12-2018

Framework for Lifecycle Enrichment of HPC Applications Towards Exascale Heterogeneous Architectures

Karan Sapra
Clemson University, karan.fui@gmail.com

Follow this and additional works at: https://tigerprints.clemson.edu/all_dissertations

Recommended Citation
https://tigerprints.clemson.edu/all_dissertations/2276

This Dissertation is brought to you for free and open access by the Dissertations at TigerPrints. It has been accepted for inclusion in All Dissertations by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.
FRAMEWORK FOR LIFECYCLE ENRICHMENT OF HPC APPLICATIONS TOWARDS EXASCALE HETEROGENEOUS ARCHITECTURES

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Engineering

by
Karan Sapra

Dr. Melissa C. Smith, Committee Chair
Committee Members:
Dr. Alex F. Feltus
Dr. Richard R. Brooks
Dr. Walt B. Ligon III
Abstract

With the advent of accelerators and architectures, researchers are faced with a daunting task to port their existing applications and algorithms to the optimal architecture and programming language. Porting existing applications or a new algorithm is both demanding and time-consuming due to the sheer number of accelerators and architectures plus the number of programming models available per architecture. This problem is further compounded for heterogeneous systems with wide availability of resources and complexity of scientific applications. In this dissertation, we focus on enriching the lifecycle of applications by providing an application to optimal architecture mapping and framework to assist in making the most effective use of resources in a heterogeneous environment. Our Application to Architecture (A2A) framework can be further divided into sub mappings: Qualitative and Quantitative. Our qualitative mapping uses benchmark application analysis to understand the application performance without in depth runtime analysis and is highly suitable for new algorithms and applications. Our quantitative mapping can provide detail numerical performance analysis for porting an application across programming models and architectures. We evaluate our overall framework using various diverse benchmark applications. Lastly, our Heterogeneous Partitioning Framework (HFP) provides existing and new applications the ability to use heterogeneous resources in an efficient manner with minor modifications to the source code in comparison to other frameworks as is shown with two case studies: Climate Earth Science Model (CESM) and GPU-based Gene Network Alignment Tool (G3NA).
Dedication

I dedicate this thesis to my wife, Benafsh, my mother, my academic advisor, Dr. Smith and my little brother, Yash without whose constant support and guidance this thesis would not have been possible.
Acknowledgement

This thesis was made possible by the help and support of my committee members, my wife, Benafsh, colleagues and the NSF Funding support. Firstly, I would like to thank my academic advisor Dr. Melissa Smith, who guided and supported me. Dr. Smiths action, support, words of wisdom, and patience allowed for the completion of this research and thesis. I would also like to thank Alex F. Feltus, Dr. Walter B. Ligon, and Dr. Richard R. Brooks for serving on my committee and instructing me in various aspects of computer architecture and parallel computing. A special thanks to Dr. Feltus for pushing me and introducing me the field of biogenetics. I am also thankful for my mother and my brother, Yash who provided constant support during my research and writing this manuscript. Lastly, I would like to acknowledge my lab mates, specially Ashrit and Benafsh for constant interaction and brainstorming that allowed us to do better research.
# Table of Contents

Framework for Lifecycle enrichment of HPC Applications towards Exascale Heterogeneous Architectures ........................................... i

Abstract .................................................................................. ii

Dedication .................................................................................. iii

Dedication .................................................................................. iii

Acknowledgments ....................................................................... iv

Acknowledgement ....................................................................... iv

List of Tables ........................................................................... vii

List of Figures ........................................................................... ix

1 Introduction ........................................................................... 1
   1.1 Motivation ........................................................................... 1
   1.2 Proposed Research ............................................................... 4
   1.3 Method of Study ................................................................... 5
   1.4 Dissertation Outline ............................................................. 5

2 Background ........................................................................... 7
   2.1 Heterogeneous Architecture ................................................. 7
   2.2 Accelerators ......................................................................... 7
   2.3 Applications and Micro-benchmarks ...................................... 9
   2.4 Accelerator based Supercomputers ...................................... 11
## List of Tables

1.1 Programming Language Specification and Characteristics. .......................... 3

1.2 General characteristics of Intel Xeon, Intel Xeon-Phi and Nvidia GPU. ........ 9

1.2 Overview of current and upcoming supercomputers. ............................... 12

4.1 Performance Impact Factors ..................................................................... 24

4.2 Classification of Algorithmic Classifiers. .................................................. 26

4.3 Algorithm Classification and Mapping to Single-(G)PU, Single-(P)hi and Single-(N)ode. ................................................................. 29

4.4 Prediction model test case implementations. ............................................. 37

4.5 Hyperparameter values of our neural network for our quantitative model. ..... 38

4.6 Minimum, average and maximum mean absolute prediction error using Performance API. ............................................................... 40

4.7 Minimum, average and maximum mean absolute prediction error using CUDA Profiling Tools Interface. ............................................. 41

4.8 Minimum, average and maximum mean absolute prediction error using Performance API Hardware Counters. ................................. 42

4.9 Minimum, average and maximum mean absolute prediction error using Performance API Hardware Counters. .................................. 43

4.10 Run time prediction results (in seconds) for real life application using Hardware Counters collected from four different platform. ....................... 47

5.1 Basic Function Definition of HFP. ............................................................... 52

6.1 Edge Matching between G3NA and IsoRankN. ......................................... 75

1 Hardware counters used to train our CPU-C model. .................................... 90
List of Figures

1.1 Comparison of K-means performance for different programming models and different architectures. (1600K items and 20 clusters) ............................................. 4

2.1 Heterogeneous supercomputer architecture overview. ................................. 8
2.2 Heterogeneous supercomputer architecture overview. ................................. 13

4.1 Tesseract: A2A Mapping. ................................................................. 27
4.2 Linear Algebra ................................................................. 31
4.3 Divide and Conqueror ......................................................... 32
4.4 N-Body ................................................................. 33
4.5 Grid Models ................................................................. 33
4.6 Dynamic Programming ................................................................. 33
4.7 Branch and Bound Algorithms. ................................................................. 34
4.8 Logic Algorithms ................................................................. 34
4.9 Graphical Models ................................................................. 35
4.10 Sorting Algorithms ................................................................. 35
4.11 Mean Absolute Prediction Error Performance API Hardware Counters. .... 40
4.12 Mean Absolute Prediction Error using CUPTI Hardware Counters. .......... 41
4.13 Mean Absolute Prediction Error using Performance API Hardware Counters. .. 42
4.14 Mean Absolute Prediction Error using Performance API Hardware Counters. .. 43
4.15 Mean Absolute Error for run time prediction of test set applications. ........... 44
4.16 Cumulative distribution function of prediction confidence for our qualitative prediction using Performance Hardware Counters. ................................. 45
4.17 Prediction accuracy of our Qualitative Model using Hardware Counters from four different platform. ................................................................. 46
5.1 Titan Supercomputer Node Structure. Each node has AMD CPU and NVIDIA K20X GPU. AMD Opteron 6724 has two NUMA domains, 8 pairs of integer cores, each pair shares a floating point core.

5.2 HFP Design Overview

5.3 HFP Runtime Overview

5.4 Variation in MIPS for overlapping task per core on Titan Supercomputer.

5.5 Runtime for varying number of ranks with range of base application (ICPU, DCPU, MEM, IO micro-benchmarks) and post-processing micro-benchmarks.

5.6 Incremental Performance improvement with benchmark applications on Titan at ORNL.

5.7 Result Evaluation of LULESH on Titan using MPI and MPI-CUDA version with and without HFP.

5.8 Hydrocube mesh of LULESH representing the speed variable at various time steps.

6.1 G3NA versus Magna++ versus IsoRankN alignment of Medium Sized Graphs.

6.2 Distribution of percentage of time spent per operation in G3NA.

6.3 G3NA alignment of random scale-free graphs.

6.4 G3NA performance on various architectures.

6.5 **BioDep-Vis workflow.** Each panel illustrates a data visualization component including gene expression distribution, gene expression network, network alignment, molecular structure visualization, subgraph, ontology graph, and evolutionary relationship graph visualization.

6.6 GPU runtime of image processing algorithms.

6.7 GPU runtime for varying image sizes.

6.8 CESM workflow runtime using 320-nodes on Titan Supercomputer.
Chapter 1

Introduction

In this chapter, we will introduce motivation of our work in Section 1.1 followed by our proposed research and method of study in Section 1.2 and Section 1.3 respectively.

1.1 Motivation

Massively-parallel architectures such as the General Purpose Graphical Processing Units (GPGPUs), multi-core architectures, Field Programmable Gate Arrays (FPGAs) and more recently, the Xeon-Phi coprocessors have established a stronghold in High Performance Computing (HPC). The Scientific community is accepting these architectures with open minds allowing for a wide adoption by porting code for optimal use on these architectures. These architectures have enabled substantial performance improvement for applications that normally take several hours or more to execute on traditional sequential processors. The recent supercomputers and data clusters are heterogeneous and divergent, for example, Cori at NERSC’s (National Energy Research Scientific Center) is based on the new Intel’s Knight landing (aka Intel Xeon-Phi). Oak Ridge National Laboratory’s Summit, the successor to Titan, and currently ranked #1 supercomputer will be based on NVIDIA’s Volta architecture. Sierra at Lawrence Livermore National Laboratory is also a NVIDIA Volta based supercomputer. Microsoft’s new data cluster has adopted Altera based FPGAs. These machines indicate a trend towards heterogeneous architecture and provide the scientific community with access to rich and diverse resources along frequent 3 to 4 year refreshes to the architecture and supercomputers available to community.
The wide range of high-performance architecture choices available present the HPC community with the challenge of selecting the most optimal architecture for their application to obtain the best performance. The performance of an application is generally measured by metric such as runtime, speedup, and power consumption. Grozea et al. [1] and Bhuiyan [2] show that significant performance improvement is achieved when a given application is mapped to an appropriate architecture corroborating that a sub-optimal Algorithm-to-Accelerator (A2A) mapping leads to unsatisfactory performance. The authors used several algorithms: Merge sort [3], Bitonic sort [4], Insertion sort [5]; and Spiking Neural Network (SNN) models: Hodgkin-Huxley [6], Izhikevich [7], Morris-Lecar [8], and Wilson [9] respectively to draw the above conclusion.

This phenomena of sub-optimal performance for an application arises due to the substantial difference in architectural design of the available compute resources and the optimization techniques employed. Consequently, certain applications are better suited for specific classes of architectures; for instance, highly data parallel and compute intensive applications are mapped optimally on GPUs [10]. Similarly, an application taking advantage of pipelined execution is suitably ported on an FPGA architecture [11].

Heterogeneous systems provide substantial potential for performance gain for a wide range of applications; however, much of their computing resources are often under-utilized due to non-optimal A2A mapping. This inefficiency leads to poor application speed-up, sub-optimal scaling efficiency, long queue delays, and increased power consumption. Bharathi et al. [12] show that a scientific application is the sum of multiple sub modular algorithms. Thus, using a heterogeneous supercomputer, such as Stampede [13] or Cori [14] which tend to provide accelerator heterogeneity, allow for optimal utilization of resources for work-flow and accommodate a variety of scientific applications. We can thus accelerate the performance of applications on these heterogeneous supercomputers.

Furthermore, due to the programming and implementation disparity, hardware bottlenecks and optimization techniques varying from architecture to architecture; substantial development time is required to build an optimized and accelerated implementation of an application for each platform. Figure 1.1 shows the performance gain achieved by implementation of K-Means application for 1600K items and 20 clusters on different architectures using variety of programming models. The figure illustrates that significant performance gain can be obtained by using dedicated programming models (i.e. those programming models that are tightly coupled with the architec-
Explicit thread creation Abstraction level Parallelization form Strictly lockstep style execution Vectorization Programming effort (subjective) Architecture

<table>
<thead>
<tr>
<th></th>
<th>Explicit thread creation</th>
<th>Abstraction level</th>
<th>Parallelization form</th>
<th>Strictly lockstep style execution</th>
<th>Vectorization</th>
<th>Programming effort (subjective)</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-threads</td>
<td>Yes</td>
<td>Low</td>
<td>SIMD MIMD</td>
<td>No</td>
<td>No</td>
<td>Medium</td>
<td>Intel Xeon</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Intel Xeon-Phi</td>
</tr>
<tr>
<td>OpenMP</td>
<td>No</td>
<td>High</td>
<td>SIMD MIMD</td>
<td>No</td>
<td>No</td>
<td>Low</td>
<td>Intel Xeon</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Intel Xeon-Phi</td>
</tr>
<tr>
<td>Concurrency Runtime</td>
<td>No</td>
<td>High</td>
<td>SIMD MIMD</td>
<td>No</td>
<td>No</td>
<td>Low</td>
<td>Intel Xeon</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Intel Xeon-Phi</td>
</tr>
<tr>
<td>OpenCL</td>
<td>Yes</td>
<td>Low</td>
<td>SIMD</td>
<td>Yes</td>
<td>Yes</td>
<td>High</td>
<td>Intel Xeon</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Intel Xeon-Phi</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FPGA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GPU</td>
</tr>
<tr>
<td>CUDA</td>
<td>Yes</td>
<td>Low</td>
<td>SIMD</td>
<td>Yes</td>
<td>No</td>
<td>High</td>
<td>GPUs</td>
</tr>
</tbody>
</table>

Table 1.1: Programming Language Specification and Characteristics.

However, the optimal performance for an architecture remains with the programming model native to accelerator the despite the various programming model. Table 1.1 shows the disparity and wide availability of programming languages available for different architectures. On the contrary, the benefit provided by using cross-platform programming models across architectures allow scientific user portability and debugging execution across various supercomputers.

Applications also differ in behavior ranging from control intensive applications, such as sorting to data intensive applications such as simulations, modeling etc. Data intensive behavior can be further classified into three sub-categories: compute-, memory-, and IO-intensive workloads. Compute-intensive workload throughput can be defined by computational efficiency of the underlying hardware, while memory-intensive throughput greatly depend on the memory bandwidth and memory-accesses while the IO-intensive throughput is related to the IO bandwidth. However, there is no single architecture suitable for optimal performance for all ranges of workload behavior. Furthermore, most scientific applications are a combination of various workloads, making optimal architecture a factor of the application and workload characterization.
Figure 1.1: Comparison of K-means performance for different programming models and different architectures. (1600K items and 20 clusters)

1.2 Proposed Research

In this research, we aim to create a performance modeling framework to aid in the development and porting of applications and algorithms to supercomputers with heterogeneous architectures. We achieve this in two stages: development and runtime. In the development stage, we utilize Algorithm-to-Accelerator (A2A) mapping to correctly map an algorithm to the most appropriate accelerator. We construct two types of A2A models: a qualitative model and a quantitative model. The qualitative model, Tesseract, is a 4-dimensional A2A mapping. Tesseract provides an optimal A2A mapping by matching an algorithm from the algorithm space to an architecture in the architecture space. The items in algorithm space of A2A are classified based on the performance; this classification is performed by mapping algorithms to appropriate regions of the architecture space using a one-to-one mapping. To evaluate Tesseract, we survey 20 unique applications in 22 different experiments, encompassing a broad spectrum of application characteristics (performance dimensions), ergo highly suitable for verifying the proposed A2A mapping. These applications are surveyed and mapped onto the following three architectures: Single Xeon-Phi co-processor (Single-Phi), Single GPU (Single-GPU), and Single Multi-Core processor (Single-Node) architecture. The mapping results are then used with a clustering algorithm to rank the performance dimensions and provide a multi-node extension of the A2A mapping. The quantitative model utilizes multi-variate factors.

In the second stage, we propose the Heterogeneous Functional Partitioning (HFP) runtime
framework that leverages the under-utilized resources on heterogeneous nodes of a supercomputer using A2A mapping results. We discuss key design considerations for such a framework and our approach to accelerating the end-to-end workflow of scientific applications. We propose and evaluate several features and optimizations that make the HFP framework efficient, high performing, and low in performance variation (jitter) for the main application. We showcase HFP integration with the Climate Earth Science Model (CESM) [15] to enable post-processing of precipitation data using the GPUs with negligible overhead. We also discuss how other services like I/O caching, checkpoint encoding, and incremental analysis in ensemble applications can leverage our HFP interface.

1.3 Method of Study

The diverse set of micro-benchmarks from various scientific fields provides a strong foundation for evaluation of both the qualitative and quantitative A2A mapping for the development stage. We further use our strategy to develop new algorithms and applications in collaboration with Feltus Lab [16] and CUTTERs Lab [17] in System Genetics Department and Bioengineering Department at Clemson University. We then implement a runtime management framework HFP as part of our aforementioned collaborative work to take advantage of heterogeneous supercomputers in collaboration with Technology Integration Group at Oak Ridge National Laboratory (ORNL). We evaluate our framework first on CESM, which utilizes a large portion former Titan and now Summit Supercomputer at ORNL. We also implement our collaborative work at Clemson University, i.e. Global GPU-based Gene Network Alignment software and HPC enabled Laproscopic Surgery with HFP using the Palmetto Supercomputer.

1.4 Dissertation Outline

Chapter 2 provides background on the heterogeneous architecture, accelerators as well as background on the micro-benchmarks we utilize. Chapter 3 provides a literature review of important related work for heterogeneous computing and case study applications. Chapters 4 and 5 explain in detail our qualitative A2A framework and HFP framework for efficient utilization of heterogeneous computing environment. In Chapter 6, we showcase utilization of these frameworks for three different classes of applications namely, Large Scale Gene Network Alignment using
GPGPUs (G3NA), Real-Time Remote Processing of Laparoscopic Surgery, and Climate Earth Science Model (CESM). Chapter 7 provides the results and analysis for the A2A and HFP frameworks. The dissertation proposal is concluded in Chapter 7 with conclusions and a research plan for the remainder of the dissertation work.
Chapter 2

Background

In this chapter, we introduce to the readers the different heterogeneous architectures and accelerators and how they are utilized in leadership machines such as Summit at Oak Ridge National Lab, Mira and Theta at Argonne National Lab and others. We then introduce the different classes of benchmark algorithms that we utilize for the evaluation of the A2A mapping and HFP framework.

2.1 Heterogeneous Architecture

A heterogeneous supercomputer architecture is defined to have one or more accelerators connected via high-speed interconnect. Scientific applications can utilize various programming models to employ these accelerators for application performance gain. In this chapter we provide an overview of heterogeneous architectures, accelerators, various programming models, and current/upcoming supercomputers.

2.2 Accelerators

In this section we describe the three architectures under consideration: Intel Xeon E5 (Single-Node), Intel Xeon-Phi (Single-Phi), and NVIDIA K20 (Single-GPU). Table 2.1 shows the summary of the architectures.
2.2.0.1 Intel Xeon-Phi E5110P

The current generation of Xeon-Phi Coprocessors reside on the PCI Express (PCIe2) bus consisting of 61 processor cores running at 1.238Ghz and interconnected via a bi-directional ring with a theoretical peak of 352GB/s using 8 memory controllers each with 2 channels of 5.5GT/s. Each core has 32KB Instruction and 32KB Data L1 cache, 512KB shared cache, and 16GB DDR5 global Memory. In Offload mode the user has the benefit of targeting specific computation onto dedicated architectures using the available offload directives. In the Symmetric mode of operation, MPI processes uniformly span the domains of both the host and the coprocessor architectures with explicit management of parallelism across the two different architectures. The Coprocessor-only mode is a subset of Symmetric mode with all MPI processes being confined to the Xeon-Phi architecture alone.

2.2.0.2 Nvidia K20 GPU

The Nvidia K20 Kepler Series GPUs are also accessible via the PCIe2 bus consisting of 2688 cores and a core clock speed of 732 MHz. The K20 architecture has 6GB DDR5 global memory available at 250GB/s bandwidth. Each core has L1 cache and L2 data cache available in three configurations (16KB L1/48KB L2, 32KB L1/32KB L2, and 48KB L1/16KB L2).
<table>
<thead>
<tr>
<th></th>
<th>Intel Xeon-E5 2680</th>
<th>Intel Xeon-Phi E5110P</th>
<th>Nvidia Tesla K20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance (GFLOPs)</td>
<td>333</td>
<td>2022</td>
<td>3935</td>
</tr>
<tr>
<td>Memory Bandwidth (GB/s)</td>
<td>51.2</td>
<td>320</td>
<td>250</td>
</tr>
<tr>
<td>Memory Size (GB)</td>
<td>32</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>8</td>
<td>61</td>
<td>2688</td>
</tr>
<tr>
<td>Clock Speed (GHz)</td>
<td>2.6</td>
<td>1.238</td>
<td>0.732</td>
</tr>
</tbody>
</table>

Table 2.1: General characteristics of Intel Xeon, Intel Xeon-Phi and Nvidia GPU.

2.2.0.3 Intel Xeon E5-2680

Intel E5-2680 processors are multi-core, Intel Hyper-Threading Technology (HT) enabled designs. Each socket has eight cores running at 2.6 GHz each, which share L3 cache, a local integrated memory controller connected via an Intel QuickPath interconnect. The cores share 32GB DDR3 global memory with 32KB data and instruction L1 cache, a 256 KB unified L2 cache, and 8 MB L3 Cache.

2.3 Applications and Micro-benchmarks

We utilize the following algorithms from various sources including the OpenDwarf Benchmark [18], Rodinia Benchmark [19], and Scalable Heterogeneous Computing Benchmark Suite (SHOC) [20].

1. **Linear Algebra** consists of dense and sparse linear algebra algorithm. It has high ratio of
math to memory operations and a high degree of data relation between work threads. We evaluate in this category: K-means, Lower Upper Decomposition (LUD), SPMV, and Triad. For K-means, we vary the data size, cluster size and the dimensions. For SPMV, we vary the input matrix size.

2. **Divide and Conquer** consists of algorithms that are data parallel in nature. These algorithms also consist of multiple data loads and divergence in threads along with spectral data pattern. We evaluate four Divide and Conquer algorithm: Reduction, Fast-Fourier transform (FFT), Scan and Stencil-2D (Stencil).

3. **Grid** has two categories of algorithms: Structured Grid and Unstructured Grid. Structure grid algorithms perform operations such as updating regular multidimensional arrays, where neighboring elements are required for most operations. Unstructured grid however consists of data structures that keep track of the location and neighborhood of points that are used to update the location and thus require multiple levels of memory reference indirection. We evaluate two grid algorithms: Speckle Reducing Anisotropic Filter (SRAD) as a structured grid and Computational Fluid Dynamics (CFD) as the unstructured grid.

4. **N-Body** problems can be considered a subset of grid algorithm. N-Body problems consist of multiple nodes that influence each other. Thus, to solve an N-body problem, the algorithm evaluates the impact of all nodes on each other. These problems have large unit stride memory access, along with high FLOPs to Non-FLOPs ratio. For this classes we consider two applications GEM and LAVAMD. GEM is biomolecular visualization and electrostatic potential computation and representation application while LAVAMD calculates particle potential and relocation due to mutual forces between particles within a large 3D space.

5. **Logic** are algorithms with high level bit-level parallelism and typically produce high throughput. We evaluate three such algorithms: Time-Division Multiplexing (TDM), Cyclic Redundancy Check (CRC), and Message-digest algorithm (MD5).

6. **Dynamic Programming** solves a complex problem by solving a series of simpler subproblems. We evaluate Smith-Waterman (SWAT) and Needleman-Wunsch (NW), two algorithms that belong to this category.
7. **Branch and Bound** algorithms perform large space search for an optimal solution using recursive and implicit methods. These algorithms can sometimes fall under Dynamic Programming but differ due to the search space. We evaluate A-Star (A*) as an algorithm belonging to this classification.

8. **Graph Models** map graphs into variables with edge and probabilities. We evaluate Hidden Markov Model (HMM) by varying the number of observations.

9. **Graph Traversals** perform random access to graph node that are mapped into a structured list. We evaluate our frameworks using one such algorithm, Breadth First Search (BFS).

10. **Sort** algorithms usually exhibit sequential and continuous memory performance. We evaluate one of the popular sorting algorithms, Radix Sort.

### 2.4 Accelerator based Supercomputers

HPC-enabled supercomputers are reaching exascale computing capabilities using accelerators such as multi-cores, GPUs and many-core architectures. Upcoming supercomputers such as Summit at Oak Ridge National Laboratory and Sierra at Lawrence Livermore National Laboratory will be exascale leadership-class computing systems for open science. The upcoming supercomputers are heterogeneous and distinctly diverse from each other as well as their predecessors. Table ?? shows a summary of upcoming supercomputers and their predecessors. We observe a strong increase in performance and performance per watt compared to predecessors. Along with a rise in the peak performance, a strong emphasis is the availability of accelerator(s) as well as higher memory capacity and bandwidth.

Leadership machines such Titan offer heterogeneous resources for large scaling applications. Titan is a heterogeneous mixture of AMD CPUs with NVIDIA GPUs per node. Titan’s AMD CPUs as shown in Figure 5.1, contains 2 NUMA domains each with 8 compute cores and 32GB memory per domain. 4 Floating Point Units (FPUs) shared between two compute cores in each NUMA domain. Each node also contains an NVIDIA K20x GPU with 6GB GDDR5 memory connected via PCIe in a master-slave orientation. The GPU-based, heterogeneous architectural path to exascale faces a similar problem with effective utilization. As legacy application codes are being ported to GPU architectures, the CPUs are often under-utilized; or, for applications that use only
<table>
<thead>
<tr>
<th></th>
<th>Titan</th>
<th>Summit</th>
<th>Sequoia</th>
<th>Sierra</th>
<th>Mira</th>
<th>Aurora</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Location</strong></td>
<td>ORNL</td>
<td>ORNL</td>
<td>LLNL</td>
<td>LLNL</td>
<td>ANL</td>
<td>ANL</td>
</tr>
<tr>
<td><strong>Time Period</strong></td>
<td>2012-2017</td>
<td>2017-</td>
<td>2012-2017</td>
<td>2017-</td>
<td>2012-2018</td>
<td>2017-</td>
</tr>
<tr>
<td><strong>Predecessor</strong></td>
<td>Titan</td>
<td>Sequoia</td>
<td>Mira</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Nodes</strong></td>
<td>18,688</td>
<td>3400</td>
<td>98,304</td>
<td>1944</td>
<td>49,152</td>
<td>50,000</td>
</tr>
<tr>
<td><strong>Processor</strong></td>
<td>AMD</td>
<td>IBM</td>
<td>IBM</td>
<td>IBM</td>
<td>IBM</td>
<td>IBM</td>
</tr>
<tr>
<td></td>
<td>Opteron</td>
<td>POWER9</td>
<td>BlueGene/Q</td>
<td>POWER9</td>
<td>BlueGene/Q</td>
<td>POWER9</td>
</tr>
<tr>
<td><strong>Dedicated Accelerator</strong></td>
<td>NVIDIA</td>
<td>NVIDIA</td>
<td>N/A</td>
<td>NVIDIA</td>
<td>N/A</td>
<td>3rd Gen Xeon-Phi</td>
</tr>
<tr>
<td><strong>Peak Performance (PFLOPS)</strong></td>
<td>27</td>
<td>150-300</td>
<td>20</td>
<td>100+</td>
<td>10</td>
<td>180</td>
</tr>
<tr>
<td><strong>Vendor</strong></td>
<td>Cray</td>
<td>IBM</td>
<td>IBM</td>
<td>IBM</td>
<td>Intel</td>
<td>Intel</td>
</tr>
<tr>
<td><strong>Power Consumption (MW)</strong></td>
<td>~ 9</td>
<td>~ 10</td>
<td>N/A</td>
<td>N/A</td>
<td>4.8</td>
<td>13</td>
</tr>
<tr>
<td><strong>Memory Per Node</strong></td>
<td>38GB (GDDR5 + DDR3)</td>
<td>512GB (HBM + DDR4)</td>
<td>16GB</td>
<td>24GB</td>
<td>16GB</td>
<td>64GB</td>
</tr>
</tbody>
</table>

Table 2.2: Overview of current and upcoming supercomputers.
CPUs, the GPUs are under-utilized. This scenario is being observed even today on the Titan machine. For instance, the Community Earth System Model (CESM) is a large-scale climate application that is CPU-only, S3D is a combustion physics application that has unused CPU cores or low GPU usage, or Amber, the molecular dynamics code that is GPU-based and has unused CPU cores. This problem will only be exacerbated on future systems, such as SUMMIT, with multiple GPUs and many-core CPUs. Thus, the challenge and opportunity on both the many-core and heterogeneous architectures is the effective utilization of unused resources towards an application’s own end-to-end tasks that can help improve both resource utilization and end-to-end application performance.

To address both the data movement and core under-utilization problems, we propose a runtime architecture for end-to-end application execution, wherein all end-to-end application assist tasks and system services are distributed across all system nodes. End-to-end data and computing tasks necessarily must be broken into smaller parts and distributed across the nodes. A key component to maximizing system performance within a given power envelope will be to intelligent decisions regarding when and how to move data between nodes (and even within nodes). Services must also be available on each node. Off-node services are becoming even more costly in terms of overly long retrieval times, higher power usage, and future networks that may impose severe bandwidth tapering relative to local and “nearby” nodes, e.g. dragonfly topologies. For example, the SUMMIT machine in 2018 will have a relatively modest interconnect bandwidth out of each compute node (Dual Rail EDR-IB at 23GB/s), compared to its superior processing power and memory capacity (approximately 40TF and 512GB of DRAM), which implies a data production

![Figure 2.2: Heterogeneous supercomputer architecture overview.](image)
rate that is better served by node-local analysis.

Not only will end-to-end application tasks be required to be co-located with data, but all system services (e.g., resiliency, monitoring) will also need to be co-located with the distributed data (or at least be represented by functional stubs that facilitate the needed system services.) This paradigm is different from many current notions of in-situ analysis that divides the entire system into subsets of nodes for various functions (e.g., Adios). This centralized approach may be less appealing given the costs above. Therefore, while partitioning is a requirement, the exascale implication is that partitioning must not be done by assigning sets of nodes to functions, but rather must be present on each and every node and that within the node different cores and/or different threads will be assigned these tasks to preserve the needed data locality. Thus our design premise is: All application and service tasks must be distributed, along with the data, to each node in a scale-invariant fashion. An application-specific approach to in-situ execution, while on-node, is a custom solution that is generally tightly integrated with the main simulation code, and cannot use legacy analysis routines or other services in a scalable fashion.

Though similar in design, the total computational power and respective workloads will differ for Summit and Sierra. Sierra, the smaller of the systems, is to be delivered to Lawrence Livermore National Laboratory (LLNL) to replace their current 20 PetaFLOP Sequoia supercomputer. LLNL will be using Sierra for the National Nuclear Security Administration’s ongoing nuclear weapon simulations, with LLNL noting that the machine will be dedicated to high-resolution weapons science and uncertainty quantification for weapons assessment. Due to its use in nuclear weapons simulations, information on Sierra is more restricted than it is for Summit. Publicly, Sierra is being quoted as offering 100+ PFLOPS of performance, over five-times the performance of Sequoia. As these supercomputers are still in development the final performance figures are unknown. Power consumption and clock speed cannot be guaranteed this early in the process, not to mention performance scaling on such a large system and it is likely Sierra will exceed its 100 PFLOPS performance floor.

Another such supercomputer that contains multiple accelerators is Stampede at Texas Advanced Computing Center (TACC), which contains dual NVIDIA Kepler GPUS and/or dual Intel’s Second Generation Xeon-Phi’s per node. Along with Stampede, the Cori Supercomputer at National Energy Research Scientific Computing Center (NERSC) and Trinity at Los Alamos National Laboratory (LANL) will also be part Intel Haswell and part Intel Xeon-Phi’s.
2.5 Hardware Parameters

One tool used for performance analysis is hardware counters, which are special purpose
registers that are part of most modern processors. These registers are triggered when certain events
take place in software. An example of such a register would be one that is triggered every time
there is a new instruction. The values obtained from such a counter can help describe the nature
of an application because of their unique values. Advantages of hardware counters consist of low
overheads and no modification required in the source code. We collect these hardware counters
and provide them as inputs to our prediction system. Depending on the platform, we have different
tools that help us capture these hardware counters.

2.5.1 PAPI: Performance API

Performance API [21] is a two layer API that enables interaction with the hardware counters
on the processor. The high level layer caters to users with simple needs while the low level layer
provides more in-depth information. The level of our prediction model requires us to use the low
level layer to capture the values of different PAPI parameters, which are then fed as inputs to our
prediction model. The PAPI parameters collected can be further classified into two categories:
derived and underived. As the name suggests, the derived parameters are obtained from the
underived parameters while the underived parameters are independent and are obtained directly
from the hardware counters.

2.5.2 CUPTI: CUDA Profiling Tools Interface

The CUDA Profiling Tools Interface is a performance analysis tool for applications running
on NVIDIA GPUs. The profiling interface is divided into two parts, the first being a callback
API that allows the interface to add additional code that monitors the application behavior. The
second part of the interface has direct access to the built in hardware counters on NVIDIA GPUs.
As mentioned above, we do not wish to introduce any additional overheads while profiling our
benchmark applications and hence we make use of the latter to collect hardware performance
counters such as instruction counts, memory transactions, cache hit/misses, etc.
2.6 Chapter Summary

In this chapter, we introduce the readers to heterogeneous architectures along with the accelerator based supercomputers that demonstrate the capability and motivation to achieve exascale computing. We also brief description of the benchmark applications utilized to evaluate the A2A and HFP frameworks.
Chapter 3

Literature Review

In this chapter we provide the literature review for our A2A framework including mapping applications and algorithms and accelerators in Section 3.1 followed by the literature review for the HFP framework that allows applications to utilize heterogeneous environments in an efficient way for optimal performance in Section 3.2. Lastly, we provide a literature review of the main case study application: Gene Network Alignment using GPGPUs (G3NA) in Section 3.3.

3.1 Algorithm-to-Architecture (A2A) Mapping

Since the development of HPC systems, porting of applications has gained popularity and a recent research trend is focused on performance analysis of emerging multi-core, many-core, and GPU architectures using different case studies. [22, 23, 24] present performance models for applications running on GPUs. Baghsorkhi et al. [22] describes a model based on symbolic evaluation that determines the effect on the conditions and complex memory accesses. Using the symbolic evaluation they identify bottlenecks in the application and aid the compiler with optimizations. Hong et al. [23] predicted performance using code analysis, architecture specification, and a set of equations, and performed accurate performance modeling for applications containing either memory-level parallelism, thread-level parallelism, or both. Ryoo et al. [24] developed performance metrics for evaluating GPU optimization configurations using Pareto-optimal curve. This configuration plotting on Pareto-optimal curve reduced the search space by 98% while maintaining the best performance configuration. Other performance models [25, 26] are based on multi-core architectures.
Chen et al. [25] utilizes Markov chains to analyze and provide multi-core throughput for an algorithm. Noonberg et al. [26] proposed a theoretical model of superscalar processor performance that is viewed as an interaction between application parallelism and architecture parallelism.

Blem et al. [27] presented a multicore prediction model that provides a tight upper bound on the performance of an application based on architecture specifications, application specification, chip organization, and multicore topology and verified their model on Intel Xeon, Intel i7, and GPU-Sim [28]. William et al. [29] introduces the Roofline model that provides a visual performance model for floating-point algorithms and multicore architectures. This model was further improved by Bhuiyan et al. [2]. They present a fitness model that predicts the run-time, thus predicting a suitable architecture for a given application. The fitness model represents an application using seven components including single precision (SP) instructions, double precision (DP) instructions, memory accesses and inter-device memory transfers. They verify the mapping using an SNN application with varying computation to communication ratios and an ADF (Antistropic Diffusion Filter) on Intel Xeon, NVIDIA GPUs, AMD Opteron, and IBMs Cell Broadband Engine. Feng et al. [30] proposed a benchmark suite to evaluate evolving heterogeneous computing. The proposed benchmark consists of thirteen applications from various domains such as linear algebra, graphical models, dynamic programming, and graph traversal. The authors evaluated these applications, which were written in OpenCL, on Intel Xeon, AMD HD5450 and HD5870, and NVIDIA’s GT520 and C2050 GPUs. The algorithms utilized in [30] are standard and often used to benchmark architecture performance. The research in this paper utilizes a comprehensive set of benchmark algorithms to propose and verify our mapping.

### 3.2 Heterogeneous Functional Partitioning (HFP) Framework

Management of resources in heterogeneous system environments has become important with the adoption of accelerators by the HPC community with focus on scientific applications. Antonopoulos et al. [31] focuses on design and implementation of parallel mesh generation on multi-level architectures and exploits the performance potential of multi-threaded architectures using a multi-level, multi-grain parallel mesh generation. He et al. and Barbosa et al. [32, 33] presents dynamic scheduling techniques to allocate newly arriving aperiodic jobs by modeling spare capabilities of a heterogeneous cluster on which periodic real-time jobs are running. Rafique
et al. [34] presents four design alternatives and configurations for building asymmetric clusters with PS3 accelerators as the compute nodes and multi-core x86 servers as driver and manager nodes while utilizing map reduce to hide the imbalances and architectural asymmetry, which they further extend in [35]. [36, 37] utilize available cores for online execution monitoring and/or security checking. Several efforts have also been made in researching a pipeline model to map computational tasks of an application to different cores [38, 39, 40, 41, 42, 43].

Previous frameworks such as [44] make a case for using the extra cores on a multi-core system towards its own workflow tasks such as checkpointing, de-duplication, and data post-processing. With the proposed Heterogeneous Functional Partitioning framework we extend this notion to nodes with heterogeneous resources (such as CPU-GPU nodes in Titan). With the host-device model there are significantly underutilized resources on the node. We develop and integrate this framework with scientific applications to enable overlapping of post-processing tasks with main simulations.

Goldrush framework [45] uses fine-grained scheduling to utilize idle resources to schedule in-situ data analytics with focus on multi-core CPUs. On the other hand, HFP runtime framework is fundamentally different and focuses on leveraging the heterogeneity in on-node resources (spatial rather than temporal). The predictive mechanism for fine-grained scheduling described in Goldrush is complementary to our coarse-grain scheduling approach.

In-situ data analytics is motivated by the mismatch in the scaling of compute and I/O. As the data movement becomes increasingly costly relative to computation, computing in physical proximity of data generation becomes relevant. Several works have addressed this challenge at different levels in the system. Works like Damaris [46] and ADIOS [47] aim at accelerating the I/O and post-processing the data by reserving cores on the node or leveraging data staging nodes. On the other hand, HFP runtime does not have such requirements and aims to be purely opportunistic in exploiting the resources.

### 3.3 Genomic Network Alignment

Early work in this field has been focused on Local Network Alignment (LNA) algorithms, such as MaWiSh [48], NetworkBLAST [49], PathBLAST [50], Graemlin [51] etc. MaWiSh implements an evolution-based scoring scheme to detect conserved clusters by defining the network
alignment as a maximum weight induced subgraph problem. PathBLAST uses both BLAST similarities of the proteins and the probabilities of interactions to find the biological pathways. NetworkBLAST on the other hand generates the network alignment graph using sequence similarities, and conducts a search over the generated graph to identify conserved pathways and clusters. The LNA algorithm of Graemlin was based on identifying dense conserved sub-networks of arbitrary structure, and of the two modules one is subject to evolutionary constraints, while the other is under no constraints. The LNA algorithms have proved to be ambiguous due to their one-to-many mapping, where one node can be aligned with different nodes in a separate local conserve subnetwork. Therefore, it is not practical to detect large connected subgraphs. To avoid the drawbacks of the LNA algorithms evolved to Global Network Alignment (GNA) algorithms. Pioneering work in this area was detailed in Singh et al [52] using IsoRank, which mapped the nodes of two input networks based on similarity of their neighborhood topology. IsoRankN [53] was a modification to the original IsoRank algorithm to find the alignment scores between any pair of networks and then utilizes the PageRank-Nibble algorithm [54] along with the alignment scores to infer the alignment clusters. Graemlin 2.0 was further developed as a parameter based learning algorithm, which finds the alignment between multiple networks relying on their phylogenetic relationships. GRAAL and H-GRAAL [55] are topologically based network alignment algorithms, which calculate the graphlet degree vector, whereas MI-GRAAL [56] uses both node similarity and topological network similarity measures. As established earlier, the study of gene product interactions is fundamental in understanding coordinated gene output for phenotype expression. The research challenge in creating and comparing gene networks lies in the lack of knowledge of how each node of one network maps to one or more nodes of other networks. This absence of information leads to solving the subgraph isomorphism problem. Isomorphism refers to a bijection function that preserves edge adjacency, between nodes of the two networks being aligned. Since exact comparisons are inappropriate between the nodes due to biological variations, the challenge lies in fitting one network into another without an exact subgraph. Therefore, the need to develop an efficient and accurate multiple network alignment algorithm arises.

Pairwise graph alignment is performed to detect subnetworks, which represent co-functional gene modules, common to both graphs or specific to one graph. There are several classification categories with which network alignment can be performed. The first category is the global versus local alignment, where local alignment mappings are chosen independently for each local region of
similarity. These local mappings can be ambiguous with one node having pairings with different local alignments. As described above, popular LNA algorithms include PathBLAST, NetworkBLAST, MaWiSh, Graemlin. Global graph alignment referenced above provides better overall alignment from every node in one network to nodes in the other networks, which leads to sub-optimal matchings in certain local regions. Major GNA algorithms are GRAAL,IsoRank, IsoRankN, and Extended Graemlin. We implement a GPU-optimized GNA algorithm in our software implementation.

We utilize G3NA as hands-on real world application that was proposed by a genomic researcher during our initial research of A2A qualitative analysis. We utilize the A2A model to begin with an optimal architecture and programming model thus minimizing the development time and providing the genomic researcher with an sub-optimal code to perform his experiments. We explore other architectures to verify the accuracy of A2A model on G3NA after the initial development and then further optimized algorithm.

3.4 Summary

In this chapter we elaborate on the related works for the A2A and HFP frameworks. Literature review for A2A consists to performance prediction and fitness models, and their benchmark application. Related works for HFP focuses on the efforts that attempt to optimally utilize resources and previously proposed frameworks. In the last section, we introduce the literature for our primary application: Gene Network Alignment using GPGPUs (G3NA), which focuses on algorithms proposed to achieve gene network alignment and their drawbacks. One of the primary basis of comparison between previously proposed algorithms and our implementation using GPUs is the significant decrease in computation time.
Chapter 4

Algorithm-to-Architecture (A2A) Framework

In this section, we discuss the proposed qualitative A2A framework for providing an initial mapping of an algorithm to the appropriate architecture with qualitative and quantitative understanding of the application contrary to comprehensive understanding of the code. We consider mapping the application to multiple architectures including single node Intel Xeon (CPU), Xeon-Phi (PHI), and NVIDIA Kepler GPU (GPU). All of the aforementioned architectures have unique specifications that distinctively match well with certain application characteristics. This chapter is divided into two, in the first part of the chapter we analyze and provide a qualitative analysis. Qualitative analysis allow user to analyze applications without runtime analysis or profiling. In the second part of the chapter, we will utilize application runtime parameter to predict performance across architecture as well as the best suited architecture for the application as well.

4.1 Qualitative Analysis

4.1.1 Algorithm Space

There are several characteristics that affect the performance of an algorithm as represented in Table 4.1. They can be divided into three major categories Computation and Memory Access Time, Communication Time, and I/O Time. All of the factors shown in the table are dependent on Input Data,
User Defined Parameters, and Algorithm. We merge these factors into four qualitative classifiers. To perform this classification, we make the following assumptions:

1. The data is loaded into memory and available for use.

2. For a parallel algorithm, the algorithm under consideration has performed network communication.

The computation time for any algorithm is directly proportional to algorithm complexity and hence it becomes the first classifier. The second classifier is constructed by the ratio of FLOPs (Floating Point Operations)-to-Non-FLOPs. The FLOPs-to-Non-FLOPs ratio provides key insight to an algorithm. Operations such as branching, indexing, and other NON-FLOPs can dominate the main algorithmic operations, thus, it is important to use these classification factors. Memory accesses for computation depends on data locality in the memory hierarchy, which we refer to as memory access behavior forming the third classifier. Memory access behavior includes non-uniform memory access (NUMA), main memory, L3/Shared, L2, and L1 memory access. On-chip memory access are faster and better for any algorithm. However, memory access behavior is a misleading classifier if the frequency of memory access is low. Thus the algorithm space is divided into the following four qualitative dimensions: time complexity, FLOPs-to-Non-FLOPs ratio, memory access behavior, and memory access frequency.

4.1.1.1 Time Complexity

Algorithm complexity does not always have a strong correlation to an optimal architecture, especially for small input sizes and weak scaling complexities such as 1, \( n \), and \( n \log n \). Often for small input sizes, the architecture under consideration does not affect the application run time to a significant degree. Algorithm complexity is easy to detect using standard techniques such as running multiple experiment by varying input data-set of varying sizes or by analyzing the code for the nested for loop and their conditions. The latter technique is an empirical way and a common technique for analysis for worst case complexity.

4.1.1.2 FLOPs-to-Non-FLOPs Ratio

Algorithmic instructions in applications can be sub-divided into two categories: FLOPs or Non-FLOPs. Most scientific applications perform significantly more FLOPs than Non-FLOPs.
<table>
<thead>
<tr>
<th>Application Performance Characteristics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation Ratio</td>
<td>Time spent performing computation</td>
</tr>
<tr>
<td>FLOPs</td>
<td>Amount of floating point operation</td>
</tr>
<tr>
<td>Non-FLOP</td>
<td>Amount of non-floating point operations</td>
</tr>
<tr>
<td>L1 Cache Accesses</td>
<td>L1-Cache Hit and Miss Ratio</td>
</tr>
<tr>
<td>L2 Cache Accesses</td>
<td>L2-Cache Hit and Miss Ratio</td>
</tr>
<tr>
<td>L3 Cache Accesses</td>
<td>L3-Cache Hit and Miss Ratio</td>
</tr>
<tr>
<td>Main Memory Accesses and Latency</td>
<td>Access Time to Main Memory</td>
</tr>
<tr>
<td>Non-Uniform Memory Access</td>
<td>Time spent performing NUMA</td>
</tr>
<tr>
<td>I/O Ratio</td>
<td>Time spent performing I/O</td>
</tr>
<tr>
<td>Communication Ratio</td>
<td>Time spent performing communication</td>
</tr>
<tr>
<td>Network Bandwidth and Latency</td>
<td>Network Delay and bandwidth</td>
</tr>
<tr>
<td>Network Congestion</td>
<td>Congestion of network during various run</td>
</tr>
</tbody>
</table>

Table 4.1: Performance Impact Factors

However, applications such as encryption/decryption and encoding/decoding algorithms consist mostly of Non-FLOPs. Williams et al. [29] illustrate that application performance can vary across different architectures based on the amount of FLOPs, which results in higher bandwidth. Optimization of applications with higher Non-FLOPs-to-FLOPs ratio are often better suited for devices with dedicated, optimized hardware such as FPGAs and Digital Signal Processors (DSPs) since these applications often perform operations such as bitwise and bit shifting. To obtain the FLOPs-to-Non-FLOPs ratio, we utilize profilers such as TAU [57], PETSc [58], and CrayPAT [59]. FLOPs-to-Non-FLOPs often vary with change in parameter and size.

4.1.1.3 Memory Access Behavior

Memory access behavior refers to dominant accesses in the memory hierarchy. Each of the architectures under consideration consist of on-chip memory as well as off-chip memory. On-chip memory accesses are less expensive in clock cycles but limited by size. Off-chip accesses can be
more expensive but have larger memory space, thus fitting algorithms with larger memory space complexity. Algorithms that allow frequent and dominant access to on-chip memory often result in better performance in comparison to applications having larger memory space complexity and frequent memory accesses.

4.1.1.4 Memory Access Frequency

Applications with low memory access frequency may not be significantly affected by memory access behavior. Thus it is important to consider the impact of memory access frequency as a compounding factor to memory access behavior. An algorithm with bad memory access behavior to main memory and high memory access frequency will result in sub-optimal performance and vice-versa. Both memory access behavior and memory access frequency can be deduced from the code (or pseudocode) or by the profilers previously mentioned.

4.1.2 Architecture Space

In this section, we describe the three architectures under consideration: Intel Xeon E5 (Single-Node), Intel Xeon-Phi (Single-Phi), and NVIDIA K20 (Single-GPU).

4.1.2.1 Intel Xeon-Phi E5110P

The current generation of Xeon-Phi Coprocessors are located on the PCI Express (PCIe2) with 61 processor cores running at 1.238Ghz and interconnected via bi-directional ring with a theoretical peak of 352GB/s using 8 memory controllers each with 2 channels of 5.5GT/s. Each core has 32KB Instruction and 32KB Data L1 cache; 512KB shared cache and 16GB DDR5 global memory. In Offload mode, the user has the benefit of targeting specific computation onto dedicated architectures using the available offload directives. In the Symmetric mode of operation, MPI processes uniformly span the domains of both the host and the coprocessor architectures with explicit management of parallelism across the two different architectures. The Coprocessor-only mode is a subset of Symmetric mode with all MPI processes being confined to the Xeon-Phi architecture alone.
4.1.2.2 Nvidia K20 GPU

The Nvidia K20 Kepler Series GPUs are also accessible via the PCIe2 bus with 2688 cores and a core clock speed of 732 MHz. The K20 architecture has 6GB DDR5 global memory available at 250GB/s bandwidth. Each core has L1 cache and L2 data cache available in three configurations (16KB L1/48KB L2, 32KB L1/32KB L2, and 48KB L1/16KB L2).

4.1.2.3 Intel Xeon E5-2680

Intel E5-2680 processors are multi-core, Intel Hyper-Threading Technology (HT) enabled designs. Each socket has eight cores running at 2.6Ghz each, which share L3 cache, a local integrated memory controller connected via an Intel QuickPath interconnect. The cores share 32GB DDR3 global memory with 32KB data and instruction L1 cache, a 256 KB unified L2 cache, and 8 MB L3 Cache.

4.1.3 Proposed Mapping

Using the algorithm classification and architecture specification, we create a unique one-to-one mapping by qualitative classification of algorithmic factors and architecture functionality. We verify this mapping using the micro-benchmarks described in Section 4.1.4. Mapping is performed by sub-classification of the above defined algorithm classifications from Table 4.1.1. In addition to sub-classification, it is necessary to have a ranking order for the four algorithm classifications. We develop this order of preference based on the performance impact.

<table>
<thead>
<tr>
<th>Classification</th>
<th>Low End</th>
<th>High End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithmic Complexity</td>
<td>Weak</td>
<td>Strong</td>
</tr>
<tr>
<td>FLOPS-to-Non FLOPS</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Memory Access Behaviour</td>
<td>Good</td>
<td>Bad</td>
</tr>
<tr>
<td>Memory Access Frequency</td>
<td>High</td>
<td>Low</td>
</tr>
</tbody>
</table>

Table 4.2: Classification of Algorithmic Classifiers.

Algorithm complexity can be classified into two categories: strong scaling and weak scaling. We consider strong scaling algorithms as higher polynomial degree greater than or equal to
Any complexity less than $O(n\log n)$ is classified as weak scaling. Very small values of $n$ result in indistinguishable performance for weak scaling algorithms. Larger values of $n$ result in strong divergence in run-time performance. This divergence grows exponentially for strong scaling applications and slowly for weak scaling applications. GPUs are strong in performing algorithm complexity reduction due to its ability to hide latency, multi-thread execution, and zero-overhead context switching. In comparison, Xeon-Phis allow for multi-thread execution environment, but have limited thread launch even though they run at higher core frequency than the GPUs. The Single-Node CPU in comparison provides the worst scaling for large $n$ due to process count limitation of $16/32$ even though it has the highest frequency of the architectures considered. Hence, small size $n$ will provide a competitive performance on CPU and Xeon-Phi.

The second algorithm classifier, FLOPS-to-Non-FLOPS is classified High or Low to indicate the amount of FLOPs in the program. FLOPs-to-Non-FLOPs can also be considered as an indirect indicator of the amount of memory accesses, branching and other operations. We consider an algorithm to have a high FLOPs-to-Non-FLOPs ratio if the algorithm performs one FLOP per two
Non-FLOP. We compute this by calculating the average latency of the three given architectures and dividing by the maximum concurrent threads that can be run on these architectures. GPUs are optimal to perform FLOPS however, when branching, divergence or synchronization is introduced, i.e. for algorithms that involve Backtracking or Branch and Bound problems, there is a strong performance degradation. Xeon-Phis provide a better architecture to accomplish this with dual memory banks in a ring, allowing for better performance when Non-FLOPs are involved, similar to CPUs. FLOPs-to-NON-FLOPs of an algorithm can change both with input data and parameters. For example, a converging algorithm might require more operations for a smaller threshold value, also increasing the Memory Access Frequency.

The third and fourth algorithmic classifier, memory access behavior and memory access frequency are tightly coupled to each other. We sub-classify memory access behavior into two sub categories: Good and Bad. A good memory access behavior is defined as having more memory accesses to on-chip memory due to its low latency. However, due to the limited size of on-chip memory, it is important to consider that on-chip memory is directly affected by input data size. Memory access pattern is independent of input data size, but rather a factor of algorithm design. A good memory access behavior correlates to strong performance on GPU and Xeon-Phi due to architecture design. We consider memory access frequency as a compounding factor. Memory access behavior will have no impact if the memory frequency is not high enough for the amount of computation. Thus, memory access frequency is a factor of both input data size and application parameters.

4.1.4 Analysis for Qualitative A2A Mapping

In this section, we describe the 20 applications used to verify Tesseract’s mapping from the Algorithm Space to the Architecture Space. We evaluate the applications described in Section 2.3 on three accelerator based supercomputers: Titan at Oak Ridge National Lab [60], Stampede at The University of Texas at Austin [13], and Palmetto at Clemson University [61]. We use Nvidia GPUs, Xeon-Phis, and Intel E5 on each of our respective supercomputers to evaluate Tesseract.

To evaluate Tesseract, we first perform classification of the applications (algorithms) as shown in Table 4.3. The classifications are based on input parameter and data size, thus for a different size the classification may change. After performing the classification for a given application based
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
<th>FLOPs-to Non-FLOPs Ratio</th>
<th>Memory Access Behaviors</th>
<th>Memory Access Frequency</th>
<th>Result based Mapping</th>
<th>Tesseract based Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEM</td>
<td>High</td>
<td>High</td>
<td>Bad</td>
<td>High</td>
<td>(P,N) to G</td>
<td>P</td>
</tr>
<tr>
<td>LAVAMD</td>
<td>High</td>
<td>High</td>
<td>Good</td>
<td>Low</td>
<td>(G,P) to G</td>
<td>G</td>
</tr>
<tr>
<td>SWAT</td>
<td>High</td>
<td>Low</td>
<td>Bad</td>
<td>High</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>NW</td>
<td>High</td>
<td>Low</td>
<td>Bad</td>
<td>High</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>SRAD</td>
<td>High</td>
<td>Low</td>
<td>Good</td>
<td>High</td>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>CFD</td>
<td>Low</td>
<td>High</td>
<td>Bad</td>
<td>High</td>
<td>G, N</td>
<td>P</td>
</tr>
<tr>
<td>HMM</td>
<td>High</td>
<td>Low</td>
<td>Good to Bad</td>
<td>High</td>
<td>(G,N) to P</td>
<td>G to P</td>
</tr>
<tr>
<td>K-means (Vary Size)</td>
<td>High</td>
<td>High</td>
<td>Good</td>
<td>High</td>
<td>(G,P) to G</td>
<td>G</td>
</tr>
<tr>
<td>K-means (Vary Cluster)</td>
<td>High</td>
<td>High</td>
<td>Good</td>
<td>High</td>
<td>(G,P) to G</td>
<td>G</td>
</tr>
<tr>
<td>K-means (Vary Dimension)</td>
<td>High</td>
<td>High</td>
<td>Bad</td>
<td>High</td>
<td>G, P</td>
<td>P</td>
</tr>
<tr>
<td>LUD</td>
<td>High</td>
<td>Low</td>
<td>Good</td>
<td>High</td>
<td>G, N</td>
<td>G</td>
</tr>
<tr>
<td>SPMV (Vary Size)</td>
<td>High</td>
<td>High</td>
<td>Good</td>
<td>Low</td>
<td>(G,N) to G</td>
<td>G</td>
</tr>
<tr>
<td>Triad</td>
<td>Low</td>
<td>High</td>
<td>Good</td>
<td>High</td>
<td>G,P</td>
<td>G</td>
</tr>
<tr>
<td>Reduction</td>
<td>Low</td>
<td>High to Low</td>
<td>Bad</td>
<td>Low</td>
<td>G</td>
<td>P</td>
</tr>
<tr>
<td>FFT</td>
<td>High</td>
<td>High</td>
<td>Good</td>
<td>Low</td>
<td>(G,P) to G</td>
<td>G</td>
</tr>
<tr>
<td>SCAN</td>
<td>High</td>
<td>Low</td>
<td>Good</td>
<td>High</td>
<td>(G,P) to G</td>
<td>G</td>
</tr>
<tr>
<td>Stencil2d</td>
<td>Low</td>
<td>High</td>
<td>Good</td>
<td>High</td>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>TDM</td>
<td>High</td>
<td>Low</td>
<td>Bad</td>
<td>High</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>CRC</td>
<td>Low</td>
<td>Low</td>
<td>Good</td>
<td>Low</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>RadixSort</td>
<td>High</td>
<td>High</td>
<td>Bad</td>
<td>High</td>
<td>G</td>
<td>P</td>
</tr>
<tr>
<td>BFS</td>
<td>High</td>
<td>Low</td>
<td>Bad</td>
<td>High</td>
<td>(G,N) to G</td>
<td>N</td>
</tr>
<tr>
<td>A*</td>
<td>High</td>
<td>Low</td>
<td>Bad</td>
<td>High</td>
<td>(N,P) to N</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 4.3: Algorithm Classification and Mapping to Single-(G)PU, Single-(P)hi and Single-(N)ode.
on the dimension for a given problem, the algorithm will lie in the algorithm space, and can be mapped to the architecture space by referring to Tesseract mapping model shown in Figure 4.1. Thus, for example if an application has a high time complexity, good memory access behavior, low memory access frequency, and high FLOPs-to-Non-FLOPs ratio, then based on Tesseract the application would perform optimally on a Single-GPU. For classification that contain transition due to varying input data, we imply ‘to’ for indicating transition. This transition indicates change in optimal architecture with change in size. G, N, and P represent Single-GPU, Single-Node, and Single-Phi, respectively.

Linear Algebra can be sub classified into two categories Dense and Sparse linear algebra. Dense linear algebra has unit-stride memory accesses to read data from rows, and strided accesses to read data from columns. These are computationally heavy. However, sparse linear algebra is an equal mix of computation and memory bandwidth. In linear algebra class of algorithm we consider 4 applications K-means, LUD, SPMV and TRIAD. For K-means we consider 3 different scenarios. In the first scenario we vary the input size between 500,000 and 2,000,000 items while keeping a constant diameter (dimensions) and cluster size. We then maintain the input size (500,000) and dimension constant(50) and vary the number of clusters between 50 and 200 and lastly we keep the input size and cluster constant and vary the dimensions with minimum of 50 and maximum of 200. For LUD benchmark application we vary the input size exponentially from 128 to 2048. Lastly, we consider the two sparse linear algebra applications SPMV and TRIAD. For SPMV we vary the size between 16384 and 131072 while for TRIAD which is a version of the stream triad benchmark, we measure the time for constant size of 16MB and perform iterative trials. Figure 4.2 shows performance results for linear algebra.

Due to the fully intensive and semi intensive computation nature of the problem, GPUs are efficient in performing these computations on large data. However, for small data sizes, the execution time difference provides less than 20% difference in run-time for the three architectures due to the low FLOPs-to-Non-FLOPs. Furthermore, as depicted by Figure 4.2c, when we increase the dimensions, more synchronization is required between the threads thus making Phi as efficient as GPU, due to availability of dual ring memory. For sparse linear algebra, due to the high bandwidth of GPU and algorithm being semi bandwidth intensive, better performance on a GPU can be achieved. The GPUs perform well for low data size for SPMV due to the ability to perform data reduction and parallel nature of problem, allowing for a fit into the lower memory hierarchy
thus having a good memory behavior for low data-size.

Divide and Conqueror, N-Body, and Structured grid classes of algorithms are inherently parallel and thus more suitable to scaling, due to their relatively low communication overhead. Furthermore, these classes of algorithms usually require high bandwidth and utilize a spectral memory access pattern thus making them ideal for GPUs. This is illustrated in Figure 4.3, 4.4, and 4.5a.

For the class of Divide and Conqueror we consider four applications namely: Reduction, FFT, SCAN and STENCIL 2D. Based on the classification of these application Single-GPU is predicted as the optimal architecture for all but Reduction. Due to inherent parallelism and division ability. We see for varying number of elements and matrix sizes for FFT, SCAN, and STENCIL2D Single-GPU outperforms Single-Phi and Single-Node on an average of 23%. However, our pre-
For N-body we consider two applications; GEM and LAVAMD, by varying the data size for GEM (number of atoms and residue) and LAVAMD (number of boxes). Tesseract predicts Single-Phi for GEM, and Single-GPU for LAVAMD; however, for GEM due to bad frequent memory behavior and coalesced memory access for large data size, the GPU is more optimal by 5% for large input sizes but Single-Phi is as efficient as GPUs for small input sizes.

Grid class of algorithms can further be sub-divided into unstructured grid and structured grid. We consider SRAD for structured grid class and CFD for unstructured grid class. Due to inherent parallel and spectral pattern of structured grid memory access, GPUs will out perform other architectures as predicted by Tesseract due to their ability to hide memory access latency. We observe this phenomenon in Figure 4.5a. As the size of the input data increases exponentially, there is a linear increase in performance for Single-GPU, however Single-Node and Single-Phi increases exponentially. Due to unpredictable access behavior of unstructured grid, GPUs are not optimal.
This should result in strong Single-Node performance, however, we observed similar performance between Single-Node and Single-GPU. This could indicate that the algorithm written in OpenCL was optimized towards performance on GPUs.

Algorithms such as Dynamic Programming and Branch and Bound shown in Figures 4.6 and 4.7 respectively are more suitable to Single-Node and Single-Phi due to their high irregular memory access pattern and frequency. This performance is observed in the applications SWAT and NW associated with Dynamic Programming and AStar(A*) associated with Branch and Bound problem for varying input and sample sizes.
Figure 4.7: Branch and Bound Algorithms.

Figure 4.8: Logic Algorithms

The Logic benchmark that consist of TDM and CRC is also more suitable towards Single-Node due to their high Non-FLOPs count as shown in Figure 4.8 as predicted by Tesseract. Similarly, algorithms that utilizes Graphical Models, such as HMM require random access to memory with relatively low FLOPS and bad memory access behavior. These factors are further compounded by their memory structure, which are inefficient for GPUs, thus being more suited to Single-Phi as represented by Figure 4.9.

Lastly, for Sorting Algorithms such as Radix Sort and BFS Single-GPUs are observed as the optimal architecture in Figure 4.10b. Tesseract predicts Single-Node as the optimal architecture. On further examination of the code we found that BFS belonging Graph Traversal algorithm class were optimized for GPUs by reduction of branch divergence leading to more optimal performance on GPUs.

4.1.5 Discussion

The Tesseract based A2A qualitative mapping disagrees for a few cases when compared to what a quantitative based mapping can achieve for experimental results. Nevertheless, when considering porting algorithms to new architectures, it may be infeasible to obtain the necessary
values for a quantitative mapping. Causes for infeasibility include the sheer size of the algorithm or the time and resources involved in obtaining the values. Furthermore, on many occasions the algorithm is not a major part or bottleneck of an application but still can be accelerated using an accelerator. A quick initial A2A mapping where the user does not invest into porting the algorithm is required. Thus, a qualitative based mapping can still valuable as shown by our case study on G3NA described in Section 6.1.

Another issue encountered was increasing the number of classifications and sub-classifications, which makes the process of initial porting cumbersome for the same reasoning as described above. Although, it may provide better accuracy in mapping, it increases the classification complexity in classification and introduces ambiguity for the user. Furthermore, it is hard to get complete coverage of the Tesseract by increasing number of classification and/or classifier without significantly increasing the number of benchmark application. Thus, we provide a simplified qualitative based mapping with only the four factors that have the strongest impact on the application performance. Tesseract can further be updated with advent of new architectures and accelerators.

Lastly, our A2A model can be extended to multi node heterogeneous system architectures.
such as Multi-Node, Multi-Phi, and Multi-GPU or even heterogeneous clusters partitioned based on architecture such Cori [14] and Stampede [13]. To extend the current A2A model to a Multi-Node heterogeneous algorithm implementation, we consider communication time as a factor. The first factor i.e. Time Complexity is thus renamed to computation-to-communication ratio, which can be either fine-grained or coarse-grained. In fine-grained, tasks per thread are relatively small in terms of execution time and larger time is spent in communication of data and synchronization between threads. However, in coarse-grained parallelism, all tasks performed spend more time in computation than communication and synchronization. For example, in a Multi-Node implementation of K-means, the computation-to-communication ratio changes from coarse grained to fine grained as we increase the maximum number of clusters that can be identified. I/O consists of two categories, i.e. parallel and serial. Parallel I/O can hence induce fine-grained parallelism within the application since all threads perform reads simultaneously. Thus, the computation-to-communication ratio can also incorporate the I/O time of a large scale application.

4.2 Qualitative Analysis

In this section, we perform application to architecture mapping using quantitative analysis and application performance prediction. In order to do this we collect hardware parameters as described in Chapter 2.

In order to perform porting, we initially collected 50 hardware counters reported from the Performance API (PAPI) and 143 with the CUDA Profiling Tools Interface (CUPTI). We utilize a reduced subset parameters to then train machine learning based models. In this paper, we present the results for our best implementation.

As a first step we perform dimension reduction across our input data due to large input parameter space and overlapping performance metrics provided by both PAPI and CUPTI. To perform dimensional reduction, we evaluate the MINE score, which is nothing but the Pearson Coefficient representing how closely variable X is correlated to variable Y. The tables in Appendix A show the hardware counters we used for each of our training models. In this paper, we present two models: Quantitative model and Qualitative model. Quantitative model performs cross platform performance prediction. For a given implementation, we can utilize machine learning to predict the porting performance from one programming model and architecture to another. The qualitative
<table>
<thead>
<tr>
<th>Implementation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU-C</td>
<td>Serial C implementation running on CPU</td>
</tr>
<tr>
<td>CUDA-K40</td>
<td>CUDA implementation running on Nvidia CUDA K40</td>
</tr>
<tr>
<td>CPU-OpenMP-16</td>
<td>OpenMP implementation running on CPU with 16 threads</td>
</tr>
<tr>
<td>CPU-OpenMP-08</td>
<td>OpenMP implementation running on CPU with 08 threads</td>
</tr>
<tr>
<td>PHI-OpenMP-240</td>
<td>OpenMP implementation running on Knights Corner based Intel Xeon Phi with 240 threads</td>
</tr>
<tr>
<td>PHI-OpenMP-120</td>
<td>OpenMP implementation running on Knights Corner based Intel Xeon Phi with 120 threads</td>
</tr>
<tr>
<td>PHI-OpenMP-272</td>
<td>OpenMP implementation running on Knights Landing based Intel Xeon Phi with 272 threads</td>
</tr>
<tr>
<td>PHI-OpenMP-136</td>
<td>OpenMP implementation running on Knights Landing based Intel Xeon Phi with 136 threads</td>
</tr>
<tr>
<td>CPU-OpenCL</td>
<td>OpenCL implementation running on CPU</td>
</tr>
<tr>
<td>GPU-OpenCL</td>
<td>OpenCL implementation running on Nvidia CUDA K40</td>
</tr>
<tr>
<td>PHI-OpenCL</td>
<td>OpenCL implementation running on Knights Corner based Intel Xeon Phi</td>
</tr>
</tbody>
</table>

Table 4.4: Prediction model test case implementations.

the model utilizes a Random Forest Classifier to find the optimal architecture for porting, discounting the programming model.

4.2.1 Quantitative Framework: Cross-Platform Performance Prediction

In our quantitative model, we predict the performance of ten different implementations as described in Table 4.4. For prediction we use four base implementation for reference, these base implementation are implementation for which we can obtain performance parameters and include CPU-C implementation, CUDA implementation, OpenMP implementation for CPU and PHI.

The underlying model for our quantitative model is a neural network based on back-propagation algorithm. Our selection of neural network learning model is influenced by neural network properties such as being robust to noise in training data and fast evaluation of the learned target function. The goal of the neural network is to learn the weights associated with linear combination performed in the perceptron. For our prediction model, we make use of a multi-layer network learned by back-propagation algorithm which is capable of expressing nonlinear relationships. The way back-propagation algorithm works is it employs gradient descent to minimize the
squared error between network output and the actual output.

We implement our neural network model using Tensorflow which is an open-source software library for machine learning. We use the built in functions to implement a neural network regressor and further tweak it to achieve better prediction results. Our model implemented 3 hidden layer in addition to the input and the output layers. The number of input nodes were variable and depended on the number of hardware counters that were highly correlated to run time. The output layer had only one node which would output the predicted run time. For our hidden layers, we used variable number of nodes which were decided after performing a series of experiments to verify which configuration gave the best result. For example, when training with hardware counters collected from a serial C implementation, the number of nodes in the hidden layers are 20, 15 and 5 respectively. With the network configuration complete, the next important step was to tweak the hyper parameters themselves to ensure optimal training.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning Rate</td>
<td>0.001</td>
</tr>
<tr>
<td>Loss Function</td>
<td>MSE</td>
</tr>
<tr>
<td>Batch Size</td>
<td>3</td>
</tr>
<tr>
<td>Training Iteration</td>
<td>1200</td>
</tr>
<tr>
<td>Momentum</td>
<td>0.5</td>
</tr>
<tr>
<td>Weight Decay</td>
<td>0.1</td>
</tr>
<tr>
<td>Dropout</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.5: Hyperparameter values of our neural network for our quantitative model.

For our qualitative model prediction using the quantitative parameter of the application, we predict platform and architecture for aforementioned implementations. Similar to our qualitative model use four base implementation as mentioned above. Our qualitative model is based of a Random Forest Classifier. Since the task of predicting the platform or architecture can be extended from the concept of a simple decision tree, we make use of Random Forest Classifier because they overcome one of the most important drawbacks of decision tree which is over fitting. A Random Forest Classifier can be explained as a multitude of decision trees whose output is the value that appears most often in the individual decision trees. The input to our random forest classifier are the hardware counters and the output is the probability with which our classifier classifies the input into each of the output bins. The output bins could either be the four architectures, four programming languages or the ten implementations depending on our classification requirements.
We finally select the bin with the highest prediction confidence. Finally, for the hyper parameters themselves, we select our number of trees to be equal to 1500 which gives us the best prediction accuracy.

Henceforth, we describe the result of various application porting models. Appendix A describes the list of application that were utilized to verify the training and accuracy of our model. The tables in Appendix A show that we have implementations for all our applications across all the available programming models with a few exceptions. We evaluate the applications described on three accelerator based supercomputers: Titan at Oak Ridge National Lab [60], Stampede at The University of Texas at Austin [13], and Palmetto at Clemson University [61]. We use Nvidia GPUs, Xeon-Phis, and Intel E5 on each of respective supercomputers to obtain performance parameter to evaluate our models.

4.2.2 Quantitative Framework: Cross-Platform Performance Prediction

For our quantitative model of prediction, we evaluate our model by calculating the mean absolute error. The mean absolute error is the difference between the predicted run time and the actual run time for a given application.

Figure 4.11 is a box plot showing the mean absolute error when we use Performance API hardware counters for predicting the performance across different platforms. We see that the average mean absolute error is lowest when predicting the performance of our CUDA implementation and stands at 0.95. We observe a maximum mean absolute error for our OpenCL Xeon Phi implementation which is 20.02. Table 4.6 below gives in depth information about the minimum, average and maximum mean absolute error for each of the platforms.

When predicting the run times for our CUDA implementation, since most of our applications have low running times, the mean absolute error remains fairly low except for a few outliers as seen in the box plot that contribute towards the maximum value for our mean absolute error. We see a similar trend for other implementations such as CPU OpenMP with 16 threads, CPU OpenMP with 8 threads, Xeon Phi OpenMP with 240 threads and GPU OpenCL. On the contrary, our predictions for Xeon Phi OpenMP 120, CPU OpenCL and Xeon Phi OpenCL have comparatively large outliers. On a positive note, although Xeon Phi OpenCL has the largest value for the outliers, the number of outliers are few and thus we can say that our model is able to predict the run time
Figure 4.11: Mean Absolute Prediction Error Performance API Hardware Counters.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Minimum</th>
<th>Average</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU CUDA-K40</td>
<td>0.02</td>
<td>0.95</td>
<td>9.66</td>
</tr>
<tr>
<td>CPU OpenMP-16</td>
<td>0.29</td>
<td>3.29</td>
<td>37.76</td>
</tr>
<tr>
<td>CPU OpenMP-08</td>
<td>0.01</td>
<td>4.45</td>
<td>39.72</td>
</tr>
<tr>
<td>Xeon Phi OpenMP-240</td>
<td>0.33</td>
<td>5.26</td>
<td>38.22</td>
</tr>
<tr>
<td>Xeon Phi OpenMP-120</td>
<td>0.88</td>
<td>9.14</td>
<td>56.04</td>
</tr>
<tr>
<td>KNL OpenMP-272</td>
<td>0.01</td>
<td>1.58</td>
<td>25.63</td>
</tr>
<tr>
<td>KNL OpenMP-136</td>
<td>0.01</td>
<td>2.41</td>
<td>32.65</td>
</tr>
<tr>
<td>CPU OpenCL</td>
<td>0.19</td>
<td>13.79</td>
<td>82.09</td>
</tr>
<tr>
<td>GPU OpenCL</td>
<td>0.07</td>
<td>1.87</td>
<td>10.29</td>
</tr>
<tr>
<td>Xeon Phi OpenCL</td>
<td>0.87</td>
<td>20.02</td>
<td>85.28</td>
</tr>
</tbody>
</table>

Table 4.6: Minimum, average and maximum mean absolute prediction error using Performance API.

correctly for most cases.

A quick look at the prediction of individual applications reveals that Correlation, Syrk, and 3mm are wrongly predicted almost every time. This leads us to believe that the PAPI hardware counters are unable to characterize the nature of these applications. Another possible explanation could be error in the collection of the PAPI hardware counters themselves.

We next use hardware counters collected from our CUDA implementation using CUDA Profiling Tools Interface to predict the performance for the remaining of our implementations. From Figure 4.12 we observe a minimum mean absolute error of 1.39 for our OpenCL GPU implementation, which was expected since both implementations share the same architecture. On the contrary, the maximum mean absolute error is again seen for Xeon Phi OpenCL prediction which is 22.36.
Table 4.7: Minimum, average and maximum mean absolute prediction error using CUDA Profiling Tools Interface.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Minimum</th>
<th>Average</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU C</td>
<td>0.02</td>
<td>11.64</td>
<td>123.16</td>
</tr>
<tr>
<td>CPU OpenMP-16</td>
<td>0.36</td>
<td>14.37</td>
<td>176.61</td>
</tr>
<tr>
<td>CPU OpenMP-08</td>
<td>.029</td>
<td>13.36</td>
<td>86.16</td>
</tr>
<tr>
<td>Xeon Phi OpenMP-240</td>
<td>1.21</td>
<td>11.76</td>
<td>90.52</td>
</tr>
<tr>
<td>Xeon Phi OpenMP-120</td>
<td>0.20</td>
<td>19.26</td>
<td>121.37</td>
</tr>
<tr>
<td>KNL OpenMP-272</td>
<td>0.06</td>
<td>2.12</td>
<td>25.50</td>
</tr>
<tr>
<td>KNL OpenMP-136</td>
<td>0.14</td>
<td>2.82</td>
<td>32.42</td>
</tr>
<tr>
<td>CPU OpenCL</td>
<td>2.61</td>
<td>20.51</td>
<td>128.09</td>
</tr>
<tr>
<td>GPU OpenCL</td>
<td>0.02</td>
<td>1.39</td>
<td>9.87</td>
</tr>
<tr>
<td>Xeon Phi OpenCL</td>
<td>1.50</td>
<td>22.36</td>
<td>90.52</td>
</tr>
</tbody>
</table>

We observe a similar trend in the outlier pattern when we use CUPTI hardware counters to predict run times with a major difference being the high value of the outlier points. This phenomena showcases the low correlation that CUPTI hardware counters share with the run times and in turn leads to slightly poor predictions.

Again while predicting the run time using CUPTI, a few applications like Correlation and Syrk are mispredicted often and we believe there exists a similar reason as we saw when predicting using PAPI hardware counters.

For our other two implementations we used Performance API to collect the values of the hardware counters for our OpenMP CPU and OpenMP Phi implementations. The results for our prediction using the OpenMP CPU implementation are shown in Figure 4.13. We see that the
Table 4.8: Minimum, average and maximum mean absolute prediction error using Performance API Hardware Counters.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Minimum</th>
<th>Average</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU C</td>
<td>0.37</td>
<td>4.10</td>
<td>39.18</td>
</tr>
<tr>
<td>CPU OpenMP-16</td>
<td>0.01</td>
<td>0.39</td>
<td>1.38</td>
</tr>
<tr>
<td>CPU OpenMP-08</td>
<td>0.04</td>
<td>1.72</td>
<td>21.63</td>
</tr>
<tr>
<td>Xeon Phi OpenMP-240</td>
<td>0.04</td>
<td>2.39</td>
<td>24.98</td>
</tr>
<tr>
<td>Xeon Phi OpenMP-120</td>
<td>0.03</td>
<td>3.49</td>
<td>44.28</td>
</tr>
<tr>
<td>KNL OpenMP-272</td>
<td>0.03</td>
<td>0.50</td>
<td>2.09</td>
</tr>
<tr>
<td>KNL OpenMP-136</td>
<td>0.01</td>
<td>0.78</td>
<td>3.15</td>
</tr>
<tr>
<td>CPU OpenCL</td>
<td>0.07</td>
<td>2.78</td>
<td>14.92</td>
</tr>
<tr>
<td>GPU OpenCL</td>
<td>0.01</td>
<td>0.44</td>
<td>1.39</td>
</tr>
<tr>
<td>Xeon Phi OpenCL</td>
<td>0.13</td>
<td>22.47</td>
<td>88.97</td>
</tr>
</tbody>
</table>

The mean absolute error is very low for prediction on all platforms except OpenCL Xeon Phi. The number of outliers are low for each of the cases and can thus be used reliably to predict for all the implementations except OpenCL Xeon Phi.

The minimum error we see when using this model is 0.01 for our CUDA implementation while the maximum error is 88.97 for our OpenCL Xeon Phi implementation.

When predicting performance on other platforms using our OpenMP Xeon Phi implementation, we see results similar to that of our OpenMP CPU implementation. In both cases, we find the OpenCL Xeon Phi to be the only platform for which our mean absolute error is large. We observe a minimum error of 0.01 for OpenMP CPU while a maximum of 75.18 for OpenCL Phi.

With regards to prediction on per application basis for our OpenMP implementation, a
Figure 4.14: Mean Absolute Prediction Error using Performance API Hardware Counters.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Minimum</th>
<th>Average</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU C</td>
<td>0.08</td>
<td>5.32</td>
<td>47.45</td>
</tr>
<tr>
<td>CPU OpenMP-16</td>
<td>0.05</td>
<td>0.39</td>
<td>1.16</td>
</tr>
<tr>
<td>CPU OpenMP-08</td>
<td>0.01</td>
<td>0.94</td>
<td>4.94</td>
</tr>
<tr>
<td>Xeon Phi OpenMP-240</td>
<td>0.02</td>
<td>1.45</td>
<td>13.24</td>
</tr>
<tr>
<td>Xeon Phi OpenMP-120</td>
<td>0.12</td>
<td>2.83</td>
<td>29.24</td>
</tr>
<tr>
<td>KNL OpenMP-272</td>
<td>0.01</td>
<td>0.43</td>
<td>2.19</td>
</tr>
<tr>
<td>KNL OpenMP-136</td>
<td>0.04</td>
<td>0.79</td>
<td>3.10</td>
</tr>
<tr>
<td>CPU OpenCL</td>
<td>0.08</td>
<td>3.15</td>
<td>12.67</td>
</tr>
<tr>
<td>GPU OpenCL</td>
<td>0.02</td>
<td>0.46</td>
<td>1.35</td>
</tr>
<tr>
<td>Xeon Phi OpenCL</td>
<td>0.07</td>
<td>18.52</td>
<td>75.19</td>
</tr>
</tbody>
</table>

Table 4.9: Minimum, average and maximum mean absolute prediction error using Performance API Hardware Counters.
large error is constantly observed when predicting the run time for Correlation application.

Selection of the correlated parameters for the OpenMP model required a slightly different approach. The hardware counters have different values for each thread and thus the values from some of these threads have more correlation than others. The phenomena is especially evident for correlation of load and store instructions from thread zero as most transfer operations are handled by this thread.

Thus we can see that, Performance API hardware counters are much better at predicting the runtime for most of the architectures when compared to the CUDA Profiling Tools Interface. One reason for this could be the definition of the counters themselves, which in case of CUPTI are more targeted towards a GPU architecture. On the contrary, the definitions for PAPI hardware counters are more general and relatable across architecture.

Figure 4.15 showcases the run time predictions of real-life applications using our quantitative model trained on Performance API hardware counters. It is worth noting that no part of this test set is a part of our actual training set and hence shows our model’s competence to predict run times for user applications. The results further confirm that prediction for Xeon Phi is challenging.

4.2.3 Qualitative Framework: Cross-architecture prediction

For our qualitative model, since we predict the best suitable architecture for a given application, we assign a positive score if our predicted architecture is actually the best architecture or else we assign a negative score. Based on these scores we evaluate the final percentage accuracy for
Figure 4.16: Cumulative distribution function of prediction confidence for our qualitative prediction using Performance Hardware Counters.

Figure 4.16 shows the cumulative distribution function of the prediction probability for each of our latter mentioned qualitative prediction model. We can see that, most of the predictions made by our model are with very high confidence.

Thus, for our qualitative analysis, the accuracy is 87% when predicting the architecture and 76% when predicting the platform using hardware counters collected from our CPU C implementation. We next predict using hardware counters collected from our CUDA implementation which gives us an accuracy of 72% and 68% when predicting the architecture and platform respectively. For our next implementation which is OpenMP CPU, we see an accuracy of 81% for architecture prediction and 73% for platform prediction. Finally when using our OpenMP Xeon Phi
implementation, the accuracy is 77% for architecture prediction and 71% for platform prediction.

Lastly, we evaluate the accuracy of our Quantitative model by predicting the run time of a real world application (LULESH). The application at hand simulates the motion of materials relative to each other when subject to forces and thus establishes itself as a very good real world application. Table 4.10 shows the run time prediction for different implementations of our application. The first column labels the implementation we are trying to predict, the second column is the actual run time collected by running the application on the specified hardware and the remaining four columns are run times predicted using hardware counters from four different platform. We see that the average mean absolute error for our prediction is 11.82 seconds and focusing on the fact that our model over predicts in majority of the cases thus giving a upper bound to the run time. From a quantitative stand point, our model predicted GPU as the best architecture, CUDA as the best programming language and GPU-CUDA as the best implementation with is in sync with the observed results.
<table>
<thead>
<tr>
<th>Platform</th>
<th>Actual</th>
<th>CPU</th>
<th>CUDA-K40</th>
<th>CPU-OMP16</th>
<th>PHI-OMP240</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>119.11</td>
<td>NA</td>
<td>85.54</td>
<td>126.72</td>
<td>161.37</td>
</tr>
<tr>
<td>CUDA-K40</td>
<td>0.0567</td>
<td>0.1796</td>
<td>NA</td>
<td>0.2164</td>
<td>1.2617</td>
</tr>
<tr>
<td>CPU-OMP08</td>
<td>38.29</td>
<td>42.51</td>
<td>33.14</td>
<td>41.73</td>
<td>65.69</td>
</tr>
<tr>
<td>CPU-OMP16</td>
<td>34.26</td>
<td>40.44</td>
<td>29.24</td>
<td>NA</td>
<td>59.13</td>
</tr>
<tr>
<td>PHI-OMP120</td>
<td>51.18</td>
<td>43.61</td>
<td>39.79</td>
<td>44.59</td>
<td>48.68</td>
</tr>
<tr>
<td>PHI-OMP240</td>
<td>47.59</td>
<td>40.38</td>
<td>33.93</td>
<td>39.44</td>
<td>NA</td>
</tr>
<tr>
<td>PHI-OMP136</td>
<td>0.2491</td>
<td>0.3391</td>
<td>0.1682</td>
<td>0.3076</td>
<td>2.3791</td>
</tr>
<tr>
<td>PHI-OMP272</td>
<td>0.1843</td>
<td>0.3024</td>
<td>0.1428</td>
<td>0.2573</td>
<td>2.1399</td>
</tr>
</tbody>
</table>

Table 4.10: Run time prediction results (in seconds) for real life application using Hardware Counters collected from four different platform.

4.3 Chapter Summary

In this chapter we describe the details of the Algorithm-to-Architecture (A2A) mapping and quantitative model for A2A mapping. We begin by describing the classifications on both the algorithm and architecture space and elaborate on the proposed mapping techniques. We further analyze the qualitative A2A mapping using the benchmark applications described in Chapter 2 over three accelerator based supercomputers; Titan, Stampede, and Palmetto. We then utilize these applications and the hardware parameters collected on the aforementioned machines to perform quantitative and qualitative analysis.
Chapter 5

Heterogeneous Functional Partitioning (HFP) Framework

The path to exascale computing presents new architectural constraints that require fundamental and radical recasting of system software services to allow continued scaling and performance in the face of ever shrinking power budgets and reliability issues. For CMOS circuits it is increasingly difficult to move data across silicon. This is a long known consequence of “Dennard Scaling” [62].

Larger fractions of system power must be dedicated to on-chip data movement and the relative distance (measured in clock-cycles) from different parts of the chip’s silicon increase dramatically. The total system power has also begun to increase with system size at a Moore’s law rate. For an exaflop machine, the cost induced total system power requirement has been set at $O(20)$ MW.

The most unfavorably scaling power use is for data movement inter-node and intra-node [63], so much so that by 2018 the proposed exascale platform’s power requirements will be dominated by the need to move data efficiently.

The data movement problem is severely compounded further by the end-to-end application execution that is needed to derive scientific insights. An end-to-end application execution involves many more tasks than just the main application simulation such as data analysis, post-processing, and simulation assist tasks for the health and reliability of the run. For exascale computing, such
operations are likely to become more complex, incurring significant data movement costs, both within and off the compute node, and may also involve redundant I/O to the storage system. Thus, there is the need to schedule an end-to-end application execution on future machines in a way that is aware of the data movement costs.

These changes construct a hard limit that present significant design challenges. Some responses seem clear such as increased core/node trends and an architectural migration to high flop/watt design such as seen in CPU/GPU systems. For example, both Jaguar at Oak Ridge National Lab (ORNL)—No. 1 on the June 2010 Top500 list—and Sequoia at Lawrence Livermore National Lab (LLNL)—No. 3 on the June 2015 list—were built with 16 cores per node; Titan at ORNL—No. 2 on the June 2015 list—comprises of a GPU and 16 CPU cores on each one of the 18,688 nodes, to achieve 17.5 petaflops; and SUMMIT, the O(100) petaflop system to be deployed at ORNL in 2018 will contain multiple GPUs and many-core CPUs per node. However, these trends themselves exacerbate some issues.

Increase in cores does not automatically translate into more effective FLOPS. Even on today’s common platforms, with 8 to 16 cores per node, it is challenging to effectively utilize the available compute power. This is clearly shown for big scientific applications such as mpiBlast [64] and FLASH [65]. Both of the aforementioned applications are I/O-intensive and compute-intensive, and were not able to use beyond 4 cores out of the 8 in an experiment we conducted on the previous Jaguar system. Experiment further suggests that even leaving one core per node idle in an octa-core node does not impact application performance [44]. This is because many current applications such as mpiBlast and FLASH were designed in an era when each node had only a single processor, and face severe resource contention at the memory and other levels of the storage or communication hierarchy, when simply deployed to occupy several cores in an SPMD manner.
Therefore, performance scalability and effective core utilization are serious issues even on today’s platforms, let alone on the emerging 60-core nodes in the Cori system at NERSC (in 2016) or the $10^3$ cores per node expected at exascale.

To address both the data movement and the core under-utilization problems, we propose a runtime architecture for an end-to-end application execution, wherein all end-to-end application assist tasks and system services are distributed across all system nodes. End-to-end data and computing tasks necessarily must be broken into smaller parts and distributed across the nodes. A key component to maximizing system performance within a given power envelope will be to be “smarter” in deciding when and how to move data between nodes (and even within nodes). Services must be represented on each node as well.

Not only will end-to-end application tasks be required to be co-located with data, but in addition all system services (e.g., resiliency, monitoring) will also need to be co-located with the distributed data (or at least be represented by functional stubs that facilitate the needed system services.) This is different from many current notions of in-situ analysis that divides the entire system into subsets of nodes for various functions (e.g. Adios). This centralized approach may be less appealing given the costs above. Therefore, while partitioning is a requirement, the exascale implication is that partitioning must not be done by assigning sets of nodes to functions, but rather must be present on each and every node and that within the node different cores and/or different threads will be assigned these tasks. This is the only way to hope to preserve needed data locality. Thus our design premise is: All application and service tasks must be distributed, along with the data, to each node in a scale-invariant fashion. An application-specific approach to in-situ execution, while on-node, is a custom solution that is generally tightly integrated with the main simulation code, and cannot use legacy analysis routines or other services in a scalable fashion. Instead, we propose a runtime framework that is generic, and would allow applications to co-execute end-to-end tasks in a scalable fashion. Lastly, our framework allows integration to large scale production applications with minor modification and restructuring to the code thus making easy for integration compared to other frameworks such as Legion, Charm++, etc.

The primary contributions of this framework can be summarized below:

1. We present the heterogeneous functional partitioning (HFP) runtime framework that is designed to leverage the under-utilized resources and reducing data movement on a heteroge-
neous nodes of a supercomputer.

2. We discuss key design considerations for such a framework and our approach to accelerating the end-to-end workflow of scientific applications with minor restructuring of the code thus allowing easy integration with production applications.

3. We evaluate several features and optimizations that make the HFP framework efficient, high performing and low in performance variation (jitter) for the main application.

4. We showcase HFP integrated with CESM, Global GPU-based Gene network alignment to enable post-processing and in-processing task.

5. We also discuss how other services like checkpoint encoding, incremental analysis in ensemble applications can leverage our HFP interface.

5.1 HFP Framework Design

The HFP framework exploits the on-node heterogeneity for efficient end-to-end computing for exascale computing.

5.1.1 Usage Model Considerations

Keeping the production ready scientific applications in mind, we design the HFP framework to support the integration of post-processing services in the workflow with minimal modifications. HFP considers scientific applications that generate several output files periodically. Most of these files are part of desired output from simulation, which may double as checkpoints. Where as some applications generate separate output files and checkpoint files. A typical scientific workflow runs a set of simulations and correspondingly the post-processing is performed as needed on the output dumps to process the data for analytics and visualization. We tailor the design of HFP framework to enable such post-processing tasks to be carried out when underutilized resources are present on a node.

A key consideration here is the ability to specify only the data elements and the operations on the data is only specified as a tag in the calls to HFP framework. This flexibility lets the user run the post-processing (e.g. sorting, statistics, data-mining) to be specified via a configuration file. In
5.1.2 Components of HFP Runtime

The HFP framework has four major components: HFP-agent, HFP-threads, HFP-management and HFP-queues. In this section, we will describe these components and how they allow HFP to scale efficiently into exascale computing. Figure 5.3 presents a high-level view of how HFP runtime receives, launches, and manages the post-processing tasks.

Table 5.1: Basic Function Definition of HFP.

<table>
<thead>
<tr>
<th>function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HFP_init</td>
<td>Initialization of HFP via HFP-Agent</td>
</tr>
<tr>
<td>HFP_post</td>
<td>Create a new HFP-thread</td>
</tr>
<tr>
<td>HFP_queue</td>
<td>Queue a task or data to user-defined queue</td>
</tr>
<tr>
<td>HFP_createQ</td>
<td>Create a new task- or data-queue</td>
</tr>
<tr>
<td>HFP_finalize</td>
<td>Cleanup of HFP-Agent, threads and queues.</td>
</tr>
</tbody>
</table>

this manner, a recompile of the application is not needed and only via the change of configuration the workflow can be altered. This is essential as we consider more complex workflows being managed by Workflow Management Systems such as Pegasus [66].

Figure 5.2: HFP Design Overview
5.1.2.1 HFP-agent

The HFP framework is designed in a hierarchical fashion where each thread of the scientific application communicates to the node-local daemon called HFP-agent. HFP-agent is a per node process launched on initialization of an HFP service and responsible for dynamically managing the status of the post-processing tasks based on the utilization of CPU, GPU, and memory. On initialization and launch of HFP-agent, all application thread connects to HFP-agent with process run information. This allows HFP to obtain the overall physical binding to allow better management of HFP-task and resources. HFP-Agent on launch also reads the HFP configuration file which contains the management schemas among other details for different scenarios along with function name and dynamic library location. This pseudo-pair of function name and dynamic library location is assigned a pseudo name by the user that is used to specify type of service for the HFP-threads.

5.1.2.2 HFP-threads

HFP-threads is the main compute unit for our framework. When the HFP-agent receives data from main application threads and type of service, it launches the corresponding task in a Pthread called HFP-threads. The core-binding and other feature implementations to improve performance of HFP-threads and reduce impact on main application for HFP-thread is performed by HFP-agent and discussed in detail Section 5.1.5. In order to receive the data from main application threads and reduce the impact of data-movement on the node we utilize the shared memory communication using the Common Communication Interface (CCI) [67] to transfer the data to HFP-agent. The shared memory transport layer described in detail in 5.1.4.1 allows fast and efficient communication to HFP-agent. HFP-threads receives the task identifier and the data from the HFP-agent, the rank allows informs the service to be performed.

5.1.2.3 HFP-management

The HFP-management is our resource monitor and feedback system. It is a sub-process of HFP-agent that regularly monitors the health of the system including main application threads, HFP-agent, HFP-task, GPU utilization, and memory management. The HFP-management performs dynamic controlling of the resource to prevent adverse performance impact due to HFP. The user can provide threshold and parameters in the configuration file that allows HFP to govern the HFP-
agent and HFP-task to be within a limit using thread priority level, thread context switching, and thread interrupts. This is discussed further in Section 5.1.5.

5.1.2.4 HFP-Queues

In order to maintain task-dependencies, data-streaming, and management, HFP utilizes single linked-list to maintain two types of queues: task queue and data queue. The data queue can further be broken into system queue (default-queue) and user-defined queues. The preparation of HFP-thread involves data transferring, data queueing, and task queueing. HFP task queues allow for queuing task for task scheduling, task priority, and task dependency based scheduling. By default HFP thread creation via HFP posts data into the system queue, however we allow for over-riding with user-defined HFP Queue. Thus, a user can create multiple HFP queues using to allow posting of data from multiple ranks in a continuous manner to the HFP Queue. HFP Queue allows for streaming and scalable processing defined in the next section.
5.1.3 Key Design Features of HFP Runtime Framework

In this section, we describe the key design features for HFP that allows for scaling and substantial performance gain for applications.

5.1.3.1 Dynamic Resource Management

In order to ensure minimal impact of post processing task on the performance of the main scientific application, it is imperative to perform dynamic monitoring and active resource management. HFP enables this by utilizing Dynamic Resource Management.

5.1.3.2 Monitoring Resources

In order to fully utilize the available resources it is essential to identify and utilize the slack periods in compute resources while running the primary application. As mentioned earlier, an HFP-monitor performs periodic thread monitoring of the resources. The periodicity and locality of monitoring can be specified and modified in the HFP configuration file. HFP utilizes the identification of slack periods to perform load balancing, resource switching, and core pinning as described below.

5.1.3.3 Load Balancing

In order to maintain optimal performance from the available resources it is imperative to balance the HFP-thread work load across all available resources. Often certain HFP-threads end before other HFP-threads due to characteristics and type of tasks, thus creating a slack period for certain resources, while other resources are burdened with multiple HFP-threads. We utilize HFP-monitor to identify and balance these imbalances in workload.

Furthermore, we utilize the load balancer to match complementary tasks such as I/O and compute, or compute and memory together in order to reduce workload imbalances. We achieve this mix-and-match of tasks using information of the post-processing task from the user during pre-runtime in the HFP configuration file. Using this user-defined characterization of HFP-threads we achieve better performance by load balancing.
5.1.3.4 Resource Switching

Slack periods on resource leads to under-utilization of available resources and correspondingly to higher run-time for applications. Often applications utilize a single type of compute resource leaving other resources un-utilized. In order to ensure utilization of all compute resources, HFP allows task to be posted for CPU-GPU computing environment.

Users can provide alternate task compute programming models. Thus, a user can provide a CPU, Multi-CPU or GPU version of the same task. HFP utilizes these various multi task implementations to allow seamless transition of task between CPUs and GPUs during detection of long slack periods. These transitions and slack periods can also be identified by users in code.

5.1.3.5 Task dependency and priority specification

During cases when multiple post-processing tasks compete for the resources, HFP provides users the capability of specifying priority for each post-processing task. Once the HFP daemon is ready to launch the next task, it observes the task list and launches the task with the highest priority. This can be specified by the user in the configuration file.

Along with setting task priorities, users also have the capability to assign dependencies to each task. This can be essential when a post-processing task is dependent on the output of another task, namely the pre-launch requirement task. HFP allows task dependencies to be specified on a per rank basis, thus for each rank the user can specify pre-launch requirements. These dependencies can be specified during run time by utilizing the return id obtained from the HFP post of the pre-launch requirement task, and passed as a parameter to the current task. This allows for the current task to access output of all its dependencies before being launched.

5.1.4 Communication Substrate via CCI

The heterogeneity in resources available on the node is the key motivation for the HFP framework. Because the cost of data-movement is increasingly higher as we approach exascale [68], this design philosophy enables us to minimize the communication of data by keeping the post-processing tasks node-local. The HFP agent orchestrates the collection of node-local statistics via the HFP-threads. If the post-processing algorithm requires the communication of data/information across nodes, HFP-agents are responsible for communication. In applications with multiple MPI
ranks per node, this results in significantly less communication time.

5.1.4.1 Shared Memory Transport

HPC systems include specialized interconnects, which typically vary between vendors as well as between new generations of systems from the same vendor. Each interconnect provides a low-level programming interface, that applications can use to get the lowest latency and highest throughput, as well as sockets. The low-level or native interface achieves much of its performance advantage over sockets by providing application direct access to the hardware (i.e. kernel-bypass) as well as zero-copy, one-sided data transfer (i.e. remote memory access or RMA) via PUT/GET or READ/WRITE semantics.

The HFP framework uses CCI to manage data movement between the application and the HFP-agent. CCI is a network abstraction layer (NAL) that allows an application to take advantage of high-performance interconnects when available or fall back to sockets when they are not. CCI provides two-sided (i.e. send/receive) semantics for small messages and one-sided (Read/Write) semantics for large messages. Using CCI shields an application from having to implement a NAL directly. CCI supports multiple interconnects including Cray’s Gemini and Aries, OpenFabric’s InfiniBand and RDMA over Converged Ethernet (RoCE), as well as sockets (TCP and UDP).

For this effort, we added shared memory (SM) support to CCI. By default, the SM transport uses a mmapped “bounce buffer” for RMA. One process copies the data into the bounce buffer and notifies the other process that the data is ready, which then copies the data out of the buffer. Using two-copies for RMA is less than ideal.

Fortunately, there are single-copy options for node-local, process-to-process data transfer. We are aware of three alternatives: Linux’s Cross Memory Attach (CMA) [69] starting with 3.2 kernels, the knem library [70] which supports 2.6.18+ kernels, and libxpmem [71]. ORNL’s Titan runs a 2.6.32 kernel, so CMA is not an option. While knem is supported on this kernel, it is not currently installed on Titan. Cray does provide libxpmem and we added support for it. If configured and available, the SM transport will use it. It improves RMA performance from 2.4 GB/s using the bounce buffer to 7.8 GB/s for in-cache transfers and 6.6 GB/s for transfers larger than can fit in cache.
5.1.5 Dynamic Control of Performance Interference

While HFP enables co-scheduling of main scientific application and HFP tasks to leverage the underutilized resources on the compute-nodes, it is important to ensure minimal degradation of the primary simulation. Previous work such as [72], [73] address this issue in the context of co-scheduling on a homogeneous multiprocessor node. Since HFP is designed to consider the heterogeneity in the resources (CPU vs. GPU, DRAM vs. Non-volatile memory), HFP-agent schedules the HFP tasks on these heterogeneous resources in consideration and performs periodic monitoring of system resources including memory, caches.

5.1.5.1 Core Binding

One of the basic and intuitive control mechanism is core binding. Two tasks running on the same physical core results in interference between the two, resulting in degraded performance. In order to prevent overlapping of scientific application and HFP-task or HFP-agent, the HFP-agent uses the core-binding information from the scientific application to mark cores not to be used (NTB) on each node during the scheduling phase as shown in the Figure 5.3. The HFP-agent then guarantees that no HFP-task or agent itself will be scheduled on these cores. Furthermore, the HFP-agent schedules HFP-task on core available further away from NTB cores in order to prevent cache poisoning and excessive bandwidth consumption. The effects of core-binding and scheduling can be seen in Figure 5.4.

Figure 5.4 showcases variation in Million Instruction Per Second (MIPS) for overlapping for two different varying types of task. We ran the base task and secondary task on each core with the secondary application being 1/10 less time consuming than base task. We performed this evaluation for four different base task (Integer CPU, Double CPU, Memory, and IO) with combination of five different secondary task (None, Integer CPU, Double CPU, Memory, and IO). We found that on Titan Supercomputer the biggest variation in MIPS occurs for Integer CPU with any secondary task. On average any secondary task has a 25% increase in performance however with DCPU, MEM and IO there is only on average 1% increase in performance.
5.1.5.2 Task Queuing

Queuing can occur in two ways, firstly if the number of HFP-threads running at single time period exceeds the user defined value in the configuration file. Secondly, if the user requires HFP-thread to perform computation intensive tasks during the scientific application compute phase. This user defined prompt allows HFP to prepare the HFP-threads but launch it when defined by the user. The preparation of HFP task involves data transferring, data queuing and task queuing. We achieve this by using a HFP-token. When a scientific application rank post data to HFP-agent to process with a queuing argument, a unique HFP-token is provided to the application and HFP-threads is queued on completion of data movement via the shared memory transport, and can be redeemed to launch the queued HFP-thread when desired. Queuing often results in memory storage for HFP-threads, which can be controlled in HFP configuration using a maximum queue size and maximum queue memory.

5.1.5.3 Data Queuing

As defined in the above section, HFP has two grouping of queues: system queues and user-defined queues. When a HFP post is called, the user has option providing a data or override with a new user-define queue. If data is supplied, then data is added to system queue and a pop is
performed when HFP-thread is created for the corresponding task. However HFP user-defined data queue allows for posting and popping data queue in continuous streaming order. Furthermore, we allow user to define the type of pop for user-defined HFP queue entry-removal based popping or index-based retrieval. The entry-removal based popping HFP queue removes the entry does reduces after each HFP_getnextdataset thus reducing the memory footprint and allowing unique copies of HFP-thread to perform any task on the data. However index-based retrieval queue allows for multiple non-unique non-duplicate HFP threads to process the same data differently.

5.1.5.4 Interference Control

Running a secondary process along with main scientific application can result in performance degradation of the resources available to secondary process that are not throttled. Such a scenario can exist commonly in heterogeneous environment when a task is offload to GPUs leading to low or zero-utilization of CPUs. Thus, if an HFP task is scheduled on these cores and the main application returns after execution of the offloaded task, hence resulting in degraded performance of main application. Using thread priority, yielding and scheduling using interrupt we can control the resource allocation to HFP-threads using configuration parameters in HFP configuration file.

5.1.5.5 Smart Execution

Using user-define HFP data queues, we enable smart execution is a capability of HFP-thread. Smart execution allows optimal choice of execution given an option HFP-thread on CPU vs HFP-thread on GPU. Smart Execution can choose based on availability and user prompt. We allow the users to provide multiple implementation of an application supporting a wide range of programming models such as OpenACC, OpenCL, CUDA and OpenMP. A user can thus provide on or multiple implementation to allow for seamless switching between resources based on the availability of resources. If the HFP-management detects unused GPU the HFP-thread can launch the GPU version instead of CPU version and vice versa. If a running HFP-management intercepts and detects a main scientific application request to use GPU, then based on HFP-configuration, the GPU based HFP-thread can run concurrently or kill itself and restart execution on CPU based on NTB cores by the HFP-agent.
5.1.5.6 Dynamic Load Balancing

In order to provide dynamic assistance to current working threads we propose dynamic load balancing mechanism. Enabling dynamic load balancing is a per thread schema using HFP user-defined queues, which can be enabled in the HFP configuration file. Enabling dynamic load balancing allows HFP-management to start multiple copies of the same thread to work on different sets of data. All dynamic load enabled HFP-threads post data to double linked list. Using HFP HFP_getnextdataset() inside the HFP-thread user can obtain the next set task to be processed.

Using dynamic load balancing we can sustain continuous utilization of cores in a dynamic way. Consider a scenario where 16 cores are available while 8 NTB and 8 available for HFP. Each rank allows one HFP-thread (thread A) to perform task A. At a later time each rank posts HFP-thread (thread B) to perform task B. If both thread A and thread B are dynamic load balancing disabled, then thread B might be posted to queue and wait for thread A to finish. Another alternative is thread A and thread B run simultaneously on 8 cores, which might be harmful to both threads. Therefore, if dynamic load balancing is available for thread A and thread B, then both can be reduced to 4 HFP-threads for simultaneously performing task A and task B.

Dynamic load balancing also allows us to perform scaling for threads launched by only one rank per node. In such a scenario, only rank per node launches task A to work on multiple data set from all rank on the node, thus with no dynamic enabled HFP-thread, only 1 thread would run with other cores being non-utilized however using dynamic load balancing we can launching assisting HFP-threads to assist in completion task.

Along with mentioning the dynamic load balancing per HFP task in HFP configuration, the user also defines the maximum number of assisting threads to be launched. The HFP management based on utilization of the cores can then use these cores when available and also reduce the assisting HFP-thread if required.

5.2 Jitter Analysis

We evaluate the efficiency of the HFP framework on the Titan supercomputer by quantifying performance jitter and further propose mitigation strategies. We created four different types of client and post-processing micro-benchmarks for our experiment:

(i) Integer CPU Intensive (ICPU), (ii) Double Floating Point Intensive (DCPU), (iii) Memory
Figure 5.5: Runtime for varying number of ranks with range of base application (ICPU, DCPU, MEM, IO micro-benchmarks) and post-processing micro-benchmarks.

Intensive (MEM), and (iv) I/O Intensive (IO). The client (baseline) micro-benchmarks are 5 times more time consuming than their corresponding post-processing micro-benchmarks. The baseline ICPU and DCPU micro-benchmark perform 14 billion arithmetic and floating point operations per process. The baseline memory micro-benchmark performs 750 million random read and writes to memory, and the base I/O micro-benchmark performs 225 million write operations to a file. Figure 5.5 represents the effect of rank scaling per node for various different types of tasks in combination with different types of clients. Each rank launches at a maximum of one HFP task or Post-processing task. We observe for all scenarios i.e 2, 4, 8, 12 ranks, the run time for application using HFP is less than performing corresponding post-processing without HFP; except in case of I/O and memory intensive clients. We believe that this is due to the fact that currently all HFP tasks are performed in Pthreads, which decreases the performance of I/O and memory reads/writes. Furthermore, due to absence of local node storage, there is an increase in network and memory activity to buffer small writes before writing to disk. Thus, performance of I/O is dependent on network bandwidth and the memory bandwidth of the node which can be limited when shared with the client application. In Figure 5.5d we observe degradation in performance due to over-subscription. Since there are only 16 cores we launch 12 additional process to the 12 base processes. There are 24 processes spread across the 8 non-bound core thus leading to degradation in performance due to over subscription.
Figure 5.6: Incremental Performance improvement with benchmark applications on Titan at ORNL.

The performance is worse for cases involving I/O and MEM for the same reason as stated above.

## 5.3 Experimental Setup and Results

We perform incremental analysis using two applications: Myocyte [74] application that models cardiac muscle cell and simulates its behavior and LavaMD [74] code that calculates particle potential and relocation due to mutual forces between particles within a large 3D space. The Myocyte application is a GPU intensive code where as LavaMD is distributed over a only CPUs. For both the applications we use GEMM and covariance as the post-processing tasks referred to as task1 and task2. The applications are evaluated over incremental implementation of HFP features to demonstrate each of their effects on overall runtime. Figure 5.6 depicts the performance runtime for both the applications with and without incremental HFP features.

The first three bar plots represent the runtime of original application, original application + 2 tasks run consecutively, and original application + 2 tasks run concurrently. The next bar plot launches the HFP application, which increases the runtime by adding HFP overhead. Adding the first HFP feature, task queuing, does not affect the performance runtime for both the applications. Since the launch of post-processing tasks are user defined, task queuing does not improve performance since no idle resource is identified. Launching the two tasks at any instance results in the same overall runtime.

The next feature incremented is core binding, which results in significant performance improvement in both the applications. For Myocyte, since the primary application is run solely on GPU, the tasks can be bound to the idle CPUs. LavaMD application was launched on odd cores and
thus the non-utilized even cores are available for the tasks. The next incremental feature evaluated is smart execution. Myocyte does not get affected by this feature since the primary application is already executed on the GPU and tasks allocated to CPUs. LavaMD on the other hand experiences performance gain since GEMM task is executed on the GPU while the main application continues on the CPU cores. The last feature is load balancing, which results in performance gain for both the applications. Multiple sub-task threads are launched for GEMM and Covariance, where each thread independently identifies idle resources.

### 5.3.1 LULESH Results

In order to evaluate efficiency of HFP, we utilize a proxy-application LULESH with MPI and MPI-CUDA version on Titan at ORNL. LULESH [75] represents a typical shock hydrocode that approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. LULESH is a highly simplified application, hard-coded to only solve a simple Sedov blast problem with analytic answers, but represents the numerical algorithms, data motion, and programming style typical in scientific C or C++ based applications. Application such as LULESH work on multiple of threads ranging from $1, 8, 27, \ldots, n^3$. Thus, leading to possibility of underutilized resources on a node. If we consider the MPI version, we have available cores as well as an available GPU. On the contrary, if we
To utilize MPI+GPU version we have tremendous number of underutilized cores on multi-core GPU supercomputer such as Titan. We utilized these under utilized resource by using HFP to perform two geometry processing algorithms and visualization dump using SILO and HDF5. The two algorithms being closest pair reduction and convex hull algorithm. We utilized both a C version and GPU version for the above mentioned algorithms. The pair reduction is node-local task that performs reduction and while the convex hull algorithm requires global mesh. Lastly, output dump writes an output every 10th iteration of the global mesh. Figure 5.8 shows the output of LULESH. In order, to achieve this we create two HFP queues one for local and one for global mesh. On each iteration, we change the node responsible for performing task on the global mesh, thus balancing the computation across the nodes. We show in the Listing 5.1 below the ease of use of HFP with minor restructuring with existing scientific application, along with the performance in Figure 5.7.

Listing 5.1: Annotation of LULESH with HFP.

```c
.. ...
int main(int argc, char *argv[])
{
...
...
#ifdef USE_MPI
    Domain_member fieldData ;

    MPI_Init(&argc, &argv) ;
    HFP_Init() ; //Initialize HFP on all nodes
    MPI_Comm_size(MPI_COMM_WORLD, &numRanks) ;
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank) ;
#else
    numRanks = 1;
    myRnk = 0;
#endif
```
As shown in Figure 5.7, HFP integration with LULESH gives better performance by approximately 10% on average compared to in-processing version and post-processing version. The in-processing version is simple glorified p-thread version to do task during the processing step with no smart execution. The post-processing performs better than the post-processing due to time-delay in transferring the data to remote cluster however, it does not impact the performance of the main scientific application. HFP can utilize and scale its task based on availability of resources. Thus during the MPI-CUDA version of LULESH, 14 cores are available for execution assuming one core is binded as slave to the GPU and the one is binded for OS related task during aprun. We also see that impact of in-processing is higher on the 2-node MPI only implementation due to limited number of cores available. However, HFP can switch between convex hull on GPU and least distance pair algorithm on GPU and utilize the cores in a scalable manner. However, we do encounter penalty for transferring the same data has to be transferred to GPU for different algorithms and we plan to address in the future work.

In later chapter, we describe integration of HFP with real world scientific applications.
5.4 Chapter Summary

In this chapter we describe the details of the Heterogeneous functional partitioning (HFP) framework and its design. We elaborate on the HFP runtime components along with the key design features. We also evaluate the efficiency of the HFP framework on the Titan supercomputer by quantifying performance jitter and further propose mitigation strategies. Lastly, we utilize the Lulesh application to demonstrate ease of framework integration in existing code.
Chapter 6

Case Studies using Scientific Applications

In this chapter, we showcase utilization of our enrichment frameworks of A2A and HFP to assist scientific applications in accelerating performance and utilizing heterogeneous framework in an efficient manner.

The first application discussed is (GPU-based Global Gene Alignment) G3NA and its visualization. We use A2A to develop G3NA for an optimal architecture. We then described the incorporation of HFP into the G3NA work-flow. The second application discussed is Remote Laproscopic Surgery using HPC that utilized A2A in the initial development of the application to in order to accelerate development. The last application we describe is Climate Earth Science Model (CESM) and incorporation of HFP into CESM with the help of researchers at Oak Ridge National Laboratory (ORNL). Performance results for each applications are discussed in the subsequent sections below.

6.1 Large Scale Gene Network Alignment using GPGPUs (G3NA)

Gene interaction graphs help genomics researchers discover specific gene interaction patterns across multiple developmental stages and treatment conditions. Graph alignment between species under analogous conditions reveals conserved and divergent subgraphs controlling bio-
logical processes. Graph alignment between gene expression states from the same species reveals common cross conditional subgraphs as well as subgraphs restricted to organ or condition specific expression patterns. This powerful measurement approach is limited by computational resources and requires optimized topology detection algorithms. Alignment of medium sized graphs (2000+ nodes) is computationally expensive and the alignment of large graphs can quickly become unrealistic without the use of advanced computing resources. Thus, we have developed a GPU based optimized framework to perform multiple global network alignment and describe the GPU-enabled Global Gene Network Alignment (G3NA) algorithm in detail. As a representative application of our open source code to medium-scale graphs, we aligned two gene interaction networks from rice and maize (each 2,000 nodes/20,000 edges) on the Palmetto Supercomputer using NVIDIA K40 GPUs. Maize-rice alignment took 34.2 seconds compared to 21,800 seconds with IsoRankN, a speed-up of 672.8x. To extrapolate to larger graphs, we aligned large random scale-free networks (8,000 nodes/60,000 edges) and observed a speed-up of 51.3x.

6.1.1 Motivation

A fundamental goal of biology is to understand complex gene product interaction to discover the mechanisms of phenotype expression. Gene products include RNA transcript concentrations, measured by nucleic acid hybridization or high-throughput DNA sequencing, and as processed peptide molecule expression levels, as measured by a variety of proteome quantification strategies.

Genes with expression dependencies (e.g. significantly correlated expression levels) can be woven into gene interaction graphs where a node is a gene product and an edge indicates dependent expression. Plant RNA gene co-expression networks (GCNs) have been constructed for Arabidopsis [76, 77, 78, 79, 80, 81], barley [82], rice [83, 84, 85], maize [86], tobacco [87], and several more species. Networks can be constructed in an intra-species manner from conditionally-mixed datasets or grouped by similar RNA source such as from the same organ, experimental treatment, or genotype prior to network construction. By dissecting the network into non-randomly interlinked communities, a researcher can identify groups of genes assumed to be involved in coordinated biological processes. Common topology between inter-species graphs imply conserved processes (i.e. remnant of ancestral coordinated expression), and common topology between intra-species
graphs implies common processes between organs, treatments, and genotypes.

Comparative gene expression network analysis has proved to be a challenging task for 2 major reasons. Firstly, computational resources and time required to process scales exponentially with network size. Secondly, understanding the complex biological significance of network comparisons is not straightforward. Several techniques have been proposed and supplemented by promising new computational approaches [55, 88, 89, 90] to perform alignment. Our algorithm is particularly inspired by IsoRankN [53]. The details of our algorithm implementation and its corresponding visualization tool are detailed in the subsequent section.

## 6.1.2 Implementation

Since there is information at node level (gene sequence similarity) and edge topological level (gene interaction pattern similarity), a useful approach is to align networks and discover common and divergent patterns of gene expression using both homology and topology criteria. In this report, we describe the implementation and application of a fast GPU-enabled Global Gene Network Alignment (G3NA) algorithm using these criteria. The algorithm can be divided into three stages: pre-processing, computation stage, post-processing stage. The pre-processing and post-processing stage uses the CPU to perform conversion of input data to and from matrices. Due to quick alignment of networks, this serves as a pre-staging area to identify nodes or clusters of interest. The computation phases is performed completely on the GPU and involves statistical information of the input graph followed by generation of similarity graph using homology and topological matching and calculation of statistical information of the input graph and similarity graph. Followed by a clustering algorithm on the similarity graph to obtain an aligned graph. We perform global graph alignment using pair-wise network alignment. A WebGL-based in-browser visualization is performed using JavaScript and is available at network.genome.clemson.edu. The compiled G3NA binary is available on G3NA website as well.

The study of gene product interactions is fundamental in understanding coordinated gene output for phenotype expression. The research challenge in creating and comparing gene networks lies in the lack of knowledge of how each node of one network maps to one or more nodes of other networks. This absence of information leads to solving the subgraph isomorphism problem. Isomorphism refers to a bijection function that preserves edge adjacency, between nodes of the
two networks being aligned. Since exact comparisons are inappropriate between the nodes due to biological variations, the challenge lies in “fitting” one network into another without an exact subgraph. Therefore, the need to develop an efficient and accurate multiple network alignment algorithm arises.

Pairwise graph alignment is performed to detect subnetworks, which represent co-functional gene modules, common to both graphs or specific to one graph. There are several classification categories with which network alignment can be performed. The first category is the global versus local alignment, where local alignment mappings are chosen independently for each local region of similarity. These local mappings can be ambiguous with one node having pairings with different local alignments. As described above, popular GNA algorithms include PathBLAST [50], Network-BLAST [49], MaWISh [48], Graemlin [51]. Global graph alignment referenced above provides better overall alignment from every node in one network to nodes in the other networks, which leads to sub-optimal matchings in certain local regions. Major GNA algorithms are GRAAL [56], IsoRank [52], IsoRankN [53], and Extended Graemlin. We implement a GPU-optimized GNA algorithm in our software implementation.

GRAAL, PathBLAST, MaWISh, and IsoRank also utilize the pairwise alignment where two networks are aligned at a time, similar to our implementation, but Graemlin, Extended PathBLAST, Extended IsoRank implement the multiple alignment algorithm where more than two networks can be aligned. Finally, the methods can be classified as “functional information” versus “topological information”. Functional information refers to when the information is external to the network topology used to define similarity between the nodes. All algorithms except GRAAL utilize functional information. The topological information utilizes only the network topology to define node similarity. In our algorithm we implement a hybrid of functional and topological information, which is further described in the following section.

Our network alignment algorithm can essentially be divided into two sections; the first section involves creating a network interaction matrix, and the second pertains to the process of “haircut” and “fluff” by utilizing the MCODE algorithm [91]. As mentioned above, our algorithm is based on global, pairwise alignment, therefore the user provides two gene product networks A and B, which we perform pre-processing on to convert them into machine readable binary matrices. Each node forms a matrix index, with the corresponding entry as a binary 1 if an interconnected edge exists between the two nodes. The user can also provide with a graph AB based on homology
information of the two genes encoding the gene product nodes.

Our algorithm first accepts the processed A and B matrices to formulate AB', which is the interaction matrix based on topological information. The AB' matrix is m x n matrix, for \( A_m \) and \( B_n \), where m and n represent the nodes in two gene product networks. To populate the AB' we determine the inverse sum for each node in A and B, using the equation The inverse sum sets a score for each node in both the networks. In order to establish the interconnection between networks, we require the user to set a topological threshold \( \tau \). Next, for each node in A, we traverse through n nodes in B, and for every jth node in B if it satisfies the condition for the ith in A

\[
|\text{InverseSumScore}[A_i] - \text{InverseSumScore}[B_j]| < \tau
\]

The node \( A_i \) and \( B_j \) are interconnected by populating their corresponding indices in the AB matrix as binary 1. We then generate the AB” matrix by setting interconnection values as 1 when

\[
AB'' = AB'orAB
\]

This simply states that we hold the nodes of two networks as interconnected if they satisfy either of the thresholds, homological or topological, provided by the user.

On establishing the interaction matrix we apply the MCODE to perform the haircut and fluff process. MCODE is a popular automated algorithm, which locates highly connected subgraphs as clusters in large networks. The algorithm involves three major steps. Step 1 assigns scores to nodes, where higher scores represent a node whose immediate neighbors are more interconnected. The algorithm first picks the immediate neighbors of the node under consideration, and of the neighbors calculate the highest k-core network, i.e. the maximum edges originating from one node. Next the Core Density is calculated by

\[
\text{CoreDensity} = \frac{\text{Edges}}{\text{PossibleEdges}}
\]

where Edges refer to the total number of edges present between the scoring node and its neighbors, where as Possible Edges is \( (\text{ScoringNode} + \text{Neighbors})^2 \). Finally, the node score is determined by
Score = \( K \times \text{CoreDensity} \)

Step 2 is for cluster finding, where the algorithm begins with a complex cluster with a highest scoring node as a seed. It then steps outward from the seed in a recursive manner, including the nodes whose scores exceed a preset threshold. This step is repeated where clusters are filtered out that contain less than k-core networks.

\[
\zeta = (1 - \text{NodeScoreCutoff})(\text{SeedNodeScore})
\]

Therefore, higher node score cutoffs will result in bigger clusters.

Step 3 pertains to haircut and fluff, where the haircut process removes all singly connected nodes from each cluster. Conversely, the fluff process determines the node densities on the neighboring nodes, and if the density is larger than Density Cutoff, those nodes are included in the cluster.

### 6.1.3 Experimental Setup

Clemson’s Palmetto GPGPU cluster was used in this research for the large-scale G3NA alignment. The 551 tera-flop cluster is composed of HP servers; each server is connected to two kepler-based K40m GPGPUs via a PCI-e Gen. Each server is equipped with two 2.4 GHz Xeon-E5 processors with eight cores each. The network interconnect is comprised of InfiniBand FDR. Our implementations were developed using CUDA 5.5.22 and OpenMPI 1.4.3 with the Debian Linux kernel.

In order to evaluate the performance of G3NA, we created various networks ranging in size from \(2^{10}\) to \(2^{20}\) nodes per graph. We generated these graphs using the Kronecker graph model. Kronecker graphs capture several well-known properties of gene interaction networks; in particular heavy tailed distribution along with a low diameter while obeying power law (i.e. scale-free graphs). Furthermore, we also aligned previously described maize, rice, and Arabidopsis thaliana gene co-expression networks, where both networks contain 2,000 nodes and 30,000 edges. Node homology for real graphs was measured with BLAST bit scores [92]
6.1.4 Preliminary Performance Results

We compare the performance of maize-rice and Kronecker generated random network alignments (4096 X 4096) for IsoRankN, Magna++ and G3NA techniques. As represented in Figure 6.1, G3NA achieves convergence in 34.2s and 36.7s, respectively in comparison to the Magna++ that requires 1271 and 1301 seconds respectively. Furthermore, IsoRankN takes 27800s and 29200s. Figure 6.2 depicts the distribution of percentage of time spent per operation. Since 88% of the operation time is in Edge Matching, we can claim that our deploying our algorithm on GPUs is essential, and we validate our claim by achieving 51.3x speedup. Figure 6.3 shows the increase in computation time based on the input of increasingly larger random generated matrices. We observe a very slow trend towards an exponential growth. Almost four times the input size (16,000) of IsoRankN (4000) tested with G3NA resulted in less than 2000s convergence time. We observe that 64 size networks alignment is slower than alignment of 128, 256 and 512 size networks. This occurs suboptimal memory access pattern due to 1024 threads per block size for the GPU and the overhead for Host to GPU and GPU to Host transfer time over the PCI-express. However, for network sizes larger than 128, our algorithm is better optimized. Lastly, in Figure 6.4, we depict the performance comparison of G3NA with its serial version and OpenMP version on Palmetto using 16 cores. We observed as predicted by Tesseract our initial endeavor to start with GPU implementation for the most optimal performance architecture. This allowed us to reduce time in providing the tool to the researcher.

![Figure 6.1: G3NA versus Magna++ versus IsoRankN alignment of Medium Sized Graphs.](image)

74
Figure 6.2: Distribution of percentage of time spent per operation in G3NA.

<table>
<thead>
<tr>
<th>G3NA (Match¹/Match²/Total)</th>
<th>IsoRankN-Native²</th>
<th>IsoRankN-Modified³</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maize Nodes</td>
<td>729/722/786</td>
<td>1585</td>
</tr>
<tr>
<td>Rice Nodes</td>
<td>492/488/549</td>
<td>1041</td>
</tr>
<tr>
<td>Edges</td>
<td>535/536/822</td>
<td>1586</td>
</tr>
</tbody>
</table>

Table 6.1: Edge Matching between G3NA and IsoRankN.

Figure 6.3: G3NA alignment of random scale-free graphs.

### 6.1.5 G3NA-Visualization (G3NA-V)

Along with addressing computational challenges, researchers need to face challenges posed by visualization limitations for displaying and exploring large-scale networks. To assist in ac-
In order to accelerate discoveries, we have developed a visualization tool called Biological Dependency Visualizer (BioDep-Vis). BioDep-Vis is a comprehensive tool for biologists exploring gene expression relationships within and between gene interaction networks. Along with visualizing aligned and conserved networks, our tool provides an interface to explore gene expressions in a deeper context such as an interactive molecular visualization, ontology hierarchies, and evolutionary trees. Due to the comprehensive nature of BioDep-Vis, the workflow typically utilized by researchers is streamlined where several types of data and visualizations can be interactively explored. The primary contributions include:

- Gene expression matrix distribution visualization.
- A multi-species graph alignment visualization including gene (node) based user interactions.
- Interactive molecular ribbon visualization linked to the genes.
- Multiple ontology database visualization.
- Comprehensive visualization tool that assimilates gene expression graphs, corresponding gene ontologies, and molecular visualization in an interactive manner for the discovery and validation of genomic information.
6.1.5.1 System Workflow

The streamlined workflow of BioDep-Vis is represented by Figure 6.5. The workflow begins with a set of genes for a given species measured in many biological contexts. Gene expression is quantified as an intensity value dependent on various factors and multiple intensity values for all genes are collected across multiple experimental conditions in a gene expression matrix. Thus a sample refers to intensities associated to all known genes for a given condition, which forms the input data for our gene expression matrix view. The heatmap is a common visualization tool in genomics and represents the distributions of intensities for all genes across different samples. Another representation of the gene expression matrix is given by the sample curve distribution.
Both these views can be utilized to explore and reduce data by eliminating outlier samples. In the set of reduced data genes, each pair of genes in the set is assigned a relation number based on statistical algorithms such as Pearson’s correlation coefficient. Filtering based on correlation threshold results in a GCN, where the node is a gene and an edge between genes indicates a significant correlation score.

A simple GCN is one of the basic visualizations generated by our tool. Using our G3NA algorithm we generate alignment data between intra/inter- species genes. This alignment data specifies an edge list where each edge comprises of a pair of nodes one from each of the original gene network graphs. The next stage of the workflow visualizes the aligned gene expression networks, where the genes from the two graphs are connected through edges. The detailed interactions for the alignment network are discussed in subsequent section. We further visualize the conserved subgraphs for each of the original networks along with clustering of the gene nodes.

The next stage of the workflow gives the user the ability to explore different ontologies associated to the genes. As an example we explore the Gene Ontology (GO) [93]. GO provides an ontology of defined terms (GO terms) representing gene product properties. The GO terms within the ontology has a term name, which may be a word or string of words; a unique alphanumeric identifier; a definition; and a namespace indicating the domain to which it belongs. Hence, GO graph is structured as a directed acyclic graph, and each term has defined relationships to one or more other terms. BioDep-Vis provides a two-way interaction between the aligned gene network nodes and the GO ontology.

Another associated view involves the molecular visualization interface. Molecular level visualization of gene nodes is achieved using the Protein Data Bank (PDB) archive [94]. The essential contribution to this visualization is its integration with the gene expression network. Finally, we visualize the evolutionary tree that is used to determine the hierarchy of alignment necessary to study the relation between any two species.

6.1.6 HFP Integration

BioDep-Vis visualizes multiple genomic networks, ontology networks and molecular representations etc in an interactive manner while utilizing compute and visualization power of GPUs and unused CPU resources. The ease of integration allows HFP to work as queuing system for
applications as well. We integrate with G3NA to utilize resources on dual-node NVIDIA GPU machine with Intel Xeon Co-processor. HFP is assigned task for alignment, filtering, clustering, and several graph processing algorithms on an impromptu basis by the users interaction with graphs. The tasks were written in OpenCL to allow flexibility in utilization of unused resources.

6.2 Real-Time Remote Processing of Laparoscopic Surgery

Laparoscopic surgery is a minimally invasive surgical technique where surgeons insert a small video camera into the patient’s body to visualize internal organs and use small tools to perform surgical procedures. However, the benefit of small incisions has a drawback of limited subsurface tissue visualization. Image-guided surgery (IGS) uses images to map subsurface structures and can reduce the limitations of laparoscopic surgery. One particular laparoscopic camera system is the vision system of the daVinci robotic surgical system. The video streams generate approximately 360 MB of data per second, demonstrating a trend towards increased data sizes in medicine. Processing this huge stream of data on a single or dual node setup is a challenging task, thus we propose High Performance Computing (HPC) enabled framework for laparoscopic surgery. We utilize high-speed networks to access computing clusters to perform various operations on pre- and intra-operative images in a secure, reliable, and scalable manner.

6.2.1 Motivation

Minimally invasive surgery (MIS) reduces patient trauma and recovery time through reducing incisions. However, small field of view of the laparoscope and incision size will result in a small visual field of underlying tissues of interest. Image-guided surgery (IGS) uses images to map a surgical region of interest, providing surgeons with visualization of subsurface structures [95]. The accurate co-registration of an IGS system with laparoscopic video allows for a resection with higher specificity [96].

We focus on the daVinci robot, which uses two parallel 1080p High Definition video cameras [97]. These video streams generate approximately 360 megabytes of data per second. Processing the data has become demanding for a local system, and high-performance computing (HPC) hardware is needed [98]. To process this data on remote HPC clusters at 30 frames per second (fps), it is required that each 11.9 MB video frame be processed by a server and returned within 1/30th of a
second for the left and right eye synchronization of the da Vinci. As a result, utilizing high-speed networks to access computing clusters will lead to real-time medical image processing and improve surgical experiences.

6.2.2 Implementation

6.2.2.1 Networking Framework

Steroid OpenFlow Service (SOS) is an OpenFlow-based network service that can seamlessly increase the performance of large data transfers over long-distance and high bandwidth networks [99]. TCP is typically unable to consume the available network bandwidth over large networks and only permits the connection source to send its window-size number of packets before receiving an acknowledgement. In a large network, this can result in the sender waiting idly for the acknowledgement before being permitted to send additional packets. SOS redirects the TCP connection to a local SOS agent using an OpenFlow switch in the network path.

As the source SOS agent (surgeon console) accumulates a buffer of data from the data transfer source, a destination SOS (HPC Cluster) agent is located in close proximity to the intended destination. The source and destination SOS agents communicate and agree to use a number of parallel TCP connections in order to rapidly transfer the data from the source SOS agent to the destination SOS agent. The destination SOS agent collects data from the parallel TCP connections and presents it to the intended destination as if it originated from the source across the large network.

6.2.2.2 Computing Framework

Our computing framework also enables reliability using replication of computation. Pre-surgery, our setup allows for transfer of per-operative images from a remote location to compute nodes and GPU Memory. We then allow warm-up of the networking protocol followed by verification of compute nodes capabilities and devices.

The compute nodes are broken into three layers: Main Compute Layer (ML) and the two Duplication Compute Layer (DL). Each layer contains node that performs duplicate computation. This computation is then verified with a quick checksum followed by verification using NVIDIA GPU Direct with compute nodes in the two DL. If the ML matches any one of DL calculation, the ML
overlay information is sent back to surgeon. In case of a failure of GPU, the computation between the two DL are verified and sent to the surgeon and an idle node in the same layer overtakes the computation task of the failed compute node.

6.2.3 Preliminary Performance Results

Based on our tesseract prediction, we implemented a system on Palmetto Supercomputer [61] using the Dual NVIDIA K40 GPUs per node, CUDA5.5 enabled with Intel Xeon and 32GB of Memory. Figure 6.6 shows runtime for HD images time of various algorithm of interest to the surgeon in CUDA where some algorithms have slower frame rates. To achieve a higher frame rate, we will enable GPU image resizing using Bi-cubic interpolation to reduce image resolution prior to the operation flowed by resizing to original image size. In order to reduce and resize to original resolution using GPUs took only 0.8msec. Furthermore if required, we can also enable compression of images allowing 1.3msec to perform compression and decompression to achieve a higher frame rate.

Figure 6.7 shows the performance of these algorithms. As seen in the figure certain algorithms such as Bilateral Filtering and Mean-shift segmentation can achieve strong performance gain. However, for other operations the reduction in image size does not improve the performance.

![Figure 6.6: GPU runtime of image processing algorithms.](image)

Hence, we develop a framework that enables reliable and secure processing of terabytes of information to aid in laparoscopic surgery using GPUs, SOS, and Open flow. The output of our
framework is an overlay that can be enabled or disabled by the surgeon at the surgical console. Furthermore, our framework enables replication to prevent failure and allow for scalability for additional operations/algorithms.

### 6.3 Climate Earth Science Model (CESM)

CESM is one of the widely used climate models and it incorporates a suite of earth system component models to represent processes of the atmosphere, land, ice (land and sea), and ocean [100]. CESM is widely used application in OLCF’s Titan Supercomputer at ORNL and it involves large runs involving post-processing stage on Rhea Supercomputing Cluster at ORNL. For the purposes of this project we focus our efforts on the atmospheric component of CESM, specifically the spectral element dynamical core of the Community Atmosphere Model (CAM-SE); a framework adaptable to arbitrary unstructured grids.

#### 6.3.1 Implementation

We enable new diagnostic post-processing using HFP framework. We provide different interesting precipitation statistics (reinterpolation) that are performed in-situ by launching HFP tasks on the GPU. It includes computing time averages of precipitation for each grid point for
several time windows, histograms/distribution of precipitation for each grid point, and finding the local time when the precipitation was maximum for a grid point. As current CESM version does not use GPU resources heavily, we can move these tasks to GPU using the HFP framework. Each rank running the atmospheric model in CESM pushes the precipitation vectors via HFP function calls to the HFP-agent. This node local agent is responsible for receiving the data from the main application and launching the designated post-processing tasks to the GPU. We write these post-processing tasks in C and use OpenACC directives to enable GPU acceleration. This methodology is used since porting the most compute intensive portion of CESM on GPUs leave the CPUs idle. In that case it will be imperative to use under-utilized compute resources on CPU and a directive based approach makes it easy to switch between different/heterogeneous target compute platforms for post-processing.

6.3.2 Experimental Results

In our experiment we compare baseline CESM with single post-processing or dual post-processing task with FP and without FP. Without using FP, we induce the scheduling delay of job scheduling data transfer to remote post-processing cluster using average time based on the availability of Rhea the post-processing cluster at ORNL. We run CESM’s post-processing analysis in-situ using our HFP framework. This analysis is specially suited for HFP framework as the
analysis involves per ranks data to be accumulated and processed and no inter-rank communication is required. Such scenarios are an ideal fit for HFP runtime to be utilized. In our Single-Node experiments, we see negligible (<1%) overhead of running the precipitation post-processing analysis on the GPU. Similarly, 320-node CESM run, we also see less than 1% overhead of precipitation post-processing analysis as shown in Figure 6.8. We argue based on HFP design, implementation and results that HFP by design is scalable as HFP-agent or HFP-threads do not communicate with each other.

6.4 Chapter Summary

In this chapter we describe three real world applications: G3NA, Real-Time Remote Processing of Laparoscopic Surgery, and CESM and their performance acceleration due to our enrichment frameworks. We also elaborate on the implementation details our proposed G3NA and its corresponding G3N-visualization tool. The visualization features of G3NA-V and its integration with HFP framework is also described. Similarly, the implementation details and preliminary experimental results for Real-Time Remote Processing of Laparoscopic Surgery and CESM applications are also detailed.
Chapter 7

Conclusion

In this chapter we summarize the work present in this dissertation along with highlighting the specific contributions made by this research.

In Chapter 1, we provide the introduction and motivation of our work, followed by background and related work in Chapter 2 and Chapter 3, respectively. In chapter 4, we provide our application / algorithm to architecture mapping. We analyze using two different methodologies a qualitative and quantitative model. The aim of the both the qualitative and the quantitative model is to allow users to port applications and algorithms across architectures and programming models. The qualitative models aims to help provide overview without an implementation thus allowing users to utilize pseudo-code to perform this analysis. The quantitative model allows for a more advanced method of analyzing and can easily be integrated into a tool and utilizes the hardware parameters of the running algorithm to predict performance on different applications. In chapter 5, we develop a framework to exploit this performance on large scale application. In chapter 6, we show some real case scenario where utilize our application to architecture mapping.

7.1 Contributions

In this research, we implemented two enrichment frameworks: Algorithm-to-Architecture (A2A) mapping and the Heterogeneous Functional Partitioning framework (HFP). We present experimental results and analysis using benchmark and real world applications to discuss the feasibility and contributions of the frameworks. The three real world applications utilized for this
research are: Large Scale Gene Network Alignment using GPGPUs (G3NA), Real-Time Remote Processing of Laparoscopic Surgery, and Climate Earth Science Model (CESM). Along with integrating the the frameworks with the applications, for G3NA we have developed an accompanying visualization tool which assists genomic research in accelerating and validating discoveries.

The research provides the following contributions:

1. A2A Qualitative: A qualitative model for identification of optimal architecture from pseudo code or serial code without run-time analysis.

2. A2A Quantitative: A quantitative model of performance estimation for porting of a given application in a given programming model and architecture to an alternate programming model and/or architecture.

3. HFP Framework: A framework for utilizing the heterogeneous environment in an efficient manner with minimal modification to the code.


### 7.2 Publications: Papers and posters


Appendices
## Appendix A  Appendix

<table>
<thead>
<tr>
<th>Platform</th>
<th>Hardware Counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA-K40</td>
<td>PAPI-SR-INS, PAPI-LD-INS, PAPI-L1-DCM, PAPI-FP-OPS, PAPI-TOT-CYC</td>
</tr>
<tr>
<td>OpenMP-16</td>
<td>PAPI-LD-INS, PAPI-SR-INS, PAPI-L1-DCM, PAPI-FP-OPS, PAPI-TOT-CYC</td>
</tr>
<tr>
<td>OpenMP-08</td>
<td>PAPI-LD-INS, PAPI-SR-INS, PAPI-L1-DCM, PAPI-FP-OPS, PAPI-TOT-CYC</td>
</tr>
<tr>
<td>OpenMP-240</td>
<td>PAPI-LD-INS, PAPI-TOT-CYC, PAPI-BR-INS, PAPI-FP-OPS, PAPI-SR-INS, PAPI-LD-INS</td>
</tr>
<tr>
<td>OpenMP-120</td>
<td>PAPI-LD-INS, PAPI-TOT-CYC, PAPI-BR-INS, PAPI-FP-OPS, PAPI-SR-INS, PAPI-LD-INS</td>
</tr>
<tr>
<td>OpenMP-272</td>
<td>PAPI-LD-INS, PAPI-TOT-CYC, PAPI-BR-INS, PAPI-FP-OPS, PAPI-SR-INS, PAPI-LD-INS</td>
</tr>
<tr>
<td>OpenMP-136</td>
<td>PAPI-LD-INS, PAPI-TOT-CYC, PAPI-BR-INS, PAPI-FP-OPS, PAPI-SR-INS, PAPI-LD-INS</td>
</tr>
<tr>
<td>CPU OpenCL</td>
<td>PAPI-LD-INS, PAPI-SR-INS, PAPI-L1-DCM, PAPI-FP-OPS, PAPI-TOT-CYC</td>
</tr>
<tr>
<td>GPU OpenCL</td>
<td>PAPI-LD-INS, PAPI-SR-INS, PAPI-L1-DCM, PAPI-FP-OPS, PAPI-TOT-CYC</td>
</tr>
<tr>
<td>Xeon Phi OpenCL</td>
<td>PAPI-LD-INS, PAPI-TOT-CYC, PAPI-BR-INS, PAPI-FP-OPS, PAPI-SR-INS, PAPI-L1-DCM</td>
</tr>
</tbody>
</table>

Table 1: Hardware counters used to train our CPU-C model.
<table>
<thead>
<tr>
<th>Platform</th>
<th>Hardware Counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP-16</td>
<td>inst-executed, inst-issued2, gst-request, gst-inst-32bit, l2-subp2-write-sector-misses, gld-request, gld-inst-32bit, inst-issued1</td>
</tr>
<tr>
<td>OpenMP-08</td>
<td>inst-executed, inst-issued2, gst-request, gst-inst-32bit, l2-subp2-write-sector-misses, gld-request, gld-inst-32bit, inst-issued1</td>
</tr>
<tr>
<td>OpenMP-240</td>
<td>inst-executed, inst-issued2, gst-request, gst-inst-32bit, l2-subp2-write-sector-misses, gld-request, gld-inst-32bit, inst-issued1</td>
</tr>
<tr>
<td>OpenMP-120</td>
<td>inst-executed, inst-issued2, gst-request, gld-request, gld-inst-32bit, inst-issued1</td>
</tr>
<tr>
<td>OpenMP-272</td>
<td>inst-executed, inst-issued2, gst-request, gld-request, gld-inst-32bit, inst-issued1</td>
</tr>
<tr>
<td>OpenMP-136</td>
<td>inst-executed, inst-issued2, gld-request, gld-inst-32bit, inst-issued1</td>
</tr>
<tr>
<td>CPU OpenCL</td>
<td>global-st-mem-divergence-replays, global-store-transaction, –l1-global-store-transaction, inst-issued1, gld-request</td>
</tr>
<tr>
<td>GPU OpenCL</td>
<td>inst-issued1, gld-request, gld-inst-32bit, fb-subp0-write-sectors, l2-subp1-total-write-sector-queries, l2-subp0-total-write-sector-queries</td>
</tr>
</tbody>
</table>

Table 2: Hardware counters used to train our CUDA-K40 model.
<table>
<thead>
<tr>
<th>Platform</th>
<th>Hardware Counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU-C</td>
<td>PAPI-FP-INS-mean, PAPI-TOT-INS-thread0, PAPI-LD-INS-thread0, PAPI-DP-OPS-mean, PAPI-SR-INS-thread0</td>
</tr>
<tr>
<td>CUDA-K40</td>
<td>PAPI-FP-OPS-total, PAPI-DP-OPS-total, PAPI-LD-INS-total, PAPI-TOT-INS-total, PAPI-REF-CYC-thread0</td>
</tr>
<tr>
<td>OpenMP-08</td>
<td>PAPI-TOT-INS-thread0, PAPI-REF-CYC-thread0, PAPI-LD-INS-thread0, PAPI-L2-DCH-thread0, PAPI-LD-INS-total</td>
</tr>
<tr>
<td>OpenMP-240</td>
<td>PAPI-TOT-CYC-thread0, PAPI-REF-CYC-thread0, PAPI-TOT-INS-thread0, PAPI-LD-INS-thread0</td>
</tr>
<tr>
<td>OpenMP-120</td>
<td>PAPI-TOT-CYC-thread0, PAPI-SR-INS-thread0, PAPI-TOT-INS-thread0, PAPI-LD-INS-thread0</td>
</tr>
<tr>
<td>OpenMP-272</td>
<td>PAPI-TOT-CYC-thread0, PAPI-SR-INS-thread0, PAPI-TOT-INS-thread0, PAPI-LD-INS-thread0</td>
</tr>
<tr>
<td>OpenMP-136</td>
<td>PAPI-TOT-CYC-thread0, PAPI-SR-INS-thread0, PAPI-TOT-INS-thread0, PAPI-LD-INS-thread0</td>
</tr>
<tr>
<td>CPU OpenCL</td>
<td>PAPI-TLB-DM-mean, PAPI-L3-TCW-thread0, PAPI-L2-STM-thread0, PAPI-L3-DCA-mean, PAPI-L2-DCM-total</td>
</tr>
<tr>
<td>GPU OpenCL</td>
<td>PAPI-FP-OPS-total, PAPI-DP-OPS-total, PAPI-LD-INS-total, PAPI-TOT-INS-mean, PAPI-REF-CYC-thread0</td>
</tr>
<tr>
<td>Xeon Phi OpenCL</td>
<td>PAPI-TLB-DM-thread0, PAPI-L3-DCA-thread0, PAPI-L2-DCM-thread0, PAPI-L2-TCM-thread0</td>
</tr>
</tbody>
</table>

Table 3: Hardware counters used to train our CPU-OpenMP model.
Table 4: Hardware counters used to train our PHI-OpenMP model.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Hardware Counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU-C</td>
<td>PAPI-VEC-INS-total, PAPI-TOT-INS-mean, PAPI-LD-INS-mean, PAPI-L1-DCA-thread0, PAPI-SR-INS-thread0</td>
</tr>
<tr>
<td>CUDA-K40</td>
<td>PAPI-VEC-INS-total, PAPI-TOT-INS-total, PAPI-LD-INS-total, PAPI-L1-DCA-mean, PAPI-L1-ICA-mean, PAPI-SR-INS-thread0, PAPI-VEC-INS-mean</td>
</tr>
<tr>
<td>OpenMP-16</td>
<td>PAPI-TOT-CYC-thread0, PAPI-LD-INS-mean, PAPI-L1-ICA-thread0, PAPI-L1-ICA-total, PAPI-SR-INS-thread0, PAPI-VEC-INS-mean</td>
</tr>
<tr>
<td>OpenMP-08</td>
<td>PAPI-TOT-CYC-thread0, PAPI-LD-INS-mean, PAPI-L1-ICA-thread0, PAPI-L1-ICA-total, PAPI-SR-INS-thread0, PAPI-VEC-INS-mean</td>
</tr>
<tr>
<td>OpenMP-120</td>
<td>PAPI-TOT-CYC-thread0, PAPI-L1-DCM-thread0, PAPI-L1-ICA-thread0, PAPI-L2-LDM-mean, PAPI-LD-INS-thread0</td>
</tr>
<tr>
<td>OpenMP-272</td>
<td>PAPI-TOT-CYC-thread0, PAPI-LD-INS-mean, PAPI-L1-ICA-thread0, PAPI-L1-ICA-total, PAPI-SR-INS-thread0, PAPI-VEC-INS-mean</td>
</tr>
<tr>
<td>OpenMP-136</td>
<td>PAPI-TOT-CYC-thread0, PAPI-LD-INS-mean, PAPI-L1-ICA-thread0, PAPI-L1-ICA-total, PAPI-SR-INS-thread0, PAPI-VEC-INS-mean</td>
</tr>
<tr>
<td>CPU OpenCL</td>
<td>PAPI-TLB-DM-thread0, PAPI-L2-LDM-thread0, PAPI-L1-DCM-thread0, PAPI-TOT-CYC-thread0, PAPI-L1-ICA-thread0</td>
</tr>
<tr>
<td>GPU OpenCL</td>
<td>PAPI-VEC-INS-total, PAPI-TOT-CYC-thread0, PAPI-LD-INS-total, PAPI-L1-DCA-mean, PAPI-SR-INS-thread0</td>
</tr>
<tr>
<td>Xeon Phi OpenCL</td>
<td>PAPI-TLB-DM-thread0, PAPI-L1-DCM-thread0, PAPI-TOT-CYC-thread0, PAPI-L1-ICA-thread0</td>
</tr>
</tbody>
</table>

Table 5: Applications and their corresponding implementations for Parboil Benchmark.

<table>
<thead>
<tr>
<th>Application</th>
<th>C</th>
<th>CUDA</th>
<th>OpenMP</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUTCP</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MRI-Q</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MRI-GRID</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAD</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STENCIL</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TPACF</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LBM</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGEMM</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPMV</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HISTOGRAM</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application</td>
<td>C</td>
<td>CUDA</td>
<td>OpenMP</td>
<td>OpenCL</td>
</tr>
<tr>
<td>-------------------</td>
<td>---</td>
<td>------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>2MM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>3MM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ADI</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ATAX</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>BICG</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CHOLLESKY</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CORRELATION</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>COVARIANCE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DOITGEN</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FDTD-2D</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>GEMM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>GEMVER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>GESUMMV</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>GRAMSCHMIDT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>JACOBI-1D</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>JACOBI-2D</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>LU</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MVT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SYMM</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SYR2K</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SYRK</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TRISOLV</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Applications and their corresponding implementations for Polybench Benchmark.
Bibliography


