and spectral methods (FFTs).

Within this set of templates or patterns, we further observe that standard libraries are available and popularly used for dense and sparse linear algebra and FFTs. Thus, our claim is that a dominant fraction of popular scientific applications developed today using a general purpose parallel model (e.g., MPI or one of the PGAS languages) involves one or more of the following patterns: structured or unstructured grids, generalized reductions, or N-body interactions. Further, N-body problems can be expressed using indirection arrays, just like unstructured grids, which we will show through our work with a mini-application called MiniMD later. Thus, we have focused on three patterns, which can in fact be used for 16 out of 23 Rodinia benchmarks [14]. Moreover, our work can be extended in the future to support additional patterns, and thus a wider class of applications.

### 5.1.1 Communication Patterns and APIs

In this section, we introduce the supported communication patterns and the corresponding APIs, which are summarized in Table 5.1.
User-defined functions for generalized reductions

```c
void (*gr_emit_fp)(Object *obj, void *input, size_t index, void *parameter)
generates (a) key-value pair(s) based on one input unit (starting at index), and
inserts the key-value pair(s) to the reduction object obj

void (*gr_reduce_fp)(VALUE *value1, VALUE *value2)
reduces value2 to value1, the pointer to a value in the reduction object
```

User-defined functions for irregular reductions

```c
void (*ir_edge_compute_fp)(Object *obj, EDGE edge,
void *edge_data, void *node_data, void *parameter)
processes an edge, generates (a) key-value pair(s), and inserts the key-value
pair(s) to the reduction object obj

void (*ir_node_reduce_fp)(VALUE *value1, VALUE *value2)
reduces value2 to value1, the pointer to a value in the reduction object
```

User-defined functions for stencil applications

```c
void (*stencil_fp)(void *input, void *output,
int *offset, int *size, void *parameter)
processes a single element in the input grid and writes the result to output grid
```

Table 5.1: User-defined Functions for Communication Patterns

**Generalized Reductions:** these arise in a variety of data-intensive and scientific applications. They have been supported by existing HPC programming systems, e.g., OpenMP [28] provides a reduction clause for combination of computation results from different threads, although the reduction is only applicable to individual scalar variables (in the case of most compilers).

The first row of Table 5.1 shows the signatures of user-defined functions for generalized reductions - while the API is similar to MapReduce [31, 16, 17], there are some differences also for achieving higher efficiency. The API involves two types of operations: *emit* and *reduce*. The `gr_emit_fp` function is invoked to process the smallest input unit, and generates one or more *key-value* pairs. The first parameter of the `gr_emit_fp` function is a system-defined data structure, referred to as the *reduction object*, which is implemented as
a hash table with support for parallel key-value insertion, and is used to accumulate the reduction result. Parameter index is generated by the runtime system, and captures the start location of the particular input unit. Users could pass extra information (e.g. cluster centers for Kmeans) through parameter. The gr_reduce_fp function indicates the operation conducted (typically commutative and associative) while inserting one key-value pair to the reduction object. This function is also invoked while global combination is being conducted.

```
Real X(numNodes), Y(numEdges); ! node/edge data arrays
Integer IA(numEdges,2); ! indirection array

for (i = 0; i < numEdges; i++) {
   X(IA(i,1)) = X(IA(i,1)) op Y(i);
   X(IA(i,2)) = X(IA(i,2)) op Y(i);
}
```

Figure 5.1: An Irregular Reduction Loop

**Irregular Reductions:** Irregular reductions arise in the context of unstructured grids, and in some of the implementations of N-body problems. Nodes in unstructured grids are connected by edges explicitly, and one can think of the computations in terms of nodes and edges. For example, the input data comprises the node data, the values associated with the edges, and array(s), referred to as the indirection array(s) (edges), which stores the end points (nodes) that are connected by each edge.

The code listed in Figure 5.1 shows a typical irregular reduction loop, denoted computations performed by iterating over all edges. Each iteration updates two elements in array X using commutative and associative operations op, with accesses through the indirection array IA. Because each node may be connected by multiple edges, the update of
the elements in \( X \) involves reductions. Overall, the key distinctive property of irregular reductions are the accesses of the node data using indirection arrays.

We now discuss how a high-level API can be used for irregular reductions. The `ir_edge_compute fp` function is invoked to process an edge, the lowest level of processing granularity. Like generalized reductions, a system-defined reduction object is provided to store the reduction result. Among the parameters of this function, `edge` is the single edge being processed, and includes the edge ID, as well as identifiers of the nodes it connects. `edge_data` stores attributes associated with `edge`. To update a node, the user needs to pass the node ID as the key to the `insert` function of reduction object. Function `ir_node_reduce fp`, a `node reduce function`, works in the same way as the `reduce` function in generalized reductions, i.e., it is invoked to update the reduction result of a node in the reduction object.

The API we have developed is simplified (and operates correctly) only with a certain types of partitioning of the unstructured grid [50]. We refer to the edges as the `computation space` and the nodes (being updated) as the `reduction space`. The `inter-process workload partitioning` strategy we applied is based on the reduction space, which involves the following steps: 1) Divide the nodes into equal partitions, 2) Group the edges: if both nodes of an edge belong to the same partition, this edge is exclusively assigned to this partition; if an edge is connecting nodes belonging to different partitions (say partitions 0 and 1), then this edge is assigned to both processes. To avoid race conditions, when an edge is being processed, only the node(s) belonging to the current partition is updated. This partitioning method allows each process to update its own set of nodes, disjoint from other nodes, which also improves the node access locality. Note that the above partitioning scheme can be (and is) applied at multiple levels, across nodes, across devices, and even within a device to improve locality.
Stencil Computations: Stencil applications involve computations of updating each element in the input based on the values of its neighbor elements. The input for stencil applications is usually a structured grid or matrix with two or higher dimensions.

The standard function we support is `stencil_fp`. It defines a stencil operation that is applied to a single element in the grid. The API also provides `get functions` like `GET_FLOAT3` and `GET_INT2`, which are pre-defined macro functions for referencing elements from input or output with a certain number of dimensions: the arguments to these functions include the data buffer to be accessed and the coordinate (offset) of the element to be accessed. The return value is the reference to that element. Parameter size is implicitly passed to the `get functions` for the runtime to compute the absolute position of an element in the input/output buffer.

5.1.2 Case Study: A Comprehensive Example Involving Multiple Communication Patterns

We now use an example to show how our programming model is used to parallelize an application that involves different communication patterns. The application we use here is Molecular Dynamics (Moldyn) [51], which simulates the interaction and motion of molecules in a period of time. It involves an irregular reduction kernel for computing the forces among molecules (CF) based on the interactions (edges) among the nodes. In addition, two kernels computing the kinetic energy (KE) and average velocity (AV), respectively, are generalized reductions.

The first listing below shows the user-defined functions. `force_cmpt` and `force_reduce` are the functions for the irregular reduction kernel CF. Function `force_cmpt` is an edge compute function. It is invoked by the irregular reduction runtime for CF kernel to compute the distance and force between the two nodes connected by an edge. If the distance of the
two nodes is within a threshold, it uses the node ID as the key, and the force as the value, to update the reduction result of each node. *node reduce function force_reduce* indicates that the reduction operation applied to the reduction result is simply accumulation. Similarly, functions `ke_emit` and `ke_reduce` are the user-defined functions for the kernel `KE`, and `av_emit` and `av_reduce` are for the kernel `AV`. Details of these four functions are not shown to keep the code listing short.

The second listing shows how the system API should be invoked to create a runtime environment and runtime instances for the kernels. Line 2 creates a *runtime environment*, which is then used to create runtime instances for different communication patterns. In this example, Line 5 and Line 7 create two runtime instances, for irregular reductions and generalized reductions, respectively. Line 9 to Line 16 is the code for the irregular reduction kernel `CF`, and Line 17 to Line 27 is the code for the two generalized reduction kernels `KE` and `AV`. If multiple kernels follow the same communication pattern in an application, the same runtime can be reused, by resetting configuration parameters (e.g., the user-defined function names and input data). In this example, generalized reduction runtime `gr` is used by both `KE` kernel and `AV` kernel. Alternatively, users could also define separate runtime instances for `KE` and `AV`.

### 5.1.3 Discussion

Our framework has certain limitations, both in terms of the patterns it can handle and how these patterns are specified. The key thing to note, however, is that use of our framework does not prevent invocation of other libraries since our framework is simply based on C/C++. Thus, for applications that have certain other patterns in addition to the ones
**Moldyn: User-defined Functions**

```c
//user-defined functions for CF kernel
DEVICE void force_cmpt(Object *obj, EDGE edge,
void *edge_data, void *node_data, void *parameter){
/*compute the distance between nodes...*/
if(dist < cutoff){
double f = compute_force();
obj->insert(&edge[0],&f);
f = -f;
obj->insert(&edge[1],&f);
}
}

DEVICE void force_reduce(VALUE *value1, VALUE *value2){
*value1 += *value2;
}

//user-defined functions for KE kernel
DEVICE void ke_emit(...){...}
DEVICE void ke_reduce(...){...}

//user-defined functions for AV kernel
DEVICE void av_emit(...){...}
DEVICE void av_reduce(...){...}
```

**Moldyn: Invoking System API**

```c
MPI_Init(...);
Runtime_env env;
env.init();
//runtime for CF
IReduction_runtime *ir = env.get_IR();
//runtime for KE & AV
GReduction_runtime *gr = env.get_GR();
//Compute Force (CF) kernel
ir->set_edge_comp_func(force_cmpt);//use force_cmpt
ir->set_node_reduc_func(force_reduce);//use force_reduce
/*(set edges, nodes...)*/
for(int i = 0; i < n_tsteps; i++){
ir->start();
result = ir->get_output();
/*update node positions and velocities...*/
}
/*(prepare input, generate splitting info...)*/
//Kinetic Energy (KE) kernel
gr->set_emit_func(ke_emit);//ke_emit as emit func
gr->set_reduc_func(ke_reduce);//ke_reduce as reduce func
...
gr->start();
double ke_output=(gr->get_output()).aggregate();
//Average Velocity (AV) kernel
gr->set_emit_func(av_emit);//av_emit as emit func
gr->set_reduc_func(av_reduce);//av_reduce as reduce func
...
env.finalize();
MPI_Finalize();
```

we support, we can still accelerate code development while allowing users to manually program other sections or use other libraries.
Some of the other limitations are as follows. Although multiple kernels of a same communication pattern in an application can be supported by our framework, only a single target object can be processed every time a runtime instance is launched. For example, only one grid can be processed in a particular stencil runtime execution, and similarly, only one set of edges can be processed in every irregular runtime execution. One requirement we impose is that function headers should begin with a system-defined macro, which is converted to device specific qualifiers at compile time. Our future work will address these limitations.

5.2 Framework Implementation and Pattern Specific Optimizations

Our framework is designed for clusters with heterogeneous CPU-GPU nodes. We also allow each node to have multiple GPUs.

5.2.1 Code Organization and Compilation

![Figure 5.2: Code-level Structure of the Framework](image-url)
We describe the organization of our framework using Figure 5.2. The application logic itself is specified by writing implementation of functions. Although we can execute an application utilizing both CPU and GPU cores, the user needs to write only one set of sequential functions processing a smallest input unit. The source code of the user-defined functions is invoked by both CPU and GPU kernels. Because function pointers on one device could not be used by other devices, the user code is copied in both CPU GPU namespaces. Besides the two kernels, what are compiled together also include both inter-node and intra-node communication libraries, along with the optimizations written specifically for each communication pattern. The rest of this section describes the implementation of these.

5.2.2 Runtime Execution Flow

We create one MPI process for each node, and intra-node parallelism is exploited through CPU multi-threading (pthread) and GPU co-processing. Inter-process communication and GPU kernel launches are handled by the CPU threads. Compared with one process per core and one process per GPU model, our approach improves the inter-core data access locality, reduces memory requirements, and avoids unnecessary costs of inter-process (but intra-node) communication or memory copies.

Inter-process workload partitioning is conducted in a distributed manner: each process fetches only the partition that belongs to itself. The input loading might involve extra processing. In irregular reductions, besides loading the local node partition, each process also inspects all the input edges and pick the ones that connect local nodes, to form the local computation space. Stencil computations need to allocate extra space for each sub-grid, as
halo regions are needed for border processing. At the end of the processing, a global reduction is conducted for generalized reductions. The global reduction is performed among the processes in a parallel binary tree order, so that up to $\log(n)$ parallel reduction steps are needed to form a final result, where $n$ is the total number of processes. Whereas for irregular reductions, since the nodes from the input are partitioned into independent spaces, there is no need to do a global reduction. Reduction results from different processes are written back to disk as separate parts. Similarly, for stencil computations, results of the sub-grids are also directly written back to disk in a distributed manner.

### 5.2.3 Distributed Memory Execution

Executing an application across nodes involves challenges of inter-process data partitioning and performing data exchange (communication) during the execution.

**Workload Partitioning and Inter-process Communication for Generalized Reductions:** Because there is no dependence among loops of generalized reductions, the input data is evenly partitioned among processes. The only inter-process data exchange is the final combination of the reduction objects from different processes.

![Diagram](image.png)

**Figure 5.3:** Arrangement of Nodes for Irregular Reductions. Solid lines: local edges, Dashed lines: cross edges
Inter-process Data Exchange for Irregular Reductions: The inter-process data partitioning method we use has already been introduced. Data exchange is only needed before execution of each reduction loop. It is handled as follows. At the time of loading input, the nodes are equally divided and each process fetches its own partition. Each process also inspects the edge input, and associate corresponding edges to its own partition, as is stated in Section 5.1.1. Cross edges in one partition need to access nodes from remote processes. We call these nodes remote nodes. After receiving its edge partition, each process inspects the cross edges, and groups the remote node IDs. Figure 5.3 shows the arrangement of local nodes and remote nodes. The local nodes are stored continuously in the front. Remote nodes from the same process are placed continuously. Initially, the two nodes of each edge have IDs with global node rank. The runtime on each process converts these IDs into the local rank. A global ID array is maintained for node data exchange. Every time node data is updated (reset) by the user program, data exchange should be conducted. The following steps show what a process does during the node data exchange: 1) Send a request to each remote process - The request message contains the number of nodes it is requesting. 2) Receive the requested number of nodes from each remote process and allocate receive buffers for requested node IDs. Allocate send buffers for node data requested by each remote process. 3) Send global remote node IDs (shown in Figure 5.3) to corresponding processes. 4) Receive requested node IDs from each remote process to the corresponding receive buffers. 5) Copy corresponding node data into the node data send buffer for each remote process and send each node data buffer to remote process, and 6) Receive remote node data from each remote process and store them in the corresponding node data slot as shown in Figure 5.3.
Steps 1 through 4 are only performed at the start of the kernel and after the edges (connections) are reset. Otherwise, the remote node ID information is not changed and only data should be exchanged (Steps 5 and 6). One of the key features of our implementation is support for *overlapped execution*. We compute local edges and cross edges separately. The former only depends on local nodes and we let its execution run concurrently with the exchange of remote node data (Steps 5 and 6).

**Grid Partitioning and Data Exchange for Stencil Computations:** Our runtime system expects users to provide certain important parameters, such as the dimensionality and size of the global grid and the virtual processor Cartesian topology on which the grid is going to be decomposed. The stencil runtime system divides the global grid equally according to these parameters, and distributes the sub-grids to corresponding processes.

![diagram](image)

**Figure 5.4:** Time Sequence of the Stencil Runtime for Heterogeneous Executions

Another important issue to be handled by the stencil runtime system is the data exchange. For convenience, data exchange at both process and device levels are discussed together here. Figure 5.4 shows two device grids inside a process. For simplicity, we use
2D grids to show how data exchanges and computations are conducted. Data exchanges among neighbor processes are conducted on each dimension, but here we only show the dimension with non-contiguous boundary, which is the most complex case.

**Steps for Inter-process Data Exchange:** Step 1 copies non-contiguous borders to contiguous buffers, which are in turn copied to the process’s send buffer. For the CPU, the non-contiguous boundary elements can be copied to the send buffer in strips using `memcpy`. However, for a GPU, non-contiguous device memory is not able to be copied out directly using `cudaMemcpy`. We allocate a host mapped (zero-copy) memory buffer, and launch a kernel to copy such a boundary into the buffer. The buffer is then directly copied into the process’s send buffer (Step 2). Similar copy procedures are conducted on other non-unit-stride access boundaries of each grid. When the send buffers for all boundaries are ready, the runtime launches asynchronous MPI send and receive for the boundary exchange. At the same time, the computation of inner elements is launched, which runs *concurrently* with the inter-process data exchange. When the data exchange and inner computation finish, contents in the receive buffer are copied to each sub-grid’s *halo* area. For GPU, data is first copied to host mapped buffer (Step 4), after which, a GPU kernel is launched to copy the buffer into the halo area on the device (Step 5).

**Inter-device Data Exchange:** After the inter-process data exchange is finished, boundaries between neighbor devices are exchanged (Step 6). Exchanges between CPU and GPU use `cudaMemcpy`. For GPU-GPU exchanges, an efficient device-to-device asynchronous memory copy, `cudaMemcpyPeerAsync` is used, which supports concurrent bi-directional transfer on the PCE-e bus. Finally, the boundary elements are processed (Step 7).
5.2.4 CPU-GPU Workload Partitioning

Due to the heterogeneity of cores within each node, we cannot evenly partition the workload across them. The processing speed of each device not only depends on its own hardware configurations, but also the properties of the application. We apply different intra-process partitioning strategies according to the communication patterns, to minimize load imbalance while keeping the overhead of scheduling small.

**Dynamic Scheduling and Overlapped Execution for Generalized Reductions:** Because generalized reductions typically have only one or a small number of time steps, input data copied to the GPU is typically used only once. The data copy, if not overlapped with the computation, could significantly influence the overall performance. Also, to avoid load imbalance, dynamic workload scheduling is necessary. Thus, we schedule tasks on devices in chunks, and devices obtain chunks by pthread locking. Each CPU core continuously receives chunks to process. The task retrieval and kernel launches of GPUs is controlled by a CPU thread and two streams are created for each GPU. Each time, the controlling CPU thread retrieves a task chunk for each GPU, and splits the chunk into two smaller blocks allocated in pinned memory. Sequentially for each stream, asynchronous memory copy of input from pinned memory to device memory is then launched, followed by the kernel launch. When all the GPUs finish processing the task blocks in both the streams, the controlling thread gets another task chunk.

**Adaptive Workload Partitioning for Irregular Reductions and Stencil Computations:** Unlike generalized reductions, irregular reductions and stencil computations usually involve many time steps. Thus, we can profile the relative processing speed of each device (specific to the particular application) and utilize the profiling information for static workload partitioning.
Partitioning for Irregular Reductions: After edge and node partitions are organized on each process, the runtime conducts inter-device partitioning among the devices. Irregular reductions involve a large number of iterations, and edges (connectivity) can be unchanged for many time steps. Thus, if dynamic scheduling is used in each time step, data chunks have to be reloaded from the host by GPUs repetitively, leading to high I/O overhead. In view of this, we utilize an adaptive partitioning. In the first time step, the reduction space is evenly partitioned. This creates nearly equal-sized edge partitions for each device. We profile the processing speed of each device, and adjust the splitting ratio: let $S_i$ be the speed of device $i$, and $N$ be the total number of nodes, then this device will be assigned $N \times \frac{S_i}{\sum_{k=1}^{ndev} S_k}$ number of nodes. The workload is re-partitioned in the second time step according to the new ratio. The newly grouped edges are copied to device memory on each GPU, which would be kept unchanged for a number of iterations until the edges are rebuilt. In addition, the node data has a full copy on each device. Although every time nodes are updated, they need to be updated to each device, the size of node data is much smaller compared with edges. Maintaining a full copy of node data also avoids inter-device data exchange.

Partitioning for Stencil Computations: The sub-grid partitioned to a process is further partitioned along the highest dimension, and a sub grid is assigned to a multi-core CPU or a many-core GPU. Similar with irregular reductions, the partitioning is conducted adaptively: the first time step utilizes even partitioning, and determines the relative speed of each device. The grid is re-partitioned in the second iteration based on the speeds of devices.
5.2.5 GPU and CPU Execution

We now focus on the problem of efficient execution on each device. The performance of an application on a GPU is typically bottle-necked by the memory accesses. The workload partitioning and execution strategy for each communication pattern in our system extensively utilizes the GPU’s memory hierarchy, with the knowledge of specific properties of each pattern. Some of the same challenges also arise for execution on CPU cores. We now describe the key optimizations performed in our system.

Reduction Localization for Generalized Reductions and Irregular Reductions: For both generalized reductions and irregular reductions, to avoid race conditions, parallel updating of the reduction objects is ensured by locking (implemented as atomic operations). If the number of distinct keys is small, the contention among the threads for updating the values associated with one key can be large, leading to significant waiting times.

For generalized reductions, the runtime system allows for users to configure the size of the reduction objects. If reduction objects are small enough, the runtime system stores them in the shared memory on each SM, which supports much faster synchronization and I/O. Local reductions are first performed in shared memory on each SM, and the reduction results from all SMs are merged to the reduction object in the device memory to form the final GPU reduction result. The runtime system creates multiple reduction objects in the shared memory if the size of each object is small enough, and each reduction object is updated by a subset of the threads in one thread block, which further reduces the contention overhead. This technique is also applied at the CPU side: each CPU core has a private reduction object, which is used for local reduction. A final combination is performed when the computation ends.
For irregular reductions, the reduction space based input partitioning scheme we discussed earlier enables the utilization of the shared memory on GPUs. Edges and nodes assigned to each device are further partitioned. The partitioning module determines the number of partitions based on the size of GPU’s shared memory and the size of each reduction element, so that each partition of the reduction space can be put in the shared memory:

\[ \text{num_parts} = \text{num_nodes} / \left( \frac{\text{shared_memory_size}}{\text{reduction_element_size}} \right). \]

At CPU side, workload is partitioned in a similar way, and CPU cores process the workload in partitions. Such a partitioning greatly improves the data locality. The reduction results from different SMs/CPU cores are simply concatenated to form the global reduction result. There is no combination involved because the reduction space was split into independent partitions.

**Grid Tiling for Stencil Computations:** The part of the grid that each device processes is tiled along every dimension. Each CPU core or a GPU SM processes grid elements in tiles. The inner element tiles and boundary element tiles are grouped separately. Because the computation of inner elements only depends on local data, inner tiles are processed concurrently with inter-process data exchange. The boundary tiles are processed after the data exchange finishes. Grid tiling is beneficial for two reasons: 1) it improves the cache performance for the neighbor element access. 2) the boundary plane tiles are grouped together, which enables a single kernel launch on GPUs to process all the boundary planes, without launching separate kernels for each plane.

The default configuration of the on-chip memory on Fermi GPUs is configured as 48KB shared memory and 16KB L1 cache, which is used for generalized reductions and irregular reductions. The stencil runtime changes the configuration into 48KB L1 cache with 16KB shared memory by setting `cudaFuncCachePreferL1` flag, which benefits the cache
Figure 5.5: Intra-node and Inter-node Scalability and Comparison Against Hand-written MPI Applications

performance for stencil computations. We are currently using fixed tile width, 16 and 32 for CPU and GPU, respectively.
5.3 Experimental Results

Our target platform is a cluster with discrete CPU-GPU nodes. For evaluating our system on such a target architecture, we used a CPU-GPU cluster, which consists of 32 nodes, with each node having a 12 core Intel Xeon 5650 CPU and 2 NVIDIA M2070 GPUs (thus, 64 GPUs in all). Each node has a system memory of 47 GB, and each GPU has a device memory of 6 GB. We used mpicxx for MVAPICH2 version 1.7, and nvcc 5.5.0 to compile all the codes, with -O3 optimization enabled.

We had the following goals in our experiments. 1) Evaluate the overall effectiveness of the framework, particularly, assessing how it can help scale applications across nodes in a cluster, by use of CPU and GPU cores, and by even using multiple GPUs within a node, 2) Compare the performance of the applications using the framework against existing hand-written benchmarks, including both MPI applications (for distributed memory performance and scalability) and CUDA applications (for GPU performance), and 3) Quantify the benefits from optimizations supported in the system. The execution times we report here include workload distribution time and computation time, i.e., the application setup and initialization times are not included.

5.3.1 Applications Used

Five applications of varying complexity and involving one or more patterns were chosen for our evaluation study.

Kmeans: Kmeans is a simple data mining application involving generalized reductions. We used a three-dimensional dataset with 40 centers (200 Million points - 2.3 GB). We
report the execution time with execution of a single iteration. For comparison with an existing MPI implementation, a widely distributed kernel was taken.

**Moldyn:** This application simulates interactions among molecules in a three-dimensional space. The primary pattern here is irregular reduction, but it involves generalized reductions as well. A dataset with 1 million nodes and 130 million edges was used and the application was executed for 1000 iterations. Because we did not find a comparable hand-written MPI application, we only focus on performance with our framework.

**MiniMD:** MiniMD is a mini-app developed by the Mantevo project at Sandia National Labs. The goal of the Mantevo project is to create representative applications (a few thousand lines of code) of full-scale scientific applications. Particularly, MiniMD is a smaller version of a large-scale software LAMMPS. MiniMD simulates the motion of a large number of atoms (or molecules) by computing the position and velocity of each atom in the system throughout a predefined number of time-steps. Though it follows a different algorithm than the Moldyn application, its major kernel, force computation, is also an irregular reduction kernel. It also includes a number of generalized reduction kernels, such as those for energy computation. The simulation was conducted for 1000 iterations in our experiments and 500,000 atoms (double precision) were simulated. The MPI implementation we used is the one developed by the Mantevo project (the code also uses OpenMP for shared memory parallelization).

**Sobel:** Sobel edge detection is an application used in image processing. Two $3 \times 3$ masks are convolved to the input matrix with two-dimensional points. Thus, the application is a 9-point stencil application. A $32,768 \times 32,768$ floating point matrix was used as the dataset. MPI benchmark used for comparison was obtained from the *UPC Benchmarking Suite*,

\[1\text{http://users.eecs.northwestern.edu/~wkliao/Kmeans/index.html}\]
developed at the George Washington University\textsuperscript{2}. Edge detection was conducted for 15 iterations.

**Heat3D**: This code simulates heat transmission in a 3D space, involving a 7-point stencil. An existing MPI implementation is widely distributed and used\textsuperscript{3}. A $512 \times 512 \times 512$ grid with double precision elements was used, and results reported are from 100 iterations.

### 5.3.2 Code Sizes

![Comparison of Code Sizes between Handwritten (MPI) Benchmarks and Versions Using our Framework](image)

Figure 5.6: Comparison of Code Sizes between Handwritten (MPI) Benchmarks and Versions Using our Framework

As shown in Figure 5.6, we compared the code sizes between implementations as part of our framework and available MPI programs. The ratios are 0.53, 0.37, 0.40, and 0.28 for Kmeans, MiniMD, Sobel, and Heat 3D, respectively. In other words, over these four applications, we have an average of 40\% of the code size of MPI implementations. A more detailed examination shows that our framework is reducing the coding effort spent on workload partitioning and communication. The reduction in code size varies as the code ratio for initialization and resetting are different. It is also important to emphasize

\textsuperscript{2}http://upc.gwu.edu/download.html

\textsuperscript{3}http://dournac.org/info/parallel_heat3d
that unlike MPI implementations, the code written with our framework can also utilize accelerators like GPUs.

5.3.3 Overall Performance from the Framework

Our first set of experiments focuses on how our framework scales while we increase devices on each process and while we increase the number of processing nodes. A comparison of distributed memory scalability against existing MPI implementations was also performed, using four applications.

Figure 5.5 shows the results of scaling our framework on a heterogeneous cluster with 5 applications. In all cases, CPU-only execution uses all 12 cores on each node, i.e., we use between 12 and 384 cores for our experiments. The speedups shown in the figure are against the executions on a single core of the CPU.

We first focus on the benefit of utilizing GPUs together with the CPU cores. For K-means, the GPU is 2.69 times faster than 12-core CPU. After combining the computing power of 12 CPU cores with one GPU, a speedup of 1.20 is achieved over 1-GPU execution, or a speedup of 3.23 over 12-core CPU execution. Combining the second GPU also results in a 1.92x speedup over the 1-GPU execution, or a speedup of 5.16 over 12-core CPU execution. Similar inter-device scalability could be seen from the other four applications, although the CPU-GPU execution time ratios are different for different applications. For Moldyn, the GPU is 1.50x faster than the CPU. CPU + 1GPU outperforms GPU-only by 1.54x, and CPU + 2GPU brings a 1.64x incremental speedup. For MiniMD, GPU is 1.70 times faster than CPU, and the CPU + 2GPU achieves an average speedup of 3.89x over using CPU only. For Heat3D, GPU is 2.4x faster than CPU, and the average speedup
of using CPU together with two GPUs over using a single CPU on each node is 5.5x. Similarly, for Sobel, CPU + 2GPU is 4.68x faster than CPU-only. The performance of stencil applications Sobel and Heat3D varies for different decompositions, and we report the best execution results here.

The overall speedups from utilizing 32 nodes and all the cores on each node vary between 562 and 1760 for different applications. Kmeans has the maximum overall speedup, as the GPUs are much faster than the CPU due to the effective usage of shared memory for generalized reductions, and there is a negligible communication overhead. The other four applications achieve a lower speedup of near or above 600, with Sobel near 1000, because of limited speedup from GPUs and various overheads.

To quantify the scheduling (including the possibility of load imbalance), synchronization, and communication overheads among devices, we can calculate the perfect speedups of CPU+1GPU and CPU+2GPU executions over the CPU-only execution. This calculation uses the relative performance of each device, and ignores any overheads. In other words, we assume that work can be perfectly distributed with no imbalance, scheduling costs, synchronization, or communication. We compare the actual execution speedups with the perfect speedups in Table 5.2. We can see that in most cases, the actual speedups are close to the perfect speedups. On average, we are achieving 89% of the perfect performance for CPU+1GPU executions and 88% for CPU+2GPU executions, indicating that different computing units are being effectively used by our framework.

We next focus on distributed memory scalability of our implementation and compare it against that of hand-written MPI applications (for four of the five applications). For a fair comparison, the following discuss focuses only on the CPU-only execution results from our framework. For Kmeans, Sobel, and Heat 3D, the hand-written MPI applications
Table 5.2: Comparison of Intra-node Scalability: Perfect (Assuming No Scheduling Overheads) and Actual

<table>
<thead>
<tr>
<th></th>
<th>Kmeans</th>
<th>Moldyn</th>
<th>MiniMD</th>
<th>Sobel</th>
<th>Heat3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU+1GPU</td>
<td>Perfect</td>
<td>3.69</td>
<td>2.5</td>
<td>2.7</td>
<td>3.24</td>
</tr>
<tr>
<td></td>
<td>Actual</td>
<td>3.23</td>
<td>2.31</td>
<td>2.15</td>
<td>2.94</td>
</tr>
<tr>
<td>CPU+2GPU</td>
<td>Perfect</td>
<td>6.38</td>
<td>4</td>
<td>4.4</td>
<td>5.48</td>
</tr>
<tr>
<td></td>
<td>Actual</td>
<td>5.16</td>
<td>3.79</td>
<td>3.89</td>
<td>4.68</td>
</tr>
</tbody>
</table>

Involves one MPI process per CPU core. In the case of Heat3D, framework achieves an average speedup of 8% over the corresponding MPI code, whereas for Sobel, there is a 11% slowdown. The reason for the slowdown with Sobel is that our stencil runtime system spends extra cycles on computing the offsets while processing the elements. Optimizations such as tiling and overlapping communication with computation hide these overheads to an extent. For Kmeans, our framework is 1.05x faster, on average, because of the use of light-weight threads instead of separate MPI processes. For MiniMD, we are 1.17x faster, which is because of overlapping communication and computation. Overall, within our framework, the speedup in going from 12 cores (1 node) to 384 cores (32 nodes) ranges between 20 and 26.

It should be noted that some of the optimizations that are helping our framework can certainly be applied on MPI (though the handwritten versions we found did not use them). However, such optimizations within MPI will be at the cost of additional programmer effort, whereas our framework applies them automatically. Again, it is important to emphasize that our framework can also benefit from accelerators.
5.3.4 Benefits from Optimizations

The design of our APIs enables a number of pattern specific optimizations. We now focus on showing the effect of overlapped execution of irregular reductions and stencil computations (for distributed memory execution), as well as the benefit of applying tiling to stencil computations.

Figure 5.7 shows the effect of tiling (for stencil computations) and overlapped execution (for both stencils and irregular reductions) with different number of nodes (one application for each pattern). The execution on each node uses the 12 available CPU cores and the 2 GPUs. The tiling optimization increases the performance of Sobel by up to 20%. Overlapping the asynchronous communication with computation benefits the overall performance significantly. On average, overlapped execution is 37% and 11% faster than non-overlapped execution for Moldyn and Sobel, respectively.

5.3.5 Comparison with GPU Benchmarks

To the best of our knowledge, no hand-written codes are available that can use both CPU and GPU cores. Thus, to examine the efficiency of our framework, we compare
the code executed by our framework against the hand-written CUDA codes, with a single Fermi GPU on a single node. Among the five applications we have used, we could obtain comparable hand-written versions for Kmeans and Sobel only. The Kmeans benchmark was obtained from the Rodinia benchmark suite [14]. 10 Million points were used for the test. The Sobel code was distributed with the NVIDIA SDK. We use an $8,192 \times 8,192$ image as the input.

The results are shown in Figure 5.8. Kmeans using our framework is only 6% slower than the hand-written benchmark. The CUDA version Sobel from CUDA SDK is optimized to use texture memory for staging the input. Our framework cannot perform such application-specific optimizations, and results in 15% slower execution.

5.4 Summary

We have focused on the problem of developing a high-level programming model for scientific applications which can exploit parallelism at multiple levels (across nodes in a cluster, across CPU cores, on GPUs, and across CPU and GPUs). This has been achieved by focusing on specific communication patterns that commonly arise in scientific applications,
and developing high-level APIs specific to each. Our extensive runtime system involves pattern specific workload partitioning (at different levels) and optimizations for dealing with communication overheads and CPU and GPU specific features.

We have evaluated the scalability of our framework with five different applications, each involving one or more of the patterns. Our framework achieves high inter-node and inter-device parallelism. At the same time, it does not lose performance compared to handwritten codes that focus on a single level of parallelism (i.e. MPI codes or CUDA). On a 32 node GPU code, speedups between 600 and 1800 are obtained.
Chapter 6: Efficient and Simplified Parallel Graph Processing over CPU and MIC

Intel Xeon Phi (MIC architecture) is a relatively new accelerator chip, which combines large-scale shared memory parallelism with wide SIMD lanes. Mapping applications on a node with such an architecture to achieve high parallel efficiency is a major challenge. In this paper, we focus on developing a system for heterogeneous graph processing, which is able to utilize both a many-core Xeon Phi and a multi-core CPU on one node. We propose a simple programming API with an intuitive interface for expressing SIMD parallelism, as well as an efficient runtime with a series of optimizations.

6.1 Programming API

We base the programming API of our framework on the BSP model [110]. We treat a graph application as an iterative process, and each iteration is divided into three important steps: message generation, message processing and vertex updating, with synchronization between these steps. To use an example – Single Source Shortest Paths Problem (SSSP) aims to find a shortest path between a single source vertex and every other vertex in the graph. The algorithm we introduce here is applied to a positive weighted directed graph. The brief steps of the SSSP algorithm are: 1) initialization: for each vertex, an attribute distance representing the possible minimum distance from the source is maintained and
initialized to a large constant, and the distance value for the source vertex is initialized
to 0. 2) relaxation: for every vertex \( u \), and for every incoming edge \( e = (v, u) \), if
\( v \text{.distance} + e \text{.weight} < u \text{.distance} \), then update \( u \text{.distance} \) to \( v \text{.distance} + e \text{.weight} \),
and 3) the above two steps are repeated until no more updates occur.

This algorithm can be easily expressed by a programming model with an API for mes-
sage generation, message processing, and vertex updating. After the initialization step, the
relaxation step could be expressed as a combination of these three steps. In each itera-
tion, each vertex receives messages from its incoming neighbors, which are the updated
minimum distances from the source. A minimum value of the values contained in these
messages can then be calculated in (message processing). If this vertex has a new mini-
mum potential distance value as a result of this calculation, it updates its distance with this
new minimum value (vertex updating) and propagates this update through messages to its
neighbors along the outgoing edges (message generation). The same procedure is repeated
in the future iterations. Similar steps are used in Pregel [75], a distributed memory graph
programming model, but there are also key differences considering the focus on shared
memory and SIMD processing.

We explain our API using user-defined functions for SSSP. In addition to three main
functions that are supported, an important characteristic of our API is simplified support
for exploiting SIMD parallelism. This will be explained at the end of this section.

User-defined functions for SSSP

```cpp
User-defined function for message generation:

```void generate_messages(size_t vertex_id,
graph<VertexValue, EdgeValue> *g) {
    float my_dist = g->vertex_value[vertex_id];
    // Graph is in CSR format.
    for (size_t i = g->vertices[vertex_id];
        i < g->vertices[vertex_id + 1]; i++) {
        send_messages<MessageValue>(g->edges[i],
        my_dist + g->edge_value[i]);
```
User-defined function for message processing:

```cpp
template <class MessageValue>
void process_messages(vmsg_array<MessageValue> &vmsgs) {
  // Reduce the vector messages to vmsgs[0].
  vfloat res = vmsgs[0];
  for (int i = 1; i < vmsgs.size(); ++i) {
    res = min(res, vmsgs[i]);
  }
  vmsgs[0] = res;
}
```

User-defined function for vertex updating:

```cpp
template <class VertexValue, class EdgeValue, class MessageValue>
void update_vertex(MessageValue &msg, graph<VertexValue, EdgeValue> *g, size_t vertex_id) {
  // Distance changed, will send msgs.
  if (msg < g->vertex_value[vertex_id]) {
    g->vertex_value[vertex_id] = msg;
    g->active[vertex_id] = 1;
  } else {
    // Distance not changed, no msgs will be sent.
    g->active[vertex_id] = 0;
  }
}
```

Basic API: The user-defined functions for SSSP is shown above. The three key functions are as follows. `generate_messages()` defines the way a certain vertex generates messages. The runtime system invokes this function for a vertex only when it is active (only the source vertex is active in the initial iteration). For an active vertex in this example, the function sends a message to every vertex it connects to, and specifically, it propagates its minimum distance from the source plus the weight of each edge.

`process_messages()` is invoked for a vertex (or simultaneously for multiple vertices) to process the received messages coming from other neighbors: in this particular example, it calculates the minimum value among all the messages that are received.

The last of the three functions, `update_vertex()` is invoked to update the value or the status of a particular vertex, typically as a result of processing of received message(s). Specifically, the parameter `msg` of function `update_vertex()` is the processing result from function `process_messages()` for the messages received by this vertex. In the example in the
code example, the function compares the minimum distances received in the messages with its own distance from the source. If the received minimum distance is shorter, it updates its own value to the smaller value, and at the same time, it sets the status of vertex to active. Otherwise, this vertex will be inactive, since it is not updated in the current iteration. An inactive vertex may not participate in the message generation for next step.

**Portable API for Exploiting SIMD Parallelism:** Our framework provides a set of SSE primitives for automatic SIMD processing within each thread, which help to utilize the wide SIMD lanes. The core units of our SSE primitives are the set of vector types (vtypes). The vtypes we currently support are **vint**, **vfloat** and **vdouble**. In contrast to scalar data types, each vtype contains a group of contiguous data elements. A set of common arithmetic/logical as well as assignment operations are overloaded to these vtypes, so that operations involving vector types, or vector type and scalar types could be easily supported without requiring users to perform complicated SSE intrinsics programming. These overloaded functions perform vectorization by invoking the built-in SIMD intrinsics. These functions are portable between MIC and CPU, that is, the same APIs are built on top of both KNC (for MIC), and SSE4.2 (for CPU), wrapping corresponding architecture-specific intrinsics. As we will explain later, our runtime system organizes received messages for each vertex in an aligned manner within a buffer, which makes SIMD processing of the messages possible.

Referring to the code example of SSSP, since the message type is **float**, the runtime passes the aligned messages as floating point vector arrays into the user-defined function. Users just need to use **vfloat** variables to process the messages in the SIMD fashion, in which case messages for up to 16 vertices (for MIC) are processed simultaneously. As we can see, the code is very similar to the serial code. It should be noted that SIMD
processing of messages only applies to messages with basic data types that are supported by SSE, such as \texttt{int}, \texttt{float} and \texttt{double}, and are limited to associative and commutative reductions, such as sum, max, or min. However, such operations are very common in most graph applications. In the SSSP example, the user-defined function computes the minimum values among the received messages for up to 16 vertices (for MIC), and stores the results as the first element of the vector array \texttt{vmsgs}.

6.2 Framework Implementation

This section describes the detailed design strategies, including data structures, algorithms, and optimizations that are applied.

The system has been designed to address the key features of the MIC architecture and the nature of the graph applications. As the number of edges connecting a node can vary tremendously, we have difficulty in obtaining load balance and avoiding contention for memory accesses (both of these challenges accentuated due to the large number of cores sharing the same memory). Exploiting SIMD parallelism and dividing the work among the MIC and the CPU are other challenges.

6.2.1 System Components and Workflow

We first introduce the components and the workflow of our system, including showing how a graph application is constructed and processed using our framework.

The overall structure of an application built using our system is shown in Figure 6.1. Besides writing user-defined functions, which have been introduced in the previous section, users must write a driver code to read the input (with the help of distributed graph loading API), and to help drive the parameters such as the maximum number of iterations. The input to the system consists of two files: the graph file stored in an adjacency list format,
and a graph partitioning file indicating which device each vertex belongs to. A separate module for generating the graph partitioning file will be described later in Section 6.2.5. The user-defined functions, in turn, are invoked by the runtime system. Symmetric runtime instances on the CPU and the Xeon Phi share the same source code and thus the same structure, though parameters such as numbers of threads running on each device are separately configured. The system is built using MPI symmetric computing, with CPU being Rank 0, and MIC being Rank 1. Multi-threading is used on each device.

On each device, a same set of steps are executed iteratively. In the first step, *message generation*, messages are generated into message buffers (either local message buffer or remote message buffer, depending on the message destination) through the system primitive `send_messages()`, called in the user function `generate_messages()`. A message is a data unit containing a value pair, in the form of \((dst_id, msg\_value)\).
Before the message processing step, an implicit remote message exchange step is performed between devices. To reduce the communication overhead, a combination is conducted to the remote message buffer. The combination result is sent to the other device as a single MPI message. Runtime system invokes the user-defined function `process_messages` for message combination. Received messages are inserted into local message buffer for further processing.

Local message buffer is then accessed by the message processing step to conduct either SIMD message reduction or scalar processing. It is also accessed to update the local vertex values in the vertex updating step. Thus, the message buffer is the core data structure in the entire system. The design of this data structure impacts the performance of all the major steps, and thus, the performance of the entire system.

### 6.2.2 Message Buffer Design

To utilize the SIMD lanes (especially the wide lanes on Xeon Phi) – more specifically – to support SIMD reduction of messages, we should organize the messages in a way that the memory space is aligned, and that messages for the same destination are loaded into the same lanes while being reduced. As a secondary consideration, our design must be cognizant of the memory limits, i.e., that the memory size on a MIC chip is only a few GBs. Suppose the width of SIMD lanes is \( w \) bytes, and the size of an element to be processed in the message is \( \text{msg\_size} \). To support SIMD processing, we should wrap \( w/\text{msg\_size} \) messages together in a way that they are aligned with a multiple of \( w \) bytes.

The runtime system stores the messages as aligned vector types, including `vint`, `vfloat`, and `vdouble`, where exactly \( w/\text{msg\_size} \) scalar data elements are stored in an aligned fashion, same as the `vtypes` mentioned in Section 6.1.
We pre-allocate the buffer in a *condensed static* fashion, before any iteration is executed. An example of this buffer is shown in Figure 6.2, where a buffer is created for the example graph in Figure 1.1. The *condensed static buffer* (CSB) is created for an input graph in the following steps: 1) Sort the vertices according to the in-degree in descending order. A *redirection map* is generated for redirecting messages from original *dst_ids* to the proper positions after sorted. The redirection map will be used in message insertion, to be stated later in this section. 2) Group the sorted vertices into *vertex groups*. Each vertex group contains $k \times w/\text{msg.size}$ vertices, where $k$ is a small constant and $w$ is the SIMD lane width in bytes. In the example in Figure 6.2, $k$ equals 2. 3) Obtain the maximum in-degree among the vertices in each vertex group (denoted as $\text{max.group.degree}$). Allocate $k$ *aligned vector* arrays for each vertex group with an array size of $\text{max.group.degree}$.

We use the directed graph shown in Figure 1.1 to show how the CSB shown in Figure 6.2 is constructed. The graph contains 16 vertices, with the maximum in-degree being 5 for Vertex 5, and minimum in-degree being 0 for Vertices 14 and 15. The vertices are sorted according to in-degrees in descending order, and the sorting result is shown in the table in figure 6.2. For simplicity, we assume the SIMD lane to be as wide as 4 messages ($w/\text{msg.size} = 4$), and that we take the value of $k$ as 2. Thus, we are combining 8 vertices into a same *vertex group*, resulting in two vertex groups in total. For the first vertex group, the maximum in-degree among all the vertices is 5, and for the second vertex group, the maximum in-degree is 1. For the first vertex group, 2 arrays of *aligned vector* type are created, with the length of each being 5. Similarly, for the second vertex group, 2 *aligned vector* arrays are created, with the length being 1.

Such a buffer design significantly reduces the memory requirement because we group vertices with similar in-degrees together and avoid unnecessary memory consumption for
small in-degree vertices. At the same time, this method allows likely use of SIMD lanes for processing one message each for a set of vertices (i.e., \( w/msg_size \) vertices).

### 6.2.3 Processing of Messages Using CSB

Despite the advantages in memory efficiency, there are certain challenges in processing messages using the CSB data structure we have introduced. Again, we use the example graph in Figure 1.1 and one iteration in the SSSP algorithm to show what challenges are faced, as well as how we address them.

Suppose at a certain iteration, vertices 6, 7, 11, 13, 14, 15 (shown in black) just updated their shortest distances from the source using the incoming messages from the previous iteration. Thus, these vertices are active in this iteration and are sending the updated values out to their neighbors in the current iteration. The messages that are being sent are shown in Table 6.1. These messages are generated and sent in the *message generation step* of our system. Messages that are sent out from vertices are to be inserted into the message buffer.

**Vertex-Column Mapping:** The first problem is of mapping of vertices belonging to a vertex group to the columns of the vector arrays. The simplest method will be to use a pre-determined mapping (e.g., direct one-to-one mapping) between the vertices and the columns in the vector arrays. In this example, among all the vertices, only Vertices 2, 6,

<table>
<thead>
<tr>
<th>Source</th>
<th>Messages (dst_id, value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>(2, value)</td>
</tr>
<tr>
<td>7</td>
<td>(2, value)</td>
</tr>
<tr>
<td>11</td>
<td>(6, value), (9, value)</td>
</tr>
<tr>
<td>13</td>
<td>(9, value), (12, value)</td>
</tr>
<tr>
<td>14</td>
<td>(10, value)</td>
</tr>
<tr>
<td>15</td>
<td>(7, value)</td>
</tr>
</tbody>
</table>

Table 6.1: Messages Being Sent in the Example Graph
7, 9, 10, 12 have incoming messages. Whenever a message arrives, its destination vertex ID is translated to a position in the sorted table through the indirection map. The value of the message is inserted to the column in that position. After inserting the messages in Table 6.1 to the CSB, we get an insertion result that is shown in Figure 6.2(a): dark blocks in the buffer are the received messages, and others are empty. We can see that because a number of vertices in each vertex group do not receive any message, we are wasting SIMD lanes while conducting message reductions.
To solve this problem, we utilize a dynamic column allocation, which helps to condense the usage of columns. Facilitating such an implementation are an index array, and a column offset, which are maintained for every vertex group. Each element in the index array corresponds to a vertex and indicates the column index for the messages with this vertex as the destination. Before the message generation step in every iteration, all elements in an index array are initialized to -1, and the column offset is initialized to 0. Whenever a thread wants to insert a new message, according to the message’s destination vertex ID, the thread checks the index array for that vertex. If the column index is not -1, the thread inserts the message to that column. Otherwise, it allocates the next available column from that vertex group, using locking in the process, by exclusively incrementing the column index and writing the allocated column index to the corresponding element in the index array.

After all messages have been inserted within each vertex group, it is possible that only a subset of vector arrays in the front contain messages, while the remaining are completely empty. This allows for greater efficiency in using SIMD lanes. Recall that the width of a vertex group is $k$ times the SIMD width, so we have the possibility that $i \ (i < k)$ loop(s) of instructions may process all the vertices in the vertex-group. Figure 6.2(b) shows the result of message insertion using dynamic column allocation for the example.

**SIMD Message Reduction:** Now, let us see how messages from the CSB are reduced using SIMD parallelism. Iteratively, the runtime invokes the user-defined function `process_messages()` to process one aligned vector array from the CSB. Take the `process_messages()` function in the SSSP code as an example: users iterate on the `vfloat` type message array, and each time process a row of $w/msg\_size$ messages from the vector array. For MIC, because the SIMD lane width is 64 bytes and the messages being processed are of type `float`, simultaneously 16 messages participate in the overloaded `min()` function, which
wraps the SSE intrinsic \_mm512\_min\_ps for MIC. For CPU, 4 messages are processed simultaneously. Similarly, other arithmetic operations (e.g., +, −, ×, ÷, etc.) also wrap the corresponding intrinsics to process \( w/msg\_size \) messages in one invocation.

**Message Insertion to Columns:** Though the message buffer we designed makes it easy to exploit SIMD message processing and avoids excessive memory consumption, there are still other challenges to be addressed. An important question to be answered is how messages are to be inserted to the columns in the presence of parallelism. An intuitive approach is to use a *locking based* method – every thread inserts the messages it generates into the message buffer directly. However, because different vertices may send messages to same destinations (e.g. both Vertex 11 and Vertex 13 send messages to the Vertex 9 in Table 6.1), and because different vertices are distributed to different threads, concurrent message insertions to the same buffer column is going to be common. To ensure correctness, locking operations must be used, i.e., whenever a message is being inserted to a message buffer column, the computing thread should lock the entire column. Such locking will be required frequently, and moreover, could lead to contention among the threads.

Based on this concern, we have implemented a *pipelining scheme* for message generation. In this scheme, we divide the computation threads into *worker threads* and *mover threads*. Worker threads are only responsible for computation and message generation. They do not insert messages into message buffers, instead, they temporarily store the messages in message queues, working sequentially. Mover threads are responsible for moving the messages from the message queues into appropriate locations in the message buffer.

The example shown in Figure 6.3 uses 3 threads as mover threads. Each worker thread maintains private message queues, with the number of these queues being equal to the number of mover threads. After generating a message, the worker thread decides which
Figure 6.3: Message Generation Pipelining Using Worker and Mover Threads

message queue to insert it to, which is done based on the destination vertex ID using the expression: \( \text{queue}_\text{id} = \text{dst}_\text{id} \mod \text{num}_\text{mover}_\text{threads} \). A specific mover thread \( \text{tid} \) iterates over all worker threads’ message queues whose \( \text{queue}_\text{id} \) equals \( \text{tid} \), and moves the messages from these queues.

This strategy guarantees that each message queue is only written by only one thread, as well as read by only one thread. Similarly, a message buffer column can only be accessed by one mover thread. This is because messages are classified based on the modulo operation on the destination vertex IDs, and thus each message class (generated as a result of this classification) is handled by only one mover thread. Thus, a mover thread needs to use locking only at the time of buffer column allocation. Also, since the worker threads and mover threads work concurrently, the computation and memory accesses are overlapped effectively if the number of mover threads is chosen appropriately.

### 6.2.4 Intra-device Load Balancing

Because the numbers of incoming and outgoing edges to/from different vertices tend to vary, the amounts of processing associated with different vertices is different. Thus, if we
simply evenly distribute the workload based on the number of vertices, load imbalance will likely occur.

We utilize a dynamic workload scheduling for all sub-steps during the execution. For the message processing step, we treat the set of all vector arrays from all vertex groups as task units. All threads dynamically retrieve these task units through a mutex-protected scheduling offset. To lower the task retrieving frequency and thus the scheduling overhead, a thread can obtain multiple tasks each time. Although the numbers of messages contained in different vector arrays can vary significantly, dynamic scheduling of these message processing tasks ensures that all threads are kept busy until no more tasks are available. Similarly, for the vertex updating step, vertices are also dynamically scheduled to all the threads in blocks. Finally, for the message generation step, vertices are dynamically scheduled to the worker threads for computation and message generation. For the mover threads, because we classify messages based on the modulo operation of the destination IDs, each message class tends to have a similar number of messages. Although each message class is statically mapped to one mover thread, workload distribution among mover threads is expected to be well balanced.

6.2.5 Graph Partitioning between CPU and MIC

Our framework is able to execute a graph application across CPU and MIC. An important issue in enabling this functionality is to partition the workload between CPU and MIC. Because a CPU-MIC node is on top of a distributed memory space, it is not feasible to utilize dynamic workload distribution, since it will lead to high data movement costs. Thus, the framework statically distributes vertices to devices, before an application is run,
using a partitioning ratio (relative amounts of computation assigned to devices) specified by the users.

We now discuss how partitioning is performed by the system. The two desired properties from the partitioning method are: 1) load balance – suppose the expected workload ratio between the CPU and the MIC is \( a : b \), \( \text{edges}_{CPU} : \text{edges}_{MIC} \) should be close to \( a : b \), where \( \text{edges}_{CPU} \) and \( \text{edges}_{MIC} \) are the number of edges processed by the CPU and MIC, respectively. 2) minimized communication volume: cross edges, i.e., the edges whose source and destination are on different devices should be as few as possible.

We now examine different partitioning methods and introduce our proposed hybrid partitioning method. An intuitive way of partitioning a graph is continuous partitioning: suppose the partitioning ratio indicated by the user is \( a : b \), the first \( \frac{a}{a+b} \times \text{num}_\text{vertices} \) vertices are assigned to CPU, and the remaining vertices are assigned to MIC. The problem is that most graph datasets are power-law graphs, that is, the out-degrees of all vertices are not evenly distributed, and typically vertices with high out-degrees are together in a short range. If we simply partition graphs according to the number of vertices, the cumulative workload (cumulative out-degree) assigned to the devices are not proportional to the partitioning ratio provided by the users.

A candidate technique for solving the above problem is the round-robin assignment of vertices to devices: iterate over the vertices, for every \( a + b \) vertices, the first \( a \) vertices are assigned to CPU, and the remaining \( b \) vertices are assigned to MIC. This technique ensures that vertices are assigned to devices in an interleaved way, and thus avoids the possibility that the clustered highly connected vertices are assigned to a single device. The problem is that this partitioning can lead to a very high volume of cross edges between two partitions, and thus high communication overhead between devices.
Based on the above observation, we derive a hybrid partitioning scheme. In this scheme, we first partition the vertices into small blocks, and then assign the blocks to the devices in a round-robin fashion. While partitioning the graph into small blocks, we use the min-connectivity volume partitioning scheme provided by the Metis software [57], so that the number of cross edges among the blocks is minimized. This partitioning not only maintains low communication overhead, but also keeps the computation ratio to be consistent with the expected partitioning ratio.

6.3 Experimental Results

In this section, we report results from a series of experiments evaluating our system from different perspectives – comparison between different execution schemes, i.e., locking based execution and pipelining execution, comparison with OpenMP code, benefit of using both CPU and MIC compared with using a single device, as well as the benefits of using SIMD parallelism. We also evaluate the effectiveness of graph partitioning module, and the overall performance by showing speedups over sequential executions.

6.3.1 Evaluation Environment

Our experiments were conducted on a node with an Intel Xeon E5-2680 CPU and an Intel Xeon Phi SE10P coprocessor. The CPU has 16 cores, each running at 2.70 GHz. The size of the main memory is 63 GB. The Xeon Phi has 61 cores each running at 1.1 GHz, with four hyperthreads per core. The total size of the memory on this card is 8 GB. We used mpic++ 4.1, from Intel IMPI library, to compile all the codes, with -O3 optimization enabled. The mpic++ was built on top of icpc version 13.1.0. For the CPU-MIC symmetric computing, we compile separate binaries for MIC and CPU from the same source code, with -mmic flag for MIC, and -xhost for CPU.
Figure 6.4: (a) - (e): Comparison of Different Versions of Five Applications. (f): Impact of SIMD Processing.

6.3.2 Applications Used

Five commonly used graph applications are used to evaluate our system. Among them, PageRank is a graph algorithm [8] used to rank the importance of websites based on the number and quality of incoming links. Implementing a PageRank algorithm using our programming model involves treating each website as a vertex, and the value associated with each vertex (PageRank value) is initialized to 1. In each iteration, the message generation
sub-step propagates the PageRank value of each vertex to its neighbors, by dividing the value by the number of outbound edges. The message reduction sub-step sums up the received PageRank values from the neighbors, utilizing SIMD processing. The vertex update sub-step updates each vertex’s PageRank value using the sum. A directed graph Pokec [66] containing 1.6 million vertices and 31 million edges is used as the input.

**BFS**, Breadth First Search, is a very popular graph traversal algorithm. While implemented using our framework, initially, the source vertex is set as active, and its vertex value, level, is 0, while other vertices are inactive. In each iteration, active vertices send their level value plus 1 as messages to neighbors. Unvisited vertices which receive messages set their level, using any message that is received, and set themselves as active, and thus, message reduction is not needed. The execution ends when no active vertex exists. The same dataset with PageRank is used for this application.

**SC** or Semi-Clustering – is a graph based clustering algorithm, typically used for social network graphs. The algorithm is applied to an undirected graph, with each vertex representing a person, and an edge representing a connection between two individuals. Semi-clusters are groups of people such that people within a semi-cluster interact frequently with each other. Each vertex may belong to more than one semi-cluster. A semi-cluster is represented as an array of vertex IDs. Each vertex has an associated value, which is a vector containing at most a number (a pre-defined maximum value) of semi-clusters. Each edge has a weight representing the interaction frequency. A score for each cluster is computed and the clusters associated with every vertex are sorted according to the score in descending order. In the message generation sub-step, each vertex sends the top-score clusters to all of its neighbors. In the message processing sub-step, each vertex combines the received clusters with the clusters from its own vertex value, and sorts them according
to the score. Clusters with highest scores are used to update the vertex value. Because the message processing step is not associative and commutative, and the message type is not basic data type, SIMD reduction is not utilized. The dataset we used is a smaller undirected graph DBLP [120] containing 436 K vertices and 1.1 M edges. The reason why we use a smaller dataset is that the message size and vertex value size are much larger than the other applications and the dataset size is limited by the volume of the main memory on MIC. We converted the undirected graph to a directed graph by duplicating each edge, in order to fit the input into our system.

SSSP or Single Source Shortest Paths, was used as the running example throughout the paper. Same as PageRank, the message reduction sub-step in each iteration utilizes SIMD processing. The same dataset with PageRank is used in this application. In addition, we randomly generated weight value for each edge.

TopoSort, Topological Sorting, outputs a linear ordering for the vertices in a DAG (Directed Acyclic Graph), such that if there is an edge pointing from $u$ to $v$, then $u$ will appear before $v$ in the ordering. Programming this algorithm on our framework involves the following steps: initially, vertices with zero in-degree are set as active, and other vertices are inactive. In each iteration, active vertices send messages containing value 1 to their neighbors, and set themselves as inactive. Vertices receiving messages sum up the messages, and decrease their in-degree value using the sum. If a vertex’s in-degree becomes 0 after the substraction, it sets itself as active. The algorithm ends when no vertices are active. We use a randomly generated DAG containing 40K vertices and 200M edges as the input.
6.3.3 Overall Performance

In this section, we evaluate the performance of each application, and compare the different versions. Specifically, we evaluate both single-device executions and CPU-MIC executions. For single-device executions, besides the versions written using our framework, we also examine how our framework compares with OpenMP. Both OpenMP and our system provide a high level programming API, and both support SIMD parallelism either provided by the ICPC compiler with OpenMP directives or by the vector API of our programming model. The OpenMP version codes were compiled with ICPC at -O3 with flags -openmp -parallel. Although the Intel compiler supports offload pragma, so that a program could be executed on both CPU and MIC, it is not easy to write an offloaded graph algorithm using it, since the communication handling is non-trivial and hard to achieve efficiency. Thus, while comparing with OpenMP, we only investigate single-device executions.

The results are shown in Figure 6.4(a) to Figure 6.4(e). The CPU OMP and MIC OMP versions are written with OpenMP directives on sequential code, with proper use of synchronization (OpenMP locks). CPU Lock and MIC Lock are single-device executions written with our framework, using locking-based message generation. Similarly, CPU Pipe and MIC Pipe are single-device versions on our framework using pipelining message generation. CPU-MIC version is the heterogeneous version written with our framework.

We investigated the compiler vectorization report for OpenMP codes (both on CPU and MIC). It turns out that the major loops of the applications written in OpenMP are not vectorized, and thus OpenMP code could not benefit from SIMD parallelism of the MIC architecture. This is because of the random memory access pattern of graph applications. The message organization and SIMD APIs in our system enables the efficient utilization of SIMD, the details of which will be shown later in this section.
**CPU-only Executions:** The framework execution strategies we designed are more suited for the MIC architecture, which has a larger thread number and high parallel memory bandwidth. In contrast, CPU has a much smaller number of threads and thus contention overhead is not severe. Also, CPU has a much smaller memory bandwidth so that the overhead of messages storage in our system offsets the benefits from reduced contention and SIMD message reduction. On average, for CPU-only executions, OpenMP outperforms our framework by 2.5%. Also, locking-based executions outperform the pipelining executions, due to that the locking-based method does not need to store messages in message queue before they are inserted to message buffer, and that no threads need to solely work on message movement. For all the applications, best performance was delivered with a total of 16 threads, i.e, 1 thread per core.

**MIC-only executions:** With the exception of BFS, pipelining executions (MIC Pipe) outperform locking-based (MIC Lock) executions, as well as OpenMP versions (MIC OpenMP), though the relative performance varies depending on the property of applications. For PageRank, all vertices generate messages along all edges every iteration, and thus the number of messages generated is large. As a result, contention is severe while locking is used. The pipelining version is 2.33x faster than the locking based method, and 1.85x faster than the OpenMP version. For both locking-based framework execution and OpenMP version, 240 threads was used to achieve the best performance, while 180 worker threads + 660 mover threads achieve the best performance for pipelining framework execution (same thread configurations are used for the remaining applications as well). For BFS, in each iteration, only a subset of vertices are active and the amount of messages generated is small, and thus the contention overhead is less severe. The locking-based framework execution is 1.19x faster than pipelining execution. MIC lock and MIC pipe are 1.54x and 1.30x faster.
than MIC OMP, respectively. Even though neither OpenMP or framework use SIMD for message processing, OpenMP is slower, likely due to the more expensive locking operations. For SC (Semi-Clustering), pipelining version performs better than locking version (1.25x faster), as well as OpenMP (1.17x faster). The speedup achieved from pipelining execution solely comes from reduced contention overhead, as SIMD reduction is not applied in this application. For SSSP, both MIC Lock and MIC Pipe run slightly faster than OpenMP (1.11x, and 1.20x speedup, respectively). The speedup of MIC Lock over OpenMP mainly comes from SIMD message reduction, while MIC Pipe also benefits from the reduced contention. The last application, TopoSort, uses a highly connected graph, which implies that in each iteration, a large number of messages are sent to a single vertex. The expensive locking used in OpenMP adversely impacts the performance. Pipelining execution is 4.15x and 3.36x faster than OpenMP version and locking-based framework execution, respectively.

**CPU-MIC executions:** For each graph dataset, a min-connectivity blocked partitioning (256 partitions) result is generated using Metis. The blocked partitioning result is reused for generating hybrid partitioning results for different ratios. Although the blocked partitioning takes time, it is only applied to each dataset once, and thus we did not include the partitioning time. We tried different partitioning ratios between CPU and MIC, and the results reported here are from the ratios that gave the best load balance. Locking-based execution was used for CPU, as it is faster than pipelining execution, while for MIC, pipelining execution was used except for BFS. Both the execution time and the communication time are separately shown in result figures.

For PageRank, MIC Pipe is 1.72x faster than CPU Lock, and CPU-MIC achieves a 1.30x speedup over the faster of single-device executions, i.e., MIC Pipe, at a graph
partitioning ratio of 3:5 (CPU 3, MIC 5). For BFS, CPU Lock is 1.30x faster than MIC Pipe. Using CPU and MIC together results in a 1.32x speedup over CPU Lock version, at a graph partitioning ratio of 4:3. CPU performs much faster than MIC for SC, due to the more complex conditional instructions involved, which CPU is better at. CPU-MIC version is 1.29x faster than CPU Lock version, using a partitioning ratio of 2:1. For SSSP, CPU and MIC have a very similar performance, and using both achieves a speedup of 1.41 (using a equal graph partitioning). The last application, TopoSort, MIC is 3.32 times faster than CPU. Using CPU and MIC together has a smaller speedup of 1.20x over the MIC only execution (using 1:4 graph partitioning). Also note that the input for TopoSort is a highly connected graph, and the number of vertices is much smaller compared with the large number of edges, so that the communication overhead is negligible.

6.3.4 Benefits of SIMD Execution for Message Processing

Among the five applications, two applications involve no SIMD message reduction – specifically, SC uses sorting in message processing step, and BFS does not have message reduction sub-step. The other three applications perform message reduction through nuanced runtime SIMD parallelization supported in our framework. We now quantify the benefits obtained from runtime SIMD parallelization. For this purpose, we re-wrote the message processing sub-step for these three applications in a scalar way, for both CPU-only and MIC-only executions.

Figure 6.4(f) shows the execution times with and without vectorization. All reported data is from execution strategies and thread configurations that deliver the best results. For CPU executions, the vectorized versions achieve a speedup of 2.24, 2.35, and 2.22 over non-vectorized versions for the message processing sub-step for PageRank, SSSP, and
TopoSort, respectively. Similarly, for MIC executions, 6.98x, 5.16x and 7.85x speedups are achieved for message processing steps. Although up to 4(CPU)/16(MIC) floating point messages are processed simultaneously by the SIMD lanes on each core, the achieved speedup is limited due to several factors: bubbles in the lanes due to the difference in the number of received messages for each vertex, and processing can become memory bound after a certain point. The benefits of SIMD processing on the overall performance depends upon the relative amount of time spent on the message processing phase. For CPU executions, these speedups are: 9%, 13% and 8% for PageRank, SSSP and TopoSort, respectively. For MIC executions, the speedups are: 18%, 23%, and 21%, respectively, for these applications.

6.3.5 Effect of Hybrid Graph Partitioning for CPU-MIC Execution

![Figure 6.5: Impact of Different Graph Partitioning Methods](image)

To illustrate the effectiveness of the hybrid graph partitioning, we executed the CPU-MIC versions with continuous partitioning, round-robin partitioning, and hybrid partitioning. The partitioning ratio used for each application is the same as that is used for achieving the best CPU-MIC execution in Section 6.3.3. Figure 6.5 shows the time distribution
of each application using the different partitioning schemes. The execution time is determined by the slower device, and the communication time is the time spent on exchanging messages between devices. For PageRank, a 3:5 partitioning ratio is used as the input to the partitioning modules. Both round-robin and hybrid partitioning methods achieve balanced partitioning results: the ratio of cumulative out-degrees on CPU and MIC is close to the expected workload ratio (3:5). Because vertices with higher out-degrees are concentrated at the front of the graph Pokec, continuous partitioning leads to a larger than expected number of edges on CPU, but less than expected number of edges on the MIC. This leads to a much longer overall running time due to significant load imbalance. Round-robin partitioning results with 2.27x more cross edges than hybrid partitioning, and thus much longer communication time. Execution using hybrid partitioning is 1.72x and 1.13x faster than continuous and round-robin partitionings, respectively. Similar trends can be seen from the other applications: for BFS, using hybrid partitioning is 1.31x and 1.09x faster than the other two partitioning methods; for SSSP, the speedups are 1.50 and 1.10; and for SC, 1.17 and 1.36. The graph DAG used in TopoSort has almost equal number of cross edges using round-robin and hybrid partitionings, and thus similar execution times using either partitioning, though running with continuous partitioning result is much slower than the other two due to the imbalanced partitioning result.

6.3.6 Overall Performance Gained from Our Framework

To see the efficiency and scalability of our framework, we compare different versions using our framework with sequential versions (written in C/C++ and executed by one core). We run sequential versions on both CPU and MIC.
<table>
<thead>
<tr>
<th>Application</th>
<th>Execution times (Sec)</th>
<th>CPU Seq</th>
<th>MIC Seq</th>
<th>CPU Multi-core</th>
<th>MIC Many-core</th>
<th>CPU-MIC Best</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PageRank</td>
<td>BFS</td>
<td>SC</td>
<td>SSSP</td>
<td>TopoSort</td>
<td>PageRank</td>
</tr>
<tr>
<td>PageRank</td>
<td>18.01</td>
<td>1.46</td>
<td>8.29</td>
<td>2.62</td>
<td>8.42</td>
<td>181</td>
</tr>
<tr>
<td>BFS</td>
<td>5.01</td>
<td>0.29</td>
<td>1.09</td>
<td>0.52</td>
<td>2.20</td>
<td>(3.6x)</td>
</tr>
<tr>
<td>SC</td>
<td>134.06</td>
<td>24.07</td>
<td>85.17</td>
<td></td>
<td></td>
<td>(181)</td>
</tr>
<tr>
<td>SSSP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TopoSort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Parallel Efficiency Obtained from Our Framework

The results are shown in Table 6.2. Note that even though the clock frequency of a CPU core is only 2.4 times faster than a core on MIC, a CPU core runs the same sequential code around 11x faster, on the average, because of out-of-order execution and other enhancements. The speedup of CPU multi-core execution on our framework ranges from 3.6x to 7.6x. Similarly, MIC many-core execution has a speedup of 32x to 129x, compared with MIC sequential execution. The last application, Topological sorting, achieves the highest speedup on MIC, due to the high connectivity of the DAG, leading to a high computation density. The combined execution of CPU and MIC, while compared with CPU sequential version, achieves a speedup of between 6.7x and 15.3x.

6.4 Summary

With growing density of transistors and increasing focus on energy efficiency, future processors will have increasing number of cores, parallelism at multiple levels, and relative small amount of memory per core. Such a trend is reflected through a new coprocessor, Intel Xeon Phi. More specifically, Xeon Phi is an example of a system where large-scale
shared memory (MIMD) as well as SIMD parallelism must be exploited to achieve high efficiency.

This chapter has focused on the challenge of exploiting these two types of parallelism simultaneously for graph processing applications. We have supported a simple graph programming API. The innovative components of our runtime system include a condensed static memory buffer, which supports efficient message insertion and SIMD message reduction while keeping the memory requirements low, and a pipelining scheme for efficient message generation by avoiding frequent locking operations. Although these techniques are specifically designed for the MIC architecture, they are able to run on a multi-core CPU, with a reasonable performance, compared with OpenMP. Our framework is able to execute a graph application across the CPU and MIC, with the assistance of our proposed hybrid graph partitioning.
Chapter 7: Exploiting Recent SIMD Architectural Advances for Irregular Applications

We have introduced a graph processing system in the previous chapter, which is a framework for graph processing over the CPU and Xeon Phi. Graph applications are typical irregular applications, involving complex random memory accesses and data dependencies. In this chapter, we extensively study the problem of accelerating irregular applications on SIMD architectures, specifically, the Intel Xeon Phi.

7.1 Exploiting New SIMD Architectural Features for Irregular Applications

We present our proposed methodology initially focusing on irregular reductions. Later, we will show how the same basic idea can be applied on other application classes.

7.1.1 Irregular Reductions

The computations in an irregular reduction can be summarized as reading and updating data arrays, indexed by the non-zeros from a sparse matrix. Because of the sparsity of the matrix, the accesses to the data arrays are non-continuous, and in most cases, the stride between accesses can be very large, leading to very low memory access efficiency. At the SIMD level, though gather/scatter operations make the irregular bulk read/write
possible, the performance is not expected to be high. In addition, cache performance is also negatively impacted by the large access strides.

Our main idea is what we refer to as **hierarchical tiling**, where the sparse matrix is stored as tiles of several different *tile sizes*. A tile is a square portion of the imaginary dense matrix – however, only the non-zeros in each tile are actually stored. Thus, a tile is represented as a COO format sparse matrix, including three aligned arrays: *rows*, *cols*, and *vals*. Figure 7.1(a) shows the tiling result of the sparse matrix involved in an irregular reduction kernel – for simplicity, we have tiles of only one size, which is $4 \times 4$ in this example.

During the execution, a tile is processed exclusively by one thread, focusing on exploiting the SIMD instructions. Our goal is to have sufficient number of non-zeros in each tile, and thus to have a high SIMD utilization ratio. At the same time, unnecessarily large tile sizes can negatively impact locality in accessing the data arrays. Certain portions of the sparse array can have a large concentration of non-zeros as compared to others – thus,
**Algorithm 9: Transforming a Tile to Conflict-free Groups**

Input: tile: A tile in COO format  
Output: A vector of conflict-free COO groups  

```plaintext
def grouped = false;
foreach nnz in tile do
    foreach g in groups do
        if g.size < 16 && nnz:row ∈ g:rows && nnz:col ∈ g:cols then
            g:add(nnz);
            grouped = true;
    // nnz is incompatible with any existing group
    if !grouped then
        // create a new group and add nnz to it
        groups:push_back(empty_coo);
        groups:back().add(nnz);
return groups;
```

dividing the entire sparse array into tiles of one size is unlikely to be effective. This is the motivation behind our idea of hierarchical tiling. Specifically, we tile the sparse matrix iteratively. For example, we first extract $128 \times 128$ tiles, with the number of non-zeros in each tile being no less than a certain predefined threshold $n$, from the original sparse matrix. We next extract $256 \times 256$ tiles, each containing at least $n$ non-zeros, from the remaining sparse matrix. We limit the total number of distinct tile sizes to 3 in our implementation, as very large tiles only harm the data access locality. At the last iteration, the remaining sparse matrix is already very sparse, and we set the threshold $n$ as 1, which means, any tile with at least 1 non-zero is extracted. Typically, the first tiling iteration extracts most of the non-zeros, if the threshold $n$ is set to a not too large value, e.g., 32.

The next issue is of write conflicts. To motivate this issue, consider again the Figure 7.1(a), which also shows the node data and reduction array accesses for each tile. Non-zeros in a same row/column lead to reads on the same node data, and similarly, they also lead to writes to the same position in the reduction array. This access pattern is challenging for both SIMD and MIMD execution, because of the write conflicts. At the SIMD level,
write conflicts among SIMD lanes should be completely avoided, as no locking mechanism is supported by the VPU. Though locking is able to protect concurrent read/write conflicts among threads at MIMD level, it is critical to avoid these. This is because not only the locking operation itself is expensive, but the contention among large number of threads can also lead to a significant overhead.

In our approach, we remove the write conflicts by further partitioning the non-zeros in each tile into conflict-free groups. The simplest way of organizing conflict-free groups is to partition the tile by diagonals, since along any single diagonal, there are no two non-zeros that have the same row number or the same column number. But due to the sparsity of the tile and the short diagonals in the upper right and lower left sides, the length of each group might be too short to efficiently utilize the SIMD lanes. In view of these, we derive a more effective greedy conflict removal method, which is shown through Algorithm 9. For a non-zero $nnz$, we try to pack it to the first conflict-free group $g$ that it is compatible with. By compatible, we imply that there is no non-zero in $g$ that has either same row ID or column ID with $nnz$. The overall idea is to improve SIMD efficiency by merging multiple sparse diagonals into groups. We limit the maximum size of each conflict-free group to be the vector length of the SIMD lane, i.e., 16 for float types.

The output of the algorithm is a vector of conflict-free COO groups. The arrays ($rows$, $cols$, and $vals$) in each COO are padded to the SIMD vector length (i.e., 16). The COO groups are repeatedly executed in SIMD, with each lane processing one non-zero each time. The $rows$ and $cols$ are used as indices in gather and scatter. $vals$ are read using aligned load operations.

Similar to SIMD processing, we also need to consider write conflicts among threads that are processing different tiles. If two tiles involve the same row/column, write conflicts
during the reduction stage can arise. To avoid locking (and associated contention) overheads, we group the tiles into conflict-free tile groups, using a same logic as Algorithm 9 (though without the maximum group size (16) limit). Each parallel step processes one such tile group using all available threads.

7.1.2 Graph Algorithms

Graph algorithms are also computations involving sparse matrices and vertex arrays. They have both similarities and differences with the irregular reductions, and thus, though we can broadly use the same methodology, we need to make certain modifications as well.

We now use a representative application, Bellman-Ford, to show the typical properties of graph algorithms – especially those that can be processed by one of the popular data parallel graph frameworks like Pregel [75]. Bellman-Ford is an iterative kernel conducted over a graph \( G = (V, E) \). It computes the shortest distances from the source \( s \) to each vertex \( v \in V \). In each iteration, it conducts a relaxation over each edge \( e(u \rightarrow v) \in E \), by decreasing the distance value of \( v \), if \( v.d > u.d + e.w \). Overall, each iteration loops on the edges (non-zeros) in the graph (sparse matrix), and the relaxation steps use each source vertex’s value \( u.d \), together with the value of the non-zero element in the sparse matrix \( e.w \), to perform a computation \( u.d + e.w \), and update the value of the destination vertex’s value \( v.d \).

The set of edges can again be stored as a sparse matrix, and we can apply the same hierarchical tiling to the sparse matrix. However, the data access pattern of graph algorithms has one critical difference over those in irregular reductions. As is shown in Figure 7.1(b), the data access for computing associated with one edge (non-zero value in sparse matrix) involves the following three steps: 1) Reading source vertex data from the array on the left,
2) Loading edge data from the sparse matrix, and 3) Updating the destination vertex on the top. Specifically, the left array is read-only, and only the top array is written. This introduces a critical difference in the algorithm for finding conflict-free groups – i.e., we only need to group the non-zeros according to the column ID, and not both row and column IDs. Two non-zero values in the sparse matrix can be grouped in the same conflict-free group as long as they have distinct column IDs. An example of grouping is shown by Figure 7.1(b): the non-zeros in the central tile are grouped into two groups. SIMD processing can again be applied to each of the groups. For MIMD parallelization, write conflicts exist only among the tiles in the same column, so the tile columns are scheduled as basic task units to different threads.

7.1.3 Sparse Matrix-Matrix Multiplication

Unlike irregular reductions and graph algorithms, Sparse Matrix Matrix Multiplication (SpMM) is an irregular application that accesses multiple sparse matrices. As a result, the data access pattern is quite different. Nevertheless, our methodology is still broadly applicable, as we will show here.

A sequential kernel for computing SpMM involves iterating over the non-zeros in the first matrix $A$. For each non-zero $e$ that is inspected, according to its column ID $e_.col$, we scale the non-zeros in the row of $e_.col$ from the second matrix $B$. The scaling results are accumulated to a hash table corresponding to the row $e_.row$.

Before presenting our proposed method, we discuss similarities and differences from the work done in context of GPUs. Demouth et al. proposed an implementation of SpMM on the GPU [32]. The idea is to load a number of non-zeros from a row of $A$ in a SIMD fashion, and use these elements to scale the elements in the corresponding row in $B$, based
on the column ID of the element from $A$. The scaling operation is a SIMDized multiplication, and the multiplication result has to be accumulated back to a hash table corresponding to the specific row in the result matrix $C$. This method is not feasible for the modern SIMD machines, as the accumulation of the multiplication result is challenging – because the column IDs of the non-zeros in the row from $B$ are not continuous, one cannot do the reduction in a SIMD manner. Instead, one needs to unpack the SIMD multiplication result (from a vector) and reduce the scalar values to the hash table. Through our preliminary experiments, we determined that the unpack step is very time consuming (taking 85% of the total execution time).

We now describe our proposed method, which is based on the fact that most sparse matrices (especially the ones from real world data, e.g., social network graphs) exhibit clustered distribution of non-zeros. Accelerating these implicit regular components (clustered non-zeros) with SIMD can provide impressive speedups for the entire computation.

The first step in our methodology is a hybrid storage format for a sparse matrix. We tile the sparse matrices $A$ and $B$ into $4 \times 4$ tiles, with each non-empty tile being stored as a dense matrix. Such $4 \times 4$ tiles fit the modern SIMD lane width, especially for float type, as will be seen in the following texts. The dense matrices (tiles) are indexed by CSR. The processing will be similar with the sequential code, and the only difference is that multiplications need to be performed between these dense tiles (in SIMD), instead of the scalar non-zeros. To divide the workload among threads, each thread is responsible for a subset of rows of the tiled matrix $A$. The computation results from different threads will be reduced to independent hash tables corresponding to the different rows in the result, and thus, there is no write conflict between the threads. The values stored in the hash table are
dense tiles, instead of scalars, and thus the reduction operations are conducted in a SIMD fashion.

The challenge of eliminating write conflicts while maintaining a high SIMD utilization ratio still remains. We show that we are able to avoid the inter-lane conflicts by permuting the multiplications and the SIMD lane mapping. Figure 7.2 shows how SIMD multiplication is conducted between two dense tiles ($T_a$ and $T_b$). For simplicity of illustration, we use $2 \times 2$ tiles, so that 2 iterations are needed, each iteration producing and accumulating 4 multiplication results. Similarly, for $4 \times 4$ tiles (as is used in our implementation on Intel MIC), 4 iterations are involved, each iteration producing and accumulating 16 multiplication results. Each shade in the figure corresponds to one SIMD lane. In each iteration, a corresponding one-to-one mapping between the elements in $T_a$ and $T_b$ is used for SIMD multiplication, and the multiplication result is reduced to the corresponding elements in the partial result tile $T_c$, in a SIMD fashion, using respective index mappings. After the tile-wise multiplication, $T_c$ is reduced to the hash table, also in a SIMD fashion. For the
example in Figure 7.2, the multiplication for the first iteration is:

\[
T_c(0,0) = T_a(0,0) \times T_b(0,0), \\
T_c(0,1) = T_a(0,1) \times T_b(1,1), \\
T_c(1,0) = T_a(1,1) \times T_b(1,0), \\
T_c(1,1) = T_a(1,0) \times T_b(0,1).
\]

and the multiplication for the second iteration is:

\[
T_c(0,0) = T_a(0,1) \times T_b(1,0), \\
T_c(0,1) = T_a(0,0) \times T_b(0,1), \\
T_c(1,0) = T_a(1,0) \times T_b(0,0), \\
T_c(1,1) = T_a(1,1) \times T_b(1,1).
\]

The mapping above ensures that the multiplication in each iteration does not have inter-lane
write conflicts, as the SIMD lanes accumulate multiplication results to different elements in
\(T_c\). As the multiplication between any two dense tiles follows a same pattern, the mapping
indices are hard coded and used repeatedly.

The 4 \times 4 tiles contain 16 elements, and can match the SIMD lane perfectly if the
elements are of type \texttt{float} or \texttt{int}, as the size of the tile equals the SIMD lane width (64
bytes). In each of the above steps, only one SIMD instruction is executed. On the other
hand, if the element type is \texttt{double}, in each step, the same SIMD statement is executed
twice, each for 8 \texttt{doubles}. Also, one can see that the \texttt{gather/scatter} intrinsics used
here are very efficient, as the elements that are accessed together within a tile are physically
stored close to each other.
7.1.4 Matrix Reordering for Irregular Reductions and Graph Algorithms

Graphs and unstructured meshes often have (large) variance in connectivity. This can mean significant irregularly in the distribution of non-zeros. As a result, the tiles can be sparse, leading to low SIMD utilization ratio and/or poor data access locality. Sparse matrix reordering, using a software like Metis [57], has been widely used for irregular applications [82, 89, 106]. In our context, we will try to apply such a reordering as a precursor to tiling, to reduce the irregularity.

Specifically, we use the software Metis to divide the entities into a number of partitions. The goal in creating these in these partitions is to maximize the number of edges within each partition, and minimize the number of edges crossing the partitions. The entities are renumbered according to the partitioning result. We note that such partitioning can be quite expensive.

7.1.5 Discussion

Preprocessing Overheads: Our proposed methods introduce certain preprocessing overheads, as is the case with any other existing optimization techniques. The tiling overhead is linear to the input size, as only one pass to the input is needed. Multilevel k-way Partitioning [57] is used in Metis for reordering of the sparse matrices. Though the reordering also performs at a linear complexity, it has a much larger constant factor and thus much longer execution time. However, these overheads can be amortized because they need to be applied only once. For example, most of the irregular application kernels used in scientific
or data intensive computations are iterative kernels. Similarly, a graph application can involve a large number of iterations, and/or a number of different algorithms may be applied on the same dataset.

**Other Irregular Applications:** Our work is also related to existing work in the context of Sparse Matrix Vector Multiplication (SpMV) [24, 72, 104]. Our method, if applied to SpMV, will create very similar schemes as the one reported in these studies. Thus, the main contribution of our work is generalizing these ideas to be applicable to irregular reductions, graph applications, and SpMM.

### 7.2 Evaluation Results

A series of experimental studies were conducted to evaluate our methodology for each application subset. Specifically, the goals were: 1) Impact of vectorization – we compare the performance of our approach against serial code, serial code with tiled inputs, as well as straightforward SIMD implementations for each application. For irregular reductions and graph algorithms, we also evaluate in details the impact of varying the tile sizes. 2) Overall performance that is delivered by using combined MIMD and SIMD parallelism on Intel Xeon Phi. and 3) Performance one can obtain from matrix reordering prior to tiling, for irregular reductions and graph algorithms.

#### 7.2.1 Evaluation Environment

Our experiments were conducted on an Intel Xeon Phi SE10P coprocessor, involving 61 cores each running at 1.1 GHz, with four hyperthreads per core. The GDDR5 main memory is 8 GB. We used Intel ICC compiler 13.1.0, to compile all the codes, with -O3 optimization enabled.
### 7.2.2 Applications Used

<table>
<thead>
<tr>
<th>Application</th>
<th>Dataset</th>
<th>Dimensions</th>
<th>NNZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Irrregular Reducions</td>
<td>Moldyn</td>
<td>32-3.0r</td>
<td>131K*131K 11M</td>
</tr>
<tr>
<td></td>
<td>45-3.0r</td>
<td>365K*365K 30M</td>
<td></td>
</tr>
<tr>
<td>Euler</td>
<td>gsm_106857</td>
<td>589K*589K 11M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kron_g500-logn19</td>
<td>524K*524K 22M</td>
<td></td>
</tr>
<tr>
<td>Graph Algorithms</td>
<td>Bellman-Ford</td>
<td>soc-Pokec 1.6M*1.6M 31M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>higgs-twitter</td>
<td>457K*457K 15M</td>
<td></td>
</tr>
<tr>
<td>PageRank</td>
<td>soc-Pokec</td>
<td>1.6M*1.6M 31M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>higgs-twitter</td>
<td>457K*457K 15M</td>
<td></td>
</tr>
<tr>
<td>SpMM</td>
<td>SpMM</td>
<td>msc10848</td>
<td>11K*11K 620K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>conf5_4-8x8-05</td>
<td>49K*49K 2M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shipsec5</td>
<td>180K*180K 5.1M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>crankseg_1</td>
<td>53K*53K 5.3M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>appu</td>
<td>14K*14K 1.9M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>598a</td>
<td>111K*111K 742K</td>
</tr>
</tbody>
</table>

Table 7.1: Applications and Datasets Used in the Experiments

Table 7.1 shows the applications and datasets that are used for each application subset. For each application, multiple datasets of different sizes are used. The evaluation of irregular reductions is based on Moldyn and Euler, two popular kernels widely used. Moldyn simulates the interaction and motion of molecules in a period of time, and involves an irregular reduction kernel for computing the forces among molecules based on the interactions, which are stored in an interaction list. Euler is a Computational Fluid Dynamics (CFD) simulation on an unstructured mesh. The input matrices for Euler come from the UFL Sparse Matrix Collection [30]. In applying our methodology, the interaction list of each of these applications are transformed into standard sparse matrices, and the optimization steps are applied. For both applications, the computation was conducted with type float.
We also use two popular graph applications – the first application, Bellman-Ford, was described in Section 7.1.2. The other application that is used is PageRank [8]. This application assigns weights to each element in a hyper linked set of documents (represented as a graph). It is widely used in search engines to estimate the importance of a web page and is an iterative application. In each iteration, each vertex distributes its rank evenly to all the vertices it directs to. After several iterations, the rank values of the vertices will precisely reflect the relative importance. The graphs used in both applications are stored as sparse matrices. The two graphs used are from the SNAP [68] graph dataset. float is the basic type for the computation for both the applications.

For SpMM, we only evaluate the kernel itself, with different matrices as the input. Each matrix is multiplied with itself. The matrices we use vary in sizes and in the the properties of non-zero distributions – specifically, we tried both clustered and randomly distributed matrices. We show test results with both single and double precision executions. We compare our optimized approach with different other implementation methods, including the straightforward approach mentioned in Section 7.1.3, as well as the highly optimized Intel MKL Sparse BLAS Level 3 routines mkl_scsrmultcsr (for float) and mkl_dcsrmultcsr (for double). The parallel MKL routines benefit from both MIMD and SIMD. The matrices used for the tests are also from the UFL Sparse Matrix Collection.

For both irregular reductions and graph algorithms, we compare against a recently published method, proposed in the context of Moldyn, by Pennycook et al. [84]. The approach here is based on an adjacency list view, so that each element is associated with a row of neighbors. The adjacency list is processed row by row, and for each row, the data of the row element (the element to which the neighbors are associated) is loaded into a vector \( V_r \). The processing of the neighbors is conducted in a SIMD manner: for every SIMD step,
data for 16 (since we are using float data type) neighbors are loaded using gather into a vector \( V_n \), according to the IDs of these neighbors. SIMD computation is conducted between \( V_r \) and \( V_n \). Each SIMD computation result is scattered back to the reduction array for the neighbors, again according to the neighbor IDs. However, for MIMD execution, this method is not expected to scale well due to the fact that it uses local reductions to avoid inter-thread (inter-row) conflicts, and an expensive final combination has to be conducted among threads. Thus, we only compare the SIMD execution with this method.

The results reported here, including the execution times and speedups, for all applications, are the average value for a single iteration (averaged over 100 iterations).

![Figure 7.3: SIMD Utilization Ratios of Datasets (Used by Irregular Reductions and Graph Algorithms) under Different Tile Sizes](image)

**7.2.3 SIMD Utilization Ratio**

SIMD utilization ratio is an important factor that influences the efficiency of the vectorization. Before showing the execution results for any application, we analyze the SIMD utilization ratios for all the datasets that are used for different applications. The average SIMD utilization ratio is calculated as \( \text{original}_{\text{nnz}} / \text{padded}_{\text{nnz}} \), where \( \text{original}_{\text{nnz}} \)
stands for the number of non-zeros in the original input, and \( \text{padded}_{\text{nnz}} \) is the number of non-zeros after the tiling, which is larger than \( \text{original}_{\text{nnz}} \), as padding is used in both the conflict-free groups (for irregular reductions and graph algorithms), and the \( 4 \times 4 \) tiles in SpMM. Figure 7.3 shows the SIMD utilization ratio for datasets used in irregular reductions and graph algorithms, under different tile sizes. The tile size here indicates the size used in the first tiling iteration – the subsequent tiling iterations just double the tiling size of the previous one. The utilization ratio keeps increasing with the tile size getting larger, due to the fact that more non-zeros are included in each tile, and that less padding is used. For four out of the six datasets shown in the figure, SIMD utilization ratios is above 80% for tile sizes larger than 4096. Moreover, three datasets have utilization ratios above 90% at the tile sizes of larger than 2048. SIMD utilization ratios for the four clustered matrices used in SpMM vary between 47.2% and 55.8%, which are acceptable since the multiplications between tiles are completely regular, and combined with the optimized access locality, the vectorization performance is acceptable, as will be shown in later sections. The SIMD utilization ratios for the two random matrices are very low, only around 7% for both, and these matrices do not fit our approach.

### 7.2.4 Results from Irregular Reductions

#### SIMD Parallelism

Figure 7.4 shows the execution times of Moldyn and Euler, under different tile sizes, with single-thread SIMD execution using our approach. We also measured L1 cache miss rates with Intel VTune. With the tile size increasing, the L1 cache miss rates first decrease and then increase. This is because very small tile size imply that gather/scatter operations switch among the tiles frequently. On the other hand, very large tiles also have bad data locality inside each tile. The overall performance is impacted by SIMD utilization
Figure 7.4: Single Thread SIMD Execution Times of Moldyn and Euler under Different Tile Sizes

Figure 7.5: Single Thread Performance of Moldyn and Euler with Different Approaches

ratio (increasing with tile sizes) and the cache miss rate (decreasing and then increasing), and the increasing access range of gather/scatters.

To see the relative performance of different SIMD parallelization schemes, we plot the performance of different execution configurations that all use only one thread. The
versions are *Serial* (serial code on original input, baseline), *Serial Tiling* (serial code on tiled input, with the tile size that gives the best performance), *SIMD Naive* (straightforward vectorization used in [84]) and *SIMD Tiling (Our)* (vectorization using our method, with the tile size that gives the best performance). The results are shown in Figure 7.5. We can see that compared with serial codes, straightforward vectorization does not achieve better performance (slows down in most cases), due to different reasons including the large data access ranges of `gather/scatter`, less efficient cache usage due to poor data locality, as well as inefficient inter-lane reductions. On the other hand, tiling significantly optimizes the data locality, which can be seen from the fact that *Serial Tiling* is obviously faster than *Serial*, ranging from 1.03x to 2.24x improvement. The cache miss rates of *SIMD Naive* are all much higher than those of *SIMD Tiling (Our)*, based on the Vtune profiling results. The overall performance of *SIMD Tiling (Our)* is significantly better. For *Moldyn*, the speedup over *Serial* is 9.05, and 7.47, on the two different inputs, respectively. For *Euler*, the speedups are 2.09 and 2.01. The reasons why *Euler* has smaller speedups are that the inputs are much sparser than those used by Moldyn and that Euler has a much lower computation to memory access ratio.

**Overall Performance of MIMD+SIMD**

The performance of versions that used different number of cores and performed SIMD execution on each core is shown in Figure 7.6. The speedups shown in the figure are over the serial executions. For each of the two irregular reduction computations, we plot the scalability under different tile sizes. The overall trend indicates that smaller tiles have better scalability than larger tiles with the number of threads increasing – the reason being that smaller tiles result in higher inter-thread parallelism, as well as better load balance. For example, the speedups using tile size of 128 keeps increasing till over 200 threads, for all
the cases. However, smaller tiles have low SIMD utilization ratios. Thus, the best overall performance is typically achieved at a medium tile size, where the right trade-off between SIMD and MIMD parallelism is achieved. The best speedups of Moldyn are 176 and 290, for 32-3.0r and 45-3.0r, respectively, both with 1024 tile size and 80 threads. For Euler, a 25.37x speedup is achieved for the small dataset, *gsm_106857*, using tile size of 2048 and 61 threads. The larger dataset, *kron_g500-logn19*, achieves a higher speedup of 64 at the thread of 200, using a tile size of 2048.

**Matrix Reordering Optimization**

For each dataset that is used above, we also generated a reordered version with Metis, using the approach described in Section 7.1.4. The same optimization methodology was then applied to reordered matrix. The reordered datasets are able to provide further speedups for certain applications and datasets. For Moldyn, speedups of 1.3 and 1.69 are achieved.
with datasets 32-3.0\textit{r} and 45-3.0\textit{r}, respectively, according to the best performance obtained with MIMD+SIMD executions. For Euler, slowdowns are observed for both inputs, due to that the reordering lowers the SIMD utilization ratios – specifically, the write conflicts within each tile is more severe after reordering. In addition, it should be noted that reordering adds significant preprocessing overhead (which is not directly reported here).

### 7.2.5 Results from Graph Algorithms

#### SIMD Parallelism

![Figure 7.7: Single Thread SIMD Execution Times of Bellman-Ford and PageRank under Different Tile Sizes](image)

Figure 7.7 shows the execution times of the two graph applications under different tile sizes. Same as irregular reductions, we also investigated the L1 cache miss rates of these applications, with the help of Vtune. The cache miss rates nearly keep decreasing as tile sizes increase, with an exception for very large tile sizes. The execution time follows a similar trend.
We also compare the performance of different single thread execution approaches shown in Figure 7.8 (same naming convention as for irregular reductions). *SIMD Naive* has a small speedup over *Serial* for both *Bellman-Ford* and *PageRank*. The low speedup is due to the poor memory access performance: the cache miss rates of *SIMD Naive* are much higher than those of *SIMD Tiling (Our)*. Moreover, large access ranges for the *gather/scatter* operations also negatively impact the performance of *SIMD Naive*. *Serial Tiling* has speedups of between 2.29 and 4.1 over *Serial*, clearly showing the benefit of increasing data locality with tiling. *SIMD Tiling (Our)* achieves speedups of between 4.9 and 7.5 over *Serial*.

**Overall Performance of MIMD+SIMD**

Figure 7.9 shows the overall speedup of SIMD+MIMD over the serial code. Similar with irregular reductions, smaller tile sizes scale better than larger tile sizes, due to higher degree of parallelism and better load balance among the threads. For *Bellman-Ford*, the best speedup achieved for *soc-Pokec* is 467.1, using tile size of 512 and 200 threads, whereas for *higgs-twitter*, the best speedup is 251, achieved at the tile size of 2048 and 61
threads. For \textit{PageRank}, on dataset \textit{soc-Pokec}, the highest speedup is 341, at the tile size of 128, and 244 threads. Finally, for dataset \textit{higgs-twitter}, the highest speedup is 180, with a tile size of 2068 and 61 threads.

**Matrix Reordering Optimization**

Similar to irregular reductions, we conduct matrix reordering for better non-zero distributions. According to the best performance with MIMD+SIMD executions, \textit{Bellman-Ford} obtains further speedups of 1.42 and 1.25, and \textit{PageRank} obtains speedups of 1.08 and 1.05 with \textit{soc-Pokec} and \textit{higgs-twitter}, respectively.

### 7.2.6 Results from SpMM

We evaluate the performance of SpMM using different data sets and different data types. The sparse matrices that are used include both clustered (datasets involving frequent dense
blocks) and randomly distributed non-zeros. Since the tile size is fixed for SpMM, the results from our approach are all based on $4 \times 4$ tiling.

![Figure 7.10: Single Thread Performance of SpMM with Clustered Datasets using Different Data Types.](image)

**Execution with Clustered Matrices:** Figure 7.10 shows the single thread performance of SpMM on four clustered matrices. We compare four different versions. *Serial* (single thread scalar execution, baseline), *SIMD Naive* (single thread row by row SIMD processing, as is described in Section 7.1.3), *MKL* (MKL single thread execution, which also utilizes SIMD), and *SIMD Tiling (Our)* (our tiling based single thread SIMD execution).

For float type execution, *SIMD Tiling (Our)* is much faster than the other three versions, achieving speedups of between 6.58 and 8.07, over *Serial*. Similarly, for double type executions, the speedups vary from 4.75 to 7.70. *SIMD Naive* does not show obvious speedups, and even have a slowdown for certain datasets, due to the inefficient memory access and scalar result accumulation. Moreover, *SIMD Tiling (Our)* also outperforms *MKL* by 1.50x to 2.89x, and 1.01x to 1.97x, for float and double types, respectively.
Execution with Random Matrices: We used two matrices, appu and 598a from Table 7.1 to evaluate the effect of our approach on random matrices. As is expected, obvious slowdowns of 6.48 and 3.88 were observed. This is due to that the poor clustering property leads to a significant amount of padding to the tiles, and thus very low SIMD utilization ratio.

Overall Performance of MIMD+SIMD: The scalability of multi-thread execution using SIMD is shown in Figure 7.11. The test was conducted using clustered matrices, as random matrices are not benefiting from our tiling-based optimization. Since the computation of different rows do not have data dependency, we simply parallelize the rows using OpenMP, for multi-threaded execution. The results include both float and double executions. Speedups are against the single thread scalar executions. To show the overall efficiency of our approach over commercial libraries, we plot both the speedups of our approach (in solid lines) and the the speedups of Intel MKL (in dashed lines), by varying the number of threads. For the small dataset, msc10848, our approach is able to achieve the maximum speedup of 100.64 with 61 threads for float execution, and the maximum speedup
of 111.77 with 50 threads for double execution. More threads do not benefit the overall performance, due to the limited amount of data and parallelism, as well as possible load imbalance because of the varying row lengths. Higher speedups are seen from the other three datasets. For float execution, the maximum speedup of 392.49 is achieved for dataset shipsec5, at the thread number of 200. For double execution, the maximum speedup of 334.34 is achieved using the same dataset and same number of threads. Compared with MKL, our approach is much faster, in terms of both performance with different thread configurations and peak performance. The maximum speedups of both approaches are achieved using dataset shipsec5, with 200 threads. Our approach is 2.81x and 2.14x faster than MKL, for float and double executions, respectively.

7.3 Summary

We have introduced a general methodology for SIMD parallelization of several distinct classes of irregular applications, by representing them as sparse matrix computations. Our proposed technique includes three general steps: locality enhancement, data access pattern identification, and conflict removal. Within the general framework, we are able to modify the specific steps to account for the differences between subclasses of applications. Results from detailed evaluation show that not only we obtain significant speedups, but we also outperform other recently published methods.
This chapter compares our existing work with related research efforts from other groups, especially the work of parallel frameworks or domain specific languages for specific communication patterns on GPUs or heterogeneous architectures, as well as on the coprocessors. We categorize the related work into three groups: MapReduce frameworks for heterogeneous architectures, task scheduling frameworks for parallel systems, and parallel programming systems or languages for communication patterns we have worked on.

### 8.1 MapReduce Frameworks for Heterogeneous Architectures

Recent years have seen a large number of efforts on MapReduce and its variants. We primarily focus on efforts specific to multi-core CPUs, GPUs, and heterogeneous systems.

Ranger et al. [87] have implemented a shared-memory MapReduce library named Phoenix in multi-core systems, and Yoo et al. [121] optimized Phoenix specifically for large-scale multi-core systems. CellMR [86] was a MapReduce framework implemented on asymmetric Cell multi-core processors. Streaming approach was adopted to support MapReduce. Mars [45] was the first attempt to harness GPU’s power for MapReduce applications. MapCG [47] was a subsequent implementation which was shown to outperform Mars. Catanzaro et al. [11] also built a framework around the MapReduce abstraction to support
vector machine training as well as classification on GPUs. StreamMR [35] was a MapReduce framework implemented on AMD GPUs.

MITHRA [37] was introduced by Farivar et al. as an architecture to integrate the Hadoop MapReduce with the power of GPGPUs in the heterogeneous environments, specific for Monte-Carlo simulations. Shirahata et al. [92] have extended Hadoop on GPU-based heterogeneous clusters. They enabled map tasks to be scheduled onto both CPUs cores and GPUs devices. GPMR [97] was a recent project to leverage the power of GPU clusters for large-scale computing by modifying the MapReduce paradigm.

8.2 Task Scheduling Frameworks for Parallel Systems

Outside of MapReduce, there have been other efforts on using CPU and GPU simultaneously for a single application. Qilin system [74] has been developed with an adaptable scheme for mapping the computation between CPU and GPU simultaneously. Their approach requires training a model for data distribution based on curve-fitting. Our work, in comparison, is based on dynamic work distribution. Teodoro et al. [105] describe a runtime framework that selects either of a CPU core or the GPU for a particular task.

Dynamic task scheduling has been studied for even a longer time. Guided self-scheduling scheme [85] initially allocates larger chunk sizes to reduce scheduling overhead, and subsequently reduces the chunk size towards the end of the computation. Several variations of the guided self-scheduling were developed [49, 76, 108], each with a different strategy for adjusting chunk size. Another effort [69] uses a combination of static and dynamic strategies. Our methods are specific to a special communication pattern and special architectures.
StarPU [4], proposed by Augonet et al., is a programming library with support for scheduling on heterogeneous cores. Xkaapi [40] is a runtime system for multi-core architectures based on work stealing, with recent work extending their approach to heterogeneous architectures. OmpSs [9] is an extension to OpenMP designed to ease heterogeneous programming. A recent system developed by Scogland et al. [91] automatically converts and schedules parallel loops to a CPU and a GPU. All of these works have targeted only the discrete CPU-GPU architectures, and have not been applied to integrated CPU-GPU systems, and all the scheduling policies are based on the more traditional kernel relaunch. Among these systems, we have extensively compared the performance of our framework against StarPU and OmpSs. We expect similar performance from the other two systems, because they also use kernel relaunch. Our previous work [17] proposes scheduling policies on coupled CPU-GPUs, specifically for MapReduce applications only.

OpenACC [2] is accelerated OpenMP for GPUs, based on a community standardization effort, with initial implementations released by Cray, CAPS, NVIDIA, and PGI. To the best of our knowledge, none of the existing OpenACC compilers support work distribution across CPU and GPU cores. Similarly, while OpenCL code can be executed on either the CPU or the GPU, it cannot simultaneously use both. A research prototype has been developed for translating OpenMP codes for GPU execution [67], but does not provide support for using CPU and GPU cores at the same time. Chen et al. propose a task-based dynamic scheduling framework [15] for GPUs. They use a queue for the CPU to insert tasks to the GPU(s). Their method is similar with the master-worker method that we have designed, although we support simultaneous use CPU and GPU cores, which their framework does not allow. A few locking-free task scheduling techniques across CPU and GPUs are proposed in [18].
Becchi et al. [5] use a data-aware scheduling of kernels that reduces the data transfer overhead due to GPU. Song et al. [94] propose a scheme for partitioning input for matrix applications on heterogeneous clusters. Our system is not limited to matrix applications and is targeted to diverse intra-node heterogeneous architectures.

Though our direct comparison (for distributed memory parallelization) has been against MPI, PGAS models have also become popular for scalable parallel programming. Models like Global Arrays [81] and UPC [26], to the best of our knowledge, do not support execution on GPU clusters, and cannot partition work across CPU and GPU cores. Languages that are based on more advanced compiler support, such as X10 [13] and Chapel [93], are more likely to be able to handle parallelism at multiple levels. For example, X10 has support for GPUs (CUDA), though not completely transparent to the programmer [27]. Overall, PGAS model implementation continues to be an area of very active research [93, 107, 38].

8.3 Parallel Programming Systems and Languages for Communication Patterns We Have Worked on

Recent years have seen an increasing number of efforts on domain specific languages, including many of them being implemented on GPUs and GPU clusters. Liszt [33] is a domain specific language for mesh-based PDE solvers, similar to irregular reductions in our system. Their framework is designed for CPU clusters, multicore architectures, and GPUs, and thus is portable across platforms. However, it cannot accelerate computation using both CPU and GPU cores. Similarly, OP2 [80] is a library designed for accelerating unstructured mesh applications on either the multi-cores or many-cores, but not both together.
Physis [77] is a parallel programming model designed for stencil computations over GPU clusters. Though it runs on heterogeneous clusters, it only utilizes the GPUs for computation. Mint [109] is a source to source compiler for translating annotated 3D stencil C code into optimized CUDA code. It can support execution on a single GPU only. Zhang et al. proposed a framework for code generation and performance tuning for 3D stencil applications on GPU clusters [122]. Similar with Physis, although it was designed for distributed execution, only GPU cores, and not the CPU cores, are exploited. Compared with these efforts, our contributions are in exploiting both CPU and GPU cores, and that our programming model can also enable an application that involves multiple distinct communication patterns.

There have been many efforts on parallelizing various graph applications on parallel systems. The closest work to ours, Pregel [75] – a programming model proposed by Malewicz et al., provides programmers a general API, which has been used in our design as well. Existing implementations of Pregel, as well as some of the other parallel graph libraries such as the Parallel Boost Graph Library [44], and CGMgraph [12] have been on distributed memory systems, which involve very different issues as compared to a many-core system like Xeon Phi. MTGL [6] is another parallel graph library, which is on shared memory system, however, it focuses on graph query algorithms.

Galois project [78, 62] extensively studies and automatically explores amorphous data-parallelism present in irregular algorithms including graphs, and it can execute Galoized serial code in parallel on shared-memory machines. Our work cannot handle data-driven or speculative parallelism, but focuses on issues associated with large-scale shared memory and SIMD parallelization for data-parallel graph applications.
Recently, there have also been many efforts focusing on parallelizing graphs and trees applications on SIMD accelerators and GPUs. Merrill et al. [79] parallelize Breadth-first Search, an important graph traversal algorithm, on the GPUs architecture by focusing on fine-grained task management. This work shares some similarities to our work, however, is not aimed towards providing a general system for graph applications. Hong et al. [48] developed a novel virtual warp-centric programming method to address the work imbalance problem in graph algorithms, but again, is not providing a general system. Jo et al. [56] designed a novel scheduling mechanism for efficient SIMD execution of tree traversal applications, and Kim et al. [59] designed FAST, an architecture-sensitive layout of the index tree on both CPU with SIMD and GPU architectures. Both of these efforts focus on tree structures, which involves different characteristics (especially from the layout reorganization consideration) than our target class. Most closely related to our work, Zhong and He [123] proposed a programming framework, Medusa, to parallelize graph processing applications on GPUs by adapting Pregel’s design. They focus on issues with SIMT architecture, whereas our design is for exploiting the more rigid SSE-like parallelism as well as larger-scale MIMD (shared memory) parallelism.

Recently, several efforts have also parallelized irregular applications on the Xeon Phi architecture, and thus share some similarities with our work. Saule and Catalyurek [90] provided a preliminary evaluations on graph applications on the Xeon Phi architecture, but did not report a system design. Liu et al. [72] parallelized Sparse Matrix-Vector Multiplication, and Pennycook et al. [84] parallelized Molecular Dynamics on Xeon Phi, with emphasis on long vector SIMD parallelization. Our work has focused on supporting a system, with a general and high-level API, for a set of graph processing applications on the same architecture.
8.4 Optimizing Irregular Applications on Diverse Parallel Platforms

Many efforts in recent years have focused on accelerating irregular applications on various parallel platforms, including multi-core machines, GPUs, as well as vector hardware. Some of these studies are focusing on specific applications, and a few are focusing on proposing a generalized approach for a subclass. Common approaches for improve data locality, such as edge or matrix reordering and even tiling have been used, though in very specific context.

**Application-specific Solutions:** Strout *et al.* [96, 95] improved the data locality for irregular kernels like Gauss-Seidel through rows/columns permutation and tiling, on traditional CPU architectures, but did not consider SIMD parallelism. Pennycook *et al.* implemented Moldyn on the Xeon Phi, using both MIMD and SIMD parallelism [84]. This is a very initial investigation, mainly analyzing the main bottleneck, as well as what performance could be delivered, using a relatively straightforward implementation specific to a single application. Iwashita *et al.* [52] proposed an approach that uses blocked coloring scheme to accelerate ICCG solver for multi-threaded execution. Their work was specific for ICCG and does not involve SIMD parallelism. Park *et al.* [83] utilized the same coloring approach to vectorize the computation of CG. Similarly, Thébault *et al.* used a similar approach for vectorizing unstructured 3D mesh computations.

**Studies Targetting Subclasses:** Recent work for optimizing graph processing using SIMD-like instruction set have mostly in the context of GPUs. Merrill *et al.* [79] parallelize breadth-first search on the GPUs by focusing on fine-grained task management. Hong *et al.* [48] developed a novel virtual warp centric programming method to address the work imbalance problem in graph algorithms for GPUs. CuSha [58] optimized graph processing on GPUs with intensive usage of shared memory, by re-organizing the graph data in shards.
Saule and Catalyurek [90] provided a preliminary evaluation on graph applications on the Xeon Phi architecture, but without particular optimizations specific to the applications and the hardware. Choi et al. [24] proposed a way of optimizing SpMV on GPUs by storing sparse matrices into small subblocks, each represented as a dense matrix. Their method aims to reduce the amount of column indices stored for the non-zeros, and thus increasing the computation to memory access ratio. Tang et al. [104] utilized a hybrid storage format with jagged partitioning to optimize SpMV, also for the Intel MIC. The system [20], developed in our research group, deals with the issue of efficiently parallelizing graph applications across CPU and Intel MIC, with a very easy-to-use programming API. [19] approaches the optimization of applications on a CPU-GPU cluster by classifying the applications based on communication patterns. A set of pattern-specific programming APIs and runtime optimizations are automatically conducted.

**Compiling for SIMD:** Compilation for automatically generating SIMD code has been a topic of much investigation, though the focus has been almost entirely on regular applications. For example, Henretty et al. [46] designed a domain-specific language and compiler for vectorizing stencil computations. To the best of our knowledge, the only existing work on compiling irregular application on SIMD instruction sets is from Kim et al. [61], who focused on irregular applications. Their target hardware is the Cell processor, with vector unit that does not support modern SIMD operations such as gather/scatter. Their approach involves software based gather/scatter support, through packing and unpacking. By conducting benefit-cost ratio analysis, they determine if a certain part of the code is worth the packing/unpacking.

On the more theoretical side, Wu et al. [118] concluded that searching an optimal data reordering (in terms of minimizing non-coalesced data access on GPUs) for irregular data
references is an NP-complete problem. Burtscher [10] et al. quantitatively studied the irregularity in the applications and showed that code optimizations are able to reduce the irregularity from either memory level or control-flow level.
Chapter 9: Conclusions

This is a summary of the contributions of our current work and a description of our future work.

9.1 Contributions

Our main objective is to investigate applications based on communication patterns. On different parallel architectures, we design pattern-specific optimizations. At the same time, we provide an API for each communication pattern, in order to simplify the programming. Our contributions can be summarized as following:

- Our initial efforts optimizes MapReduce for GPUs, with efficient utilization of the on-chip shared memory. By applying reduction based manner, we designed a compact data structure, reduction object, for storing intermediate reduction results over the memory hierarchy. Our method has been proved to significantly outperform traditional MapReduce frameworks on GPUs.

- We further extend our MapReduce work to a new coupled CPU-GPU. In this work, we designed efficient locking-free scheduling schemes for distributing work between CPU and GPU, with effective usage of the zero-copy memory. The distinct scheduling methods we propose include map-dividing scheme and pipelining scheme. We
have evaluated different scheduling schemes using different applications, showing under which case each scheduling method works best.

- Based on the observation of the common properties of heterogeneous architectures, we propose several general scheduling methods for parallel architecture with heterogeneous cores. These scheduling methods include master-worker model, core-level token passing model and device-level token passing model. All these scheduling methods are locking-free and the device-level token passing model also has a relatively less scheduling overhead.

- We constructed a programming framework for a heterogeneous cluster with a multi-core CPU and many-core GPUs on each node. Different from traditional programming models like MapReduce, we provide separate APIs for each communication pattern. We conduct intensive underlying pattern-specific optimizations for each communication pattern, at both intra-node level and inter-node level. Our programming model is similar with MapReduce, in the way that users only need to write serial code for processing a minimum input unit for a specific pattern code.

- We have built a graph processing system for a heterogeneous node comprising a CPU and a relatively new coprocessor (represented by the Intel Xeon Phis). Our system follows a similar programming model with Google Pregel, offering a very simple programming API. Our system is featured by a very novel design of the message buffer, which supports SIMD message processing, as well as reasonable memory consumption. We also propose a pipelining execution strategy, which greatly lowers the locking overhead. The hybrid graph partitioning is able to keep the load balance between CPU and Xeon Phi, as well as keep the communication overhead low.
Lastly, in view of the challenges of accelerating irregular applications on SIMD architectures, we propose a general methodology for efficient acceleration of irregular applications on the Intel Xeon Phi. Our approach is based on the observation that most irregular applications can be treated as sparse matrix based computations. Our approach consists a few general steps: matrix tiling to improve data locality, access pattern identification, and grouping to avoid write conflicts. Our approach is able to achieve considerable scalability at both SIMD and MIMD levels.

9.2 Future Work

Based on the work we have conducted so far, we can extend our work in a few directions. The first possible direction is to investigate irregular applications with adaptive change of the irregular data structures, which is very common in scientific simulations. Accelerating such applications on SIMD architectures needs to consider the overhead of reorganizing the changing data structures. The second direction is to make use of the new SIMD architectures, e.g., the second generation MIC architecture, *Knights Landing*.

9.2.1 Supporting Irregular Applications with Adaptive Data Structures on SIMD Architectures

Lots of irregular applications involve the update of interaction lists among particles. The interactions among particles can change with the simulation going on. If we still want to use the method described in Chapter 7, we have to apply our optimization steps repeatedly to the varying matrices. This would lead to a very significant overhead. The points we need to observe are:
• First, the data organization is serial in our current implementation, which takes quite a long time to complete. Thus it is worthwhile to reduce the data reorganization time, through parallelization.

• Second, the change of the indirect data structures is usually adaptive, which means, only a small portion of the non-zeros change between different interaction rebuilds. Thus it is critical that we do not re-do the reorganization for all the elements.

• Finally, an automatic parameter selection mechanism could be developed. This is possible to be implemented using performance prediction, by analyzing the property of the input, as well as the pattern of the computation.

9.2.2 Supporting the New MIC Architectures

The development of the parallel platforms is always fast. For example, the Intel MIC architectures are advancing to the second generation. A series of optimizations and new designs will appear on the new hardware. Accordingly, at the software level, our optimization of the applications should take advantage of the new hardware features. The following are the new features that we need to consider (using the Knights Landing as example):

• Higher memory bandwidth. The total parallel bandwidth will be up to 400 GB/s.

• Out-of-order execution will be supported by the X86 cores. This hardware optimization will significantly reduce the CPI of programs.

• The AVX512 instructions will add extra flexibility to the SIMD programming. This means that more complex computation patterns will be able to be accelerated by SIMD.

These new features will address more challenging problems in parallel processing.
At the same time, research on accelerating applications using MPI and Intel MIC over big data is being broadly conducted [99, 100, 116, 103, 102, 101, 98], also, big data analysis over different platforms with different data management formats are also popular [114, 113, 115, 117, 124, 22, 23]. The works have been focusing on different aspects including data transfer, data virtualization, data indexing, as well as correlation analysis and correlation mining. It is worthwhile to combine these techniques with my dissertation work. Some important sets of data mining algorithms, are also open to the acceleration of parallel platforms [71, 70].

The problems that have been investigated in this dissertation work are limited to the utilization of hardware resources for different sets of applications, and do not involve the issues such as system failures and fault tolerance. A significant amount of efforts have been made in this area [64, 21, 63, 65]. These are very important aspects to be considered when an application is being implemented on a cluster. Our future work would include the development of different fault tolerance schemes for application subsets.
Bibliography


202


[79] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU Graph Traversal. PPoPP ’12. ACM.


204


207
