12-2016

A performance focused, development friendly and model aided parallelization strategy for scientific applications

Anagha S. Joshi
Clemson University

A performance focused, development friendly and model aided parallelization strategy for scientific applications

---

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

---

by
Anagha S. Joshi
December 2016

Accepted by:
Dr. Melissa C. Smith, Committee Chair
Dr. Harlan B. Russell
Dr. Richard R. Brooks

Abstract

The amelioration of high performance computing platforms has provided unprecedented computing power with the evolution of multi-core CPUs, massively parallel architectures such as General Purpose Graphics Processing Units (GPGPUs) and Many Integrated Core (MIC) architectures such as Intel's Xeon phi coprocessor. However, it is a great challenge to leverage capabilities of such advanced supercomputing hardware, as it requires efficient and effective parallelization of scientific applications. This task is difficult mainly due to complexity of scientific algorithms coupled with the variety of available hardware and disparate programming models.

To address the aforementioned challenges, this thesis presents a parallelization strategy to accelerate scientific applications that maximizes the opportunities of achieving speedup while minimizing the development efforts. Parallelization is a three step process (1) choose a compatible combination of architecture and parallel programming language, (2) translate base code/algorithm to a parallel language and (3) optimize and tune the application. In this research, a quantitative comparison of run time for various implementations of k-means algorithm, is used to establish that native languages (OpenMP, MPI, CUDA) perform better on respective architectures as opposed to vendor-neutral languages such as OpenCL. A qualitative model is used to select an optimal architecture for a given application by aligning the capabilities of accelerators with characteristics of the application. Once the optimal architecture is chosen, the corresponding native language is employed. This approach provides the best performance with reasonable accuracy (78%) of predicting a fitting combination, while eliminating the need for exploring different architectures individ-

ually. It reduces the required development efforts considerably as the application need not be re-written in multiple languages. The focus can be solely on optimization and tuning to achieve the best performance on available architectures with minimized investment in terms of cost and efforts.

To verify the prediction accuracy of the qualitative model, the OpenDwarfs [1] benchmark suite, which implements the Berkeley's dwarfs in OpenCL, is used. A dwarf is an algorithmic method that captures a pattern of computation and communication [2]. For the purpose of this research, the focus is on 9 application from various algorithmic domains that cover the seven dwarfs of symbolic computation, which were identified by Phillip Colella [3], as omnipresent in scientific and engineering applications. To validate the parallelization strategy collectively, a case study is undertaken. This case study involves parallelization of the Lower Upper Decomposition for the Gaussian Elimination algorithm from the linear algebra domain, using conventional trial and error methods as well as the proposed "Architecture First, Language Later" strategy. The development efforts incurred are contrasted for both methods. The aforesaid proposed strategy is observed to reduce the development efforts by an average of 50%.

Dedication

I dedicate this thesis to my family for their constant support and encouragement.

Acknowledgements

I would like to acknowledge the invaluable guidance and support of my advisor Dr. Melissa C. Smith (who deserves all of the credit and none of the blame!). I thank my committee members Dr. Harlan B. Russell and Dr. Richard Brooks for their review and valuable comments on this thesis. I would also like to acknowledge the members of Future Computing Technology Lab of Clemson University for their support and cooperation. My gratitude towards my family, without them this thesis would not have been possible. Finally, I would like to thank Clemson University for the excellent academic environment throughout the program.

Table of Contents

Page

List of Tables

Table **Page**

List of Figures

Figure **Page**

Chapter 1

Introduction

1.1 Motivation

As the parallel computing hardware is evolving, high performance computers are assuming an increasingly central role in scientific research. High performance computing has allowed scientist to reduce time to solution, enrich complexity of models, and enhance realism of simulations. However, exploitation of these supercomputing capabilities is a challenge due to complexity of the scientific applications, lack of inherent parallelism and need to be well aligned with the latest available hardware such as multi-core processors, GPGPUs and Many Integrated Core architecture co-processors each having disparate programming models. These new systems, although all parallel, have varying capabilities making one more suitable for certain application than its counterparts. However, no straightforward solution exists to answer the question - which of the available architecture would yield best performance?

The state of practice approach in academia and industry is either trial and error methods or selection based on parameters other than fitness. Trial and error methods are costly in terms of time, resources and efforts, which is often limiting. Selection based on factors such as cost, availability and/or programmability often hampers the achievable performance for the given application. Thus, it is of great importance to define a strategy

for parallelization that would map the application to an architecture without any trial and error methods. Architecture specific tools can then be used to translate and optimize the given application to achieve best performance with the least amount of development efforts. Having a qualitative model that predicts a suitable architecture for a given application with reasonable accuracy is also valuable to assist a job scheduler in a heterogeneous cluster settings. Mapping applications to matching architectures can yield considerable improvement in performance and resource management. However, a complete solution for such a setting would call for a more complex algorithm along with the predictive model for an efficient and effective scheduling. It is out of scope for this research.

1.2 Our Work

This thesis presents a 3 step "Architecture First, Language Later" parallelization strategy: (1) choose a compatible combination of architecture and parallel programming language, (2) translate base code/algorithm to a parallel language and (3) optimize and tune the application.

The K-means algorithm from the Linear Algebra domain is used as a an example to compare the run times on different architectures in different languages for a fixed input size and parameters. This example establishes the extent of performance variability between each combination and suggests that a native language achieves best performance for any given architecture. A qualitative model proposed in [7], uses four common parameters to map applications to the architectures. The work in this thesis verifies the prediction accuracy of said model by using 9 applications from the OpenDwarfs Benchmark. The verified model predicts the optimal architecture for the given application. Thus, the architecture predicted by the qualitative model along with corresponding native language yields the optimal combination mentioned in step 1. Translation of code to a native language as the 2nd step, involves usage of language specific standards and documentation. The details of the optimization techniques suggested as step 3 are out of scope for this work.

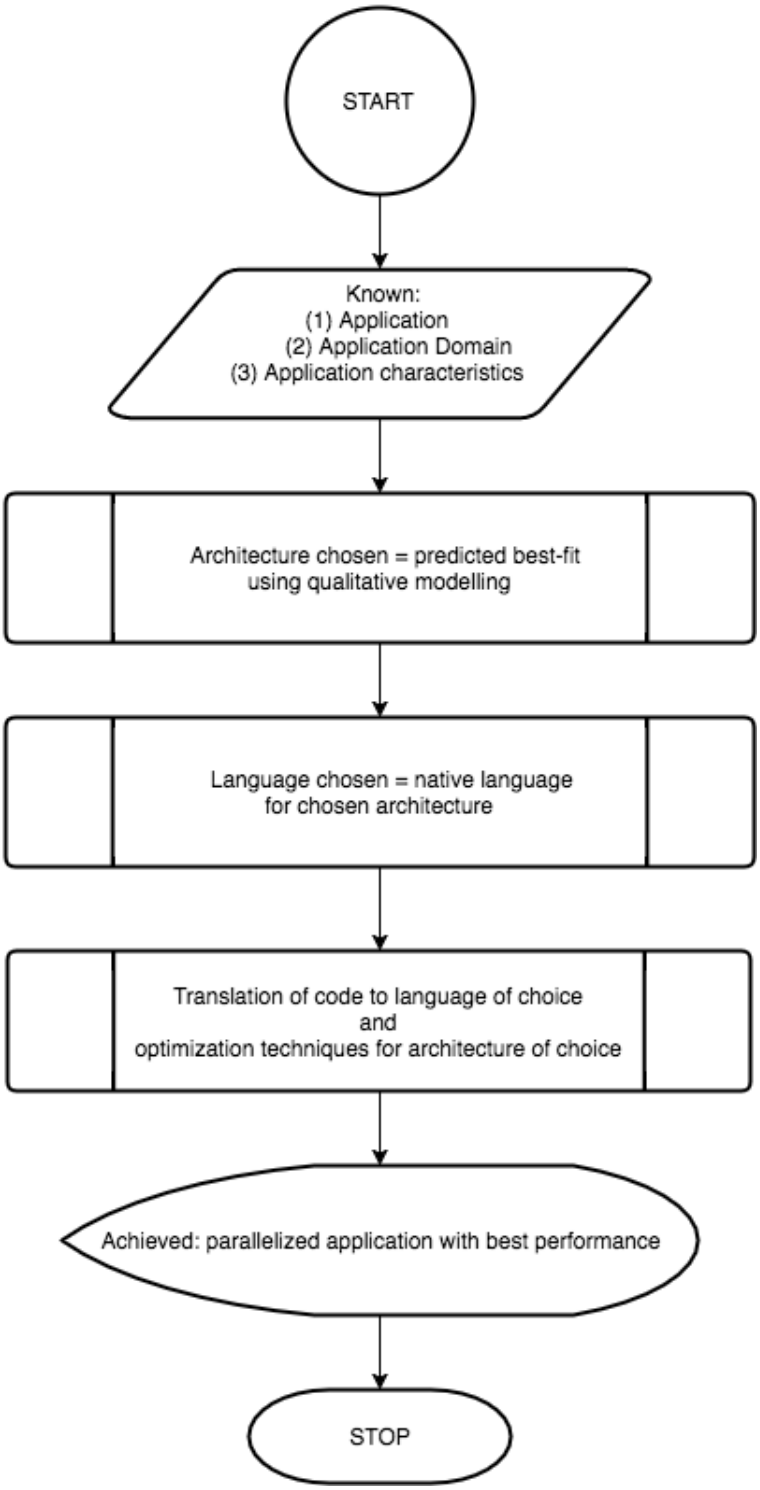

Figure 1.1: Flowchart for Strategic Process of Parallelization.

Figure 1.1 depicts the steps for strategic parallelization using "Architecture First, Language Later" strategy. The effectiveness of the aforesaid parallelization strategy is investigated by a parallelization experiment for the Lower Upper Decomposition algorithm from the Linear Algebra domain. This case study employs trial and error methods as well as the proposed "Architecture First, Language Later" parallelization strategy.

The trial and error methods involve development of code in MPI, CUDA and MPI+CUDA hybrid implementation. In comparison, parallelization using "Architecture First, Language Later" strategy is used to approach the same problem, which provides final results with code conversion for only the selected parallel programming model, as opposed to multiple models in trial and error method. The two methods are evaluated based on achieved results; i.e. run time of implementations, parallelization efforts in terms of man-hours for code conversion, amount of necessary accesses to hardware and degree of knowledge and expertise required for the method. The overall results and efforts incurred are contrasted for both methods.

1.3 Thesis Outline

Chapter 2 reviews the relevant literature. Chapter 3 provides details of popular parallel programming platforms and their corresponding programming models along with the background information about benchmarks and algorithms used in this research study. This chapter also discusses the challenges associated with portability among these architectures. Chapter 4 explores experimental design and details the claims regarding the effectiveness of the "Architecture First, Language Later" parallelization strategy. This chapter additionally includes details of the methodology and measures defined to investigate these claims. Chapter 5 describes the case study with contrasting approaches for parallelization. Chapter 6 presents the results for individual steps of the aforementioned parallelization strategy along with observations from the case study. The thesis concludes in Chapter 7 with conclusions and suggestions for future work.

Chapter 2

Related Work

The purpose of this chapter is to review some of the prominent literature related to the performance-centric parallelization approaches and aspects of portability. Section 2.1 reviews results from various studies summarizing the performance modelling approaches and factors affecting performance of parallel applications. Section 2.2 examines the architecture studies conducted for multi-core CPUs, General Purpose GPUs (GPGPU) and Many Integrated Core (MIC) architectures such as the Xeon Phi. Section 2.3 highlights the diversity among parallel programming models. The chapter is concluded with a summary in Section 2.4.

2.1 Review Of Performance Modelling Approaches

In the field of accelerator-based computing, there is an absence of a reliable reference model for performance prediction and tuning. In [8], authors survey performance models for parallel applications running on heterogeneous platforms, including the most popular families of accelerators: General Purpose GPUs (GPGPUs) and Intel's Xeon Phi. The paper identifies three main approaches to modeling the performance of parallel systems: analytical modeling, machine learning and simulation. It is concluded that no accurate model exists that is valid for a wide set of architectures. There is a trade off between

accuracy and generality. Even though device-specific models are found to be more accurate, they can become obsolete very quickly due to the fast pace at which manufacturers market products with additional and improved features. Learning from these observations, this research proposes a more generic yet reasonably accurate analytical hybrid approach for performance modelling.

In [9], the author makes a significant effort toward developing a multi-suite performance prediction model for heterogeneous systems. The most note-worthy is the definition, reasoning and validation of a hybrid approach that combines quantitative and qualitative modelling methods. By evaluating both methods for modelling GPGPU's computation and communication patterns, the author asserts that quantitative methods are more suitable for less complex systems, which can be described using a small set of measurable parameters, while qualitative methods are better suited for complex systems with reproducible characteristics. Based on these assertions, the hybrid approach combined the subjective-analytical model for GPGPU computations and objective-analytical models for communications to perform adequately fruitful performance predictions. With prediction error rates less than 5%, this approach is validated for precise predictions.

Bhuiyan [10] introduced a Fitness performance model to predict the theoretical best accelerator, or group of accelerators, for an application. The fitness model uses total number of flops, device-to-device transfer, and host-to-device transfer sizes for application characterization. In addition, characteristics of accelerators, such as time required for the execution of floating point operations (SFLOP), device to device transfer (SDBT), and host to device transfer (SHDT) are utilized. Theoretical run-time calculations, based on the aforementioned size and time quantities, are then used to predict the ranks of the accelerators for the given application. The model requires highly quantitative data, which is difficult to obtain especially for new applications. Additionally, the dependency of the model on the quantitative hardware characteristics makes the model highly susceptible to variations in versions/models in the same architecture family. Thus, the qualitative model used as the framework in this research aims to extend the basic idea of fitness model while

making it more generic and robust.

2.2 Parallel Architecture Studies

High Performance Computing (HPC) is now a multi/many-core world. The HPC community seems to be divided when it comes to selecting either multi-core CPU, GPGPU or MIC as the best performing architecture. Depending on the application characteristics, the unique characteristics of the architecture results in varying degrees of achieved performance. Thus there is no scale to rank the given architectures unless it is application dependent.

Some literature seems to favour the relatively new MIC architecture introduced by Intel, i.e. Intel Xeon Phi. In [11], authors have evaluated the performance of sparse matrix multiplication kernels on the Intel Xeon Phi. Their results show that Xeon Phi's sparse kernel performance is very promising and even better than that of cutting-edge general purpose processors and GPUs.They have attributed this success to Xeon Phi's 60 cores, wide registers and vectorization capabilities. Additionally the experiments suggested that having a relatively small 512kB L2 cache per core is not a problem for Intel Xeon Phi. However it is for a particular class of applications and thus this may not hold true for other application domains.

In contrast to the paper mentioned above, Abdullah, kok Yong, Karuppiah and Chong [12] have demonstrated that the GPU outperforms MIC for a Multi-Keyword Text Search algorithm. For their comparison study, they have implemented different optimizations on both accelerators to fully utilize their architectural advantages such as vectorization and branch divergence method on the MIC, and coalesced memory and shared memory accesses for the GPU. They have used use NVIDIA K20c and NVIDIA K40 for GPUs and Intel® Xeon Phi™ 5100 for the MIC.

While accelerators have been fairly popular, in some cases multi-core CPUs tend to be a more apt choice. One such instance is the paper titled "Believe It or Not! Multi-

core CPUs can Match GPU Performance for a FLOP-Intensive Application!"[13]. The authors evaluated performance of a real-world image processing application that uses a cross-correlation algorithm to compare a given image with a reference one. They suggest that in this particular scenario, the GPU suffered due to (1) a smaller shared memory, (2) unaligned device memory access patterns, (3) expensive atomic operations, and (4) weaker single-thread performance.

There are a large number of publications targeted to a specific architecture or application domain. However, a comprehensive literature review covering a variety of popular architecture families for a range of scientific applications is sparse. One prominent paper [14], compares performance of CPUs, GPUs, and MICs for operations with different data access patterns (regular and irregular), computation intensity, and types of parallelism, from a class of applications with digital pathology as the primary application. The authors have demonstrated that different types of co-processors are more appropriate for specific data access patterns and types of parallelism through the experimental results.

2.3 Parallel Programming Models

The coding of a parallel application is generally strongly influenced by the choice of parallel hardware, even though there exists platform independent solutions. The most popular schemes employ either MPI[15], OpenMP[16], CUDA[17], OpenCL [18] or hybrid approaches such as MPI + X where X is OpenMP or CUDA or OpenACC[19]. OpenCL (Open Computing Language) is the open, royalty-free standard for cross-platform, parallel programming and thus is supported by most architectures.

In [20] the authors compare CUDA and OpenCL implementations of a Monte Carlo Chemistry application [20]. The paper showed OpenCL providing code portability but not necessarily providing performance. Furthermore, they showed that platform-specific languages often, but not always, outperformed OpenCL. Other studies [21], [22] also suggest that OpenCL provides portability at the cost of partial performance degradation.

2.4 Summary Of Literature Review

In this chapter, we discussed some of the influential literature targeting performance modeling approaches. Some key take-aways were the trade offs between accuracy and generality, the idea of a fitness matching model and hybrid approach technique that combines qualitative and quantitative models into one. Further, we explained that building on these lessons this research proposes a more generic model with reasonable prediction accuracy aided by a hybrid approach of qualitative modeling for fitness and quantitative modeling for selecting programming methodology. Additionally, we discussed architecture studies that demonstrate that a specific type of processor or co-processor/accelerator is more appropriate for certain type of applications with unique set of computation and communication patterns. The next chapter provides a detailed background information relevant to this thesis.

Chapter 3

Background

This chapter will introduce parallel programming platforms such as multicore and manycore processors and cover the details of computational accelerators in particular Nvidia K20 GPU and Intel Xeon Phi. Some of the popular programming models such as the Compute Unified Device Architecture (CUDA) by Nvidia, Open Computing Language (OpenCL) and Message Passing Interface (MPI) are detailed in the section 3.3. This chapter also discusses the cost of code portability.

3.1 Parallel Programming Platforms

In [23], van der Steen and Dongarra have provided a regularly updated complete performance overview of recent supercomputers. Although this thesis can not cover the extensive overview of current parallel computer technology, key architecture concepts are discussed in the following subsections 3.1.1, 3.1.2 and 3.1.3.

3.1.1 Flynn's Taxonomy

A widely used taxonomy for describing concurrency and data streams in a given parallel architecture was presented by Flynn [24]. Flynn's taxonomy categorizes computers depending on the number of instruction streams and data streams, where 'stream' is a

sequence of instruction or data on which the computer performs operations. Out of the four possibilities as seen in Figure 3.1 [25], Single Instruction multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) variants are more dominating concepts today.[26]

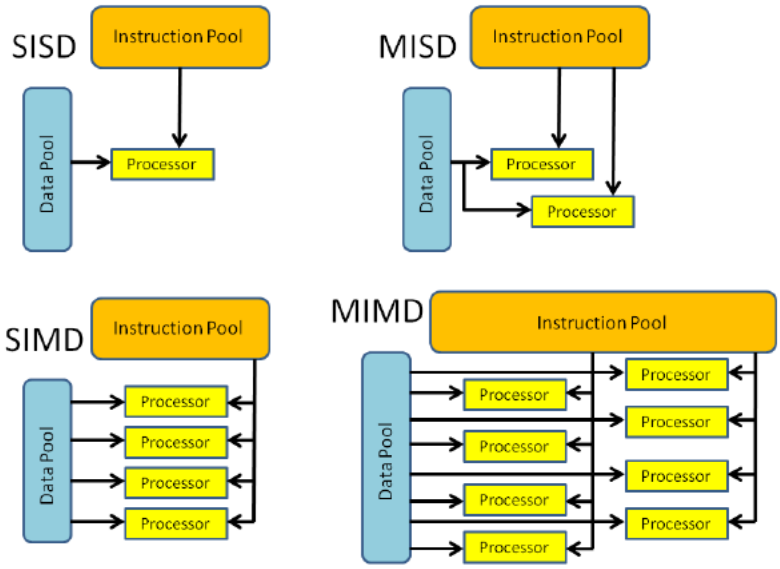

Figure 3.1: Multicore vs Manycore

Single Instruction Multiple Data **SIMD**: A single instruction stream concurrently operates on multiple data streams to achieve parallel processing on whether single core or multiple cores. Examples include vector processors, modern superscalar microprocessors, and Graphics Processing Units (GPUs).

Multiple Instruction Multiple Data**MIMD**: Multiple instruction streams on multiple processors process different data items in parallel resulting in parallelized operations.

Single Instruction Single Data (**SISD**): A single instruction stream operating on a single data stream is the serial form of computation with no concurrency in operation.

Multiple Instruction Single Data (**MISD**): This format is not regarded as a useful paradigm in practice.

3.1.2 Shared Memory vs Distributed Memory

Shared Memory: In shared memory organization, as the name suggests, all processors share the memory modelled as a global address space. All processors can view the changes made to a memory location by any other processor. Shared memory models are further classified into Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA), based upon the time required to access the shared memory.[27]

Uniform Memory Access (UMA): As depicted by Figure 3.2[27], identical processors are symmetrically, i.e. symmetrical multi-processors (SMP), connected to shared memory with equal access rights and access times. Any update to a memory location by one of the processors will be immediately available to all other processors. Thus, this configuration is sometimes referred to as Cache Coherent UMA (CC-UMA).

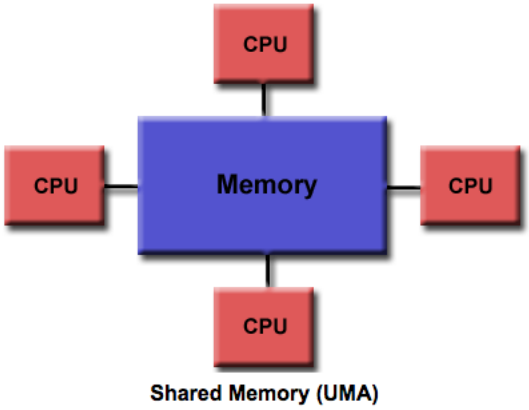

Figure 3.2: Shared Memory - Uniform Memory Access (UMA)

Non-Uniform Memory Access (NUMA): This is a configuration where two or more SMPs are physically linked as seen in Figure 3.3[27]. The access time varies depending on the location of the memory and the processor accessing it and hence the name non-uniform.

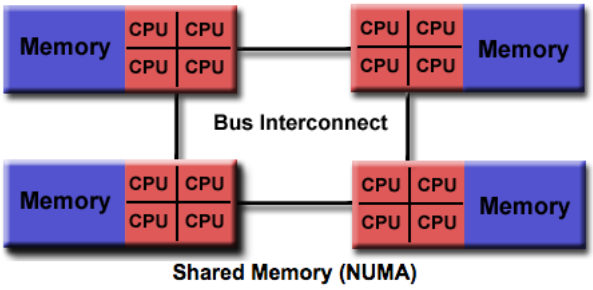

Figure 3.3: Shared Memory - Non-Uniform Memory Access (NUMA)

The advantage of shared memory organization is that the global address space simplifies the programmer's view of memory, however, scaling is a challenging task as more CPUs are added.

Distributed Memory: Distributed memory systems do not have a global memory space. As observed in Figure 3.4 [27] utilizes a communication network to connect processors. Each processor has a private local memory. When a processor needs to access data from another processor, it is the programmer's responsibility to explicitly define data communication via creating tasks and maintaining synchronization between tasks. The obvious advantage of distributed memory organization is scalability as adding new processors to the system is relatively easy. However, the programmer must bare the burden of memory management.

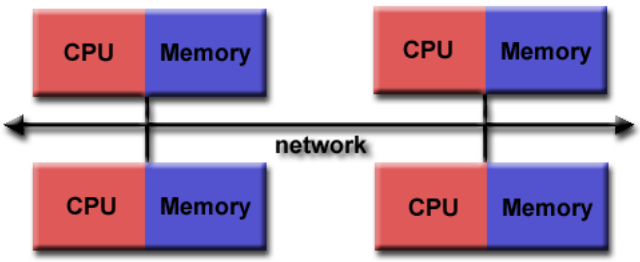

Figure 3.4: Distributed Memory

3.1.3 Multicore vs Manycore

Parallel computers are computers with multiple processing units. These multiple processing units can be in the form of multicore or manycore. A famous quote by Seymour Cray states, "If you were plowing a field, which would you rather use: two strong oxen or 1,024 chickens?". Two strong oxen refers to multicore processors which are designed for latency and 1024 chickens refers to manycore processors that are designed for throughput as depicted in Figure 3.5.

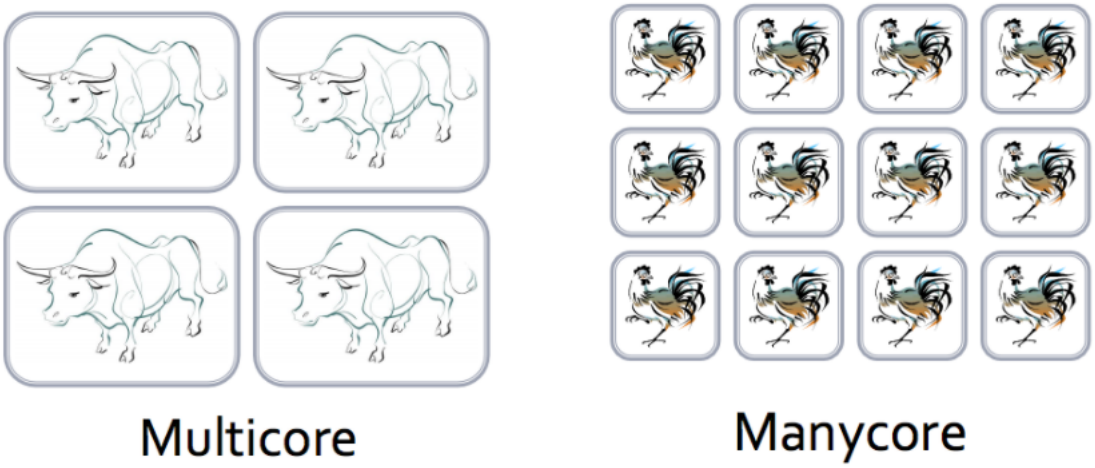

Figure 3.5: Multicore vs Manycore

3.2 Computational Accelerators

In High performance computing, the community is not satisfied with currently available compute power and thus there are a lot of initiatives other than using multiple processors for better performance. A class of such advances is hardware accelerators. An accelerator is a stand-alone device that is added to a general-purpose computer to help improve performance. General Purpose Graphics Processing Units (GPGPUs), Field Programmable Gate Arrays(FPGAs), Many Integrated Cores cards, etc. are examples of computational accelerators.

Nvidia K20 GPU Graphics processing units (GPU) are intended to render graph-

ics. GPUs can also be used in conjunction with CPU to perform general purpose computation which is traditionally handled by the central processing unit (CPU). Graphics processing units (GPU) together with CPUs when used to achieve accelerated computation via parallel programming, GPU is referred to as General Purpose Graphics Processing Unit (GP-GPU). Significant performance gain is achieved by offloading compute-intensive operations to the GPU, while the remaining code still runs on the CPU as depicted in Figure 3.6.

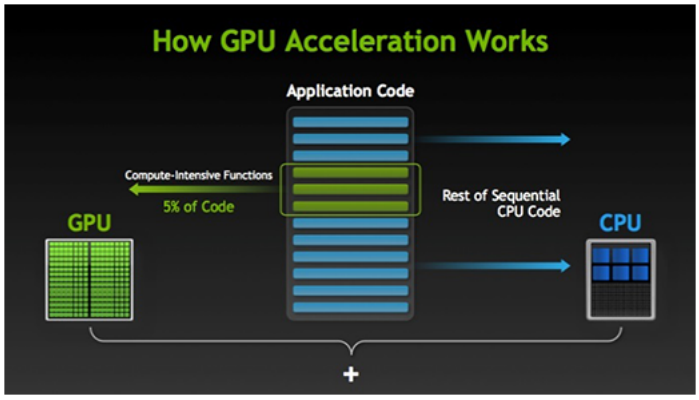

Figure 3.6: How GPU Acceleration Works [4]

NVIDIA is a leading company in the GPU field. A simplified block diagram for Tesla Kepler K20 GPU model is shown in Figure 3.7. There are 15 Streaming Multiprocessors (SMXs) with 192 CUDA cores each. A diagram of an SMX with some internal detail is given in figure 3.8 [23].

The hardware organization of a GPU can be explained as follows: 1 to 3 dimensional thread blocks constitute a grid. Each individual thread block consists of 1-3 dimensional group of threads. The CUDA work distributor assigns thread blocks to SMX units. The SMX unit breaks thread blocks into warps. A warp is a groups of 32 threads. Each SMX unit has warp schedulers that selects an eligible warp. Each thread has its own set of general purpose registers and local memory. All threads can access the thread block resources which include shared memory and barriers. All threads in a grid can access grid resources including constant memory, texture bindings, and surface bindings. All threads can access

Figure 3.7: Simplified block diagram of the NVIDIA Tesla K20 GPU [5]

global memory. Parallelism is achieved by multiple CUDA cores executing multiple threads concurrently. The massively parallel hardware architecture of General Purpose Graphics Processing Units (GP-GPUs) make them well-suited to scientific computing. GP-GPUs do especially well with floating point arithmetic.

Intel Xeon Phi Intel's first many core product is called the Xeon Phi. The Xeon Phi accelerator has its own memory and is connected to the host CPU via PCI3 Gen3 16x [23].

Figure 3.9 shows a basic block diagram of the Intel Xeon Phi while Figure 3.10 lays out the diagram of a Phi processor core. Intel Xeon Phi has more than 50 cores with each core capable of four hardware threads. Cores are interconnected by a high-speed bidirectional ring. The co-processor is cache coherent. Each core has a local 512-KB L2 cache. [6].

Figure 3.8: Diagram of a Streaming Processor of the Tesla K20 [5]

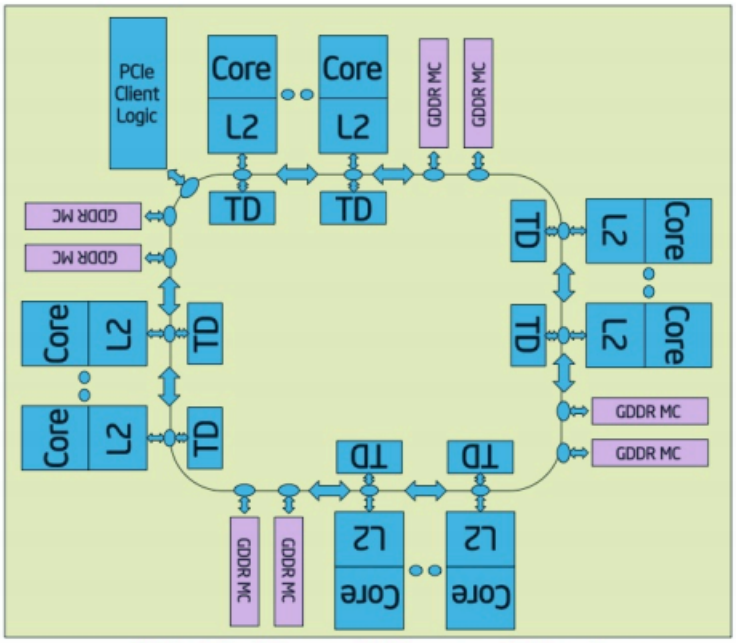

Figure 3.9: Diagram of Intel Xeon Phi Coprocessor [6]

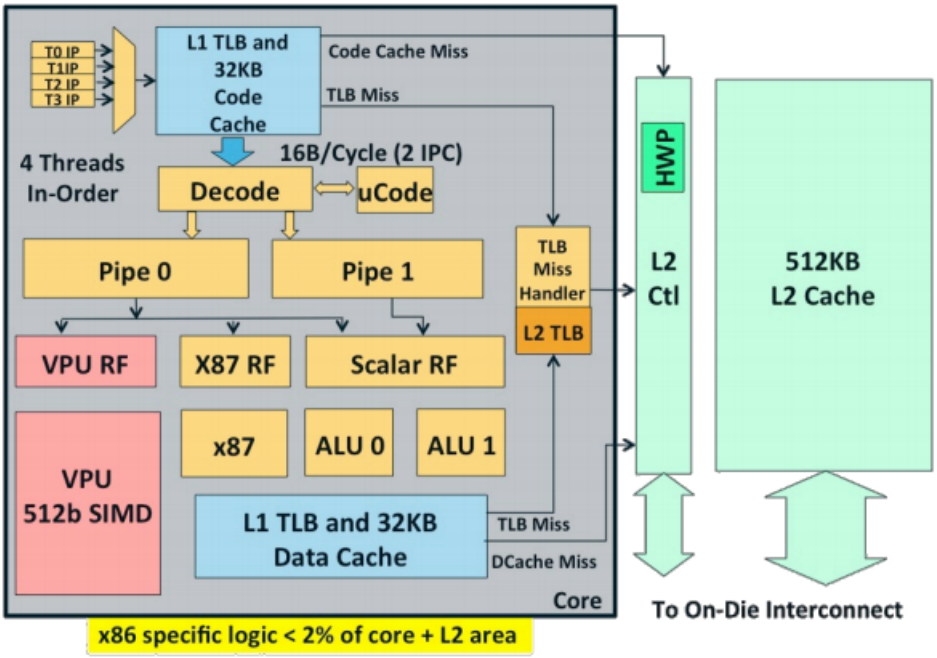

Figure 3.10: Diagram of a Phi processor core [6]

3.3 Parallel Programming Models

A parallel programming model is an abstraction of the computer system. As there are many parallel architectures, there are many possible models for parallel computing. Parallelization can be thought of as a process to speed up serial program by arranging multiple computations to happen simultaneously. Unfortunately, there is no universal process to follow.

3.3.1 Message Passing Interface (MPI)

MPI is a standard/specification message passing library implemented by many vendors and is used for programming systems with distributed memory. Each process has a different address space and thus processes must explicitly communicate with each other. MPI can also support distributed program execution on heterogeneous hardware. MPI specifications have been defined for C, C++ and Fortran.

MPI follows a task channel model. The hardware is viewed as a set of processors, each with its own local memory. Message passing between processors is facilitated by an interconnect network. Since a processor can access data and instruction only in it's local memory, a processor must send a message to another processor to access the data in the other processor's memory indirectly. The number of concurrent processes are explicitly specified by the programmer at the beginning of the program. Every process has a unique ID number and executes the same program. A process performs computations on its local variables and communicates with other processes as and when required. Processes use message passing to communicate as well as to synchronize with each other.

3.3.2 Compute Unified Device Architecture (CUDA)

The CUDA programming model comprises of Host and Device hardware and code. The Host is normally a traditional CPU dealing with complicated control flow where as the Device is typically a massively parallel processor (e.g. GPU) equipped with a large number

of arithmetic execution units for data-level parallelism. A CUDA program consists of one or more phases that are executed on either the host or a device. CUDA allows us to program both CPUs and GPUs using one single program.

Part of the program is in traditional C and will run on Host/CPU and the other part is also coded in C but with some extensions that CUDA provides to express parallelism, which will run on the Device/GPU. The CUDA compiler (nvcc) is used to compile the program, which splits the two parts accordingly and generates an executable for each CPU and GPU.

CUDA assumes that the GPU is the co-processor to the CPU and it also assumes that they both have their own separate memories where they store data. In this relationship, the CPU is the in charge. It runs the main program and sends directions to the device. So on a high level, the general outline of a CUDA code is as follows:

1. Allocating memory on GPU (CudaMalloc)
2. Moving Data from CPU to GPU (cudaMemcpyHostToDevice)
3. Launching the kernel on GPU
4. Moving the data back from GPU to CPU (cudaMemcpyDeviceToHost)

A CUDA kernel is launched as array of threads. Multiple threads execute the same code on different data. The mapping of threads to CUDA cores is governed by GPU concepts such as thread blocks, grid, warp and GPU hardware such as warp schedulers, CUDA cores (streaming processors) and Streaming Multiprocessors (SMXs) as explained in section 3.2.

3.3.3 Open Computing Language (OpenCL)

OpenCL is a framework for heterogeneous computing that includes a language OpenCL C, an API and libraries. The OpenCL standard is maintained by the Khronos Group consortium.

The platform model for OpenCL is similar to CUDA and views the underlying hardware as a host connected to one or more devices. An OpenCL program consists of a host program that runs on the host and kernels that execute on the device. Devices can

be a GPU, a multi-core CPU or a many core device. OpenCL thus provides portability, i.e. code written in OpenCL can be executed many kinds of devices irrespective of the vendor and/or hardware architecture model. Any given device is assumed to have multiple computing units. Each computing unit has processing elements and it's own memory. A computing unit executes a work-group, which is a set of work items. The work items in a group share the memory of the compute device and use work-group barriers as means of synchronization. Each work item is an instance of the kernel. The kernel executes over a grid defined by the host code. Thus parallelism is achieved by multiple work items executed concurrently by multiple processing elements of the compute units.

3.4 Portability Discussion

Almost all of the parallel programming models are too closely aligned to a particular parallel system, except OpenCL. Thus parallel programs are often non-portable across various architectures due to the tight coupling between the programming model and the hardware architecture. The non-portability of HPC applications results in higher cost, effort and time for development. This sometimes discourages HPC users from migrating to newer architectures and thus restricts them from possible performance gains and benefits.

3.4.1 Benefits Of Portability

Portable code aids the application scientists to focus on algorithmic aspects of the code as opposed to investing time in the implementation aspects. Portability of the application allows them to experiment easily with different architectures as it needs minimal to no changes in the code. In heterogeneous cluster environments, portability allows dynamic allocations of the code depending on resource availability, which is advantageous as a large portion of the community often shares the high performance computing clusters.

3.4.2 Costs Of Portability

Portability usually necessitates some degree of sacrifice in terms of performance. Portability is achieved by abstracting some hardware details in order to perceive them as similar and interchangeable. This commonality takes away some of the details which when utilized could reduce the run time, i.e. improve the performance. In high performance computing, performance is obviously the number one priority, which makes the choice between better performing non-portable code vs. slightly less performing portable code even more difficult.

3.5 Summary Of Background

This chapter provided details of a few key concepts in parallel programming such as Flynn's Taxonomy, Shared Vs. Distributed Memory and Manycore Vs. Multicore processors. The hardware design details of the most prominent computing accelerators such as General Purpose Graphics Processing Unit (GP-GPU) and Many Integrated Core (MIC) architecture were also elaborated in this chapter. Additionally, various programming models such as MPI, CUDA, OpenCL were detailed. This chapter demonstrated the great degree of variability that exists in parallel architectures and parallel programming paradigms. Benefits and costs of Code Portability were also analyzed in this chapter. The next chapter discusses the design of the research experimentation and details the metrics for accessing the results.

Chapter 4

Experimental Design

The purpose of this chapter is to explore experimental design of the research and detail the claims regarding the effectiveness of the "Architecture First, Language Later" parallelization strategy. Section 4.1 examines the research question and proposed hypothesis. Section 4.2 details the methodology for application to architecture mapping and programming model selection. Section 4.3 highlights the measures used to quantify the outcome. Section 4.4 reviews the hardware details for the setup for the experiment.

4.1 Hypothesis and Research Question

Research Question: Is there an effective way to parallelize scientific applications so as to achieve best performance for comparatively minimal development efforts?

Hypothesis: If "Architecture First, Language Later" parallelization strategy is adopted, the parallelized application would achieve best performance with minimal development efforts. Since the aforementioned strategy predicts the best architecture by mapping architecture's capabilities to the needs of the application, best performance is ensured. Development efforts are reduced significantly since it eliminates the need for trial and error methods to obtain satisfactory performance as the architecture and it's native language is chosen by application characterization.

4.2 Methodology

Figure 4.1 below depicts the entire mapping between application to architecture and architecture to programming model for the "Architecture First, Language Later" parallelization strategy.

4.2.1 Application to Architecture Mapping

As shown in Figure 4.1, the first step of application to architecture mapping is *application characterization*. This involves ranking the application as high or low for four parameters namely, Time complexity, FLOPs to Non-FLOPs Ratio, Dominant Memory Access Ratio and Memory Access Frequency.The model then predicts an optimal architecture based on these parameters.

For a given application, depending on *high*/*low* ratings on the aforementioned parameters, navigating the image following corresponding arrows yields a specific architecture as an optimal architecture prediction. For example, for a given application xyz, Time Complexity is *high* while all the other three parameters are *low*. Then navigating to the right along with *high* arrow for Time Complexity, followed by moving forward along the direction of *low* arrow for FLOPs to Non-FLOPs Ratio leads to a two dimensional square with two options. Moving horizontally along the direction of *low* arrow for Dominant Memory Access Ratio and travelling vertically in the direction of *low* arrow for Memory Access Frequency, yields 'Xeon Phi' as the architecture which is predicted to achieve best performance for the given xyz application as depicted in Figure 4.2.

Table 4.1 summarizes different factors that impact performance of any given application. As observed , characteristics are combined into a single parameter that accounts for the effects of similar characteristics. Below is a description of how the parameter is defined, ranked and linked to capabilities of the architecture. Multicore Intel CPU, Many Integrated Core (MIC) Xeon Phi and General Purpose Nvidia GPU are the architectures considered for the model. Details of each of these parameters are provided below.

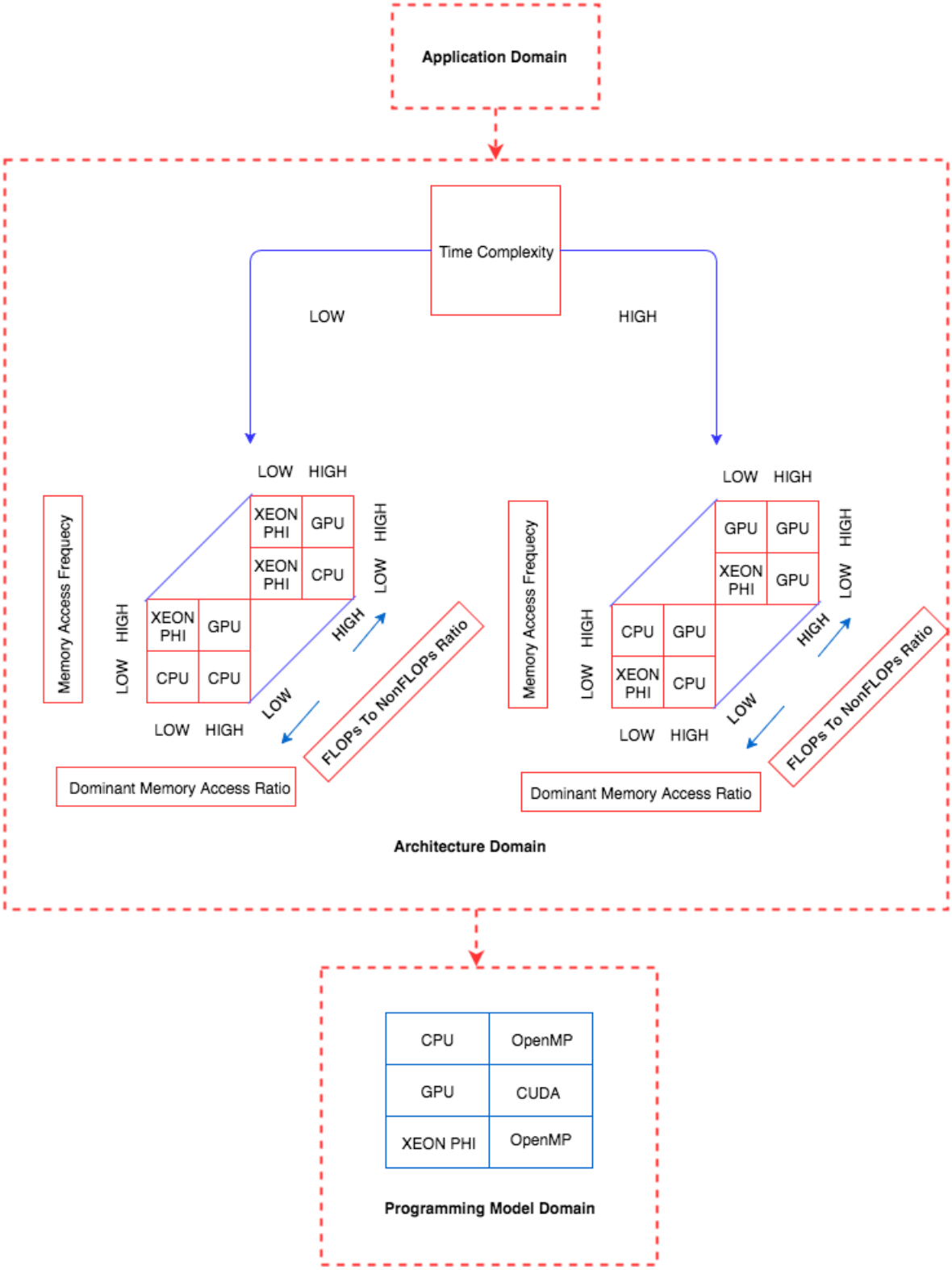

Figure 4.1: Mapping between Application, Architecture and Programming Model Domains

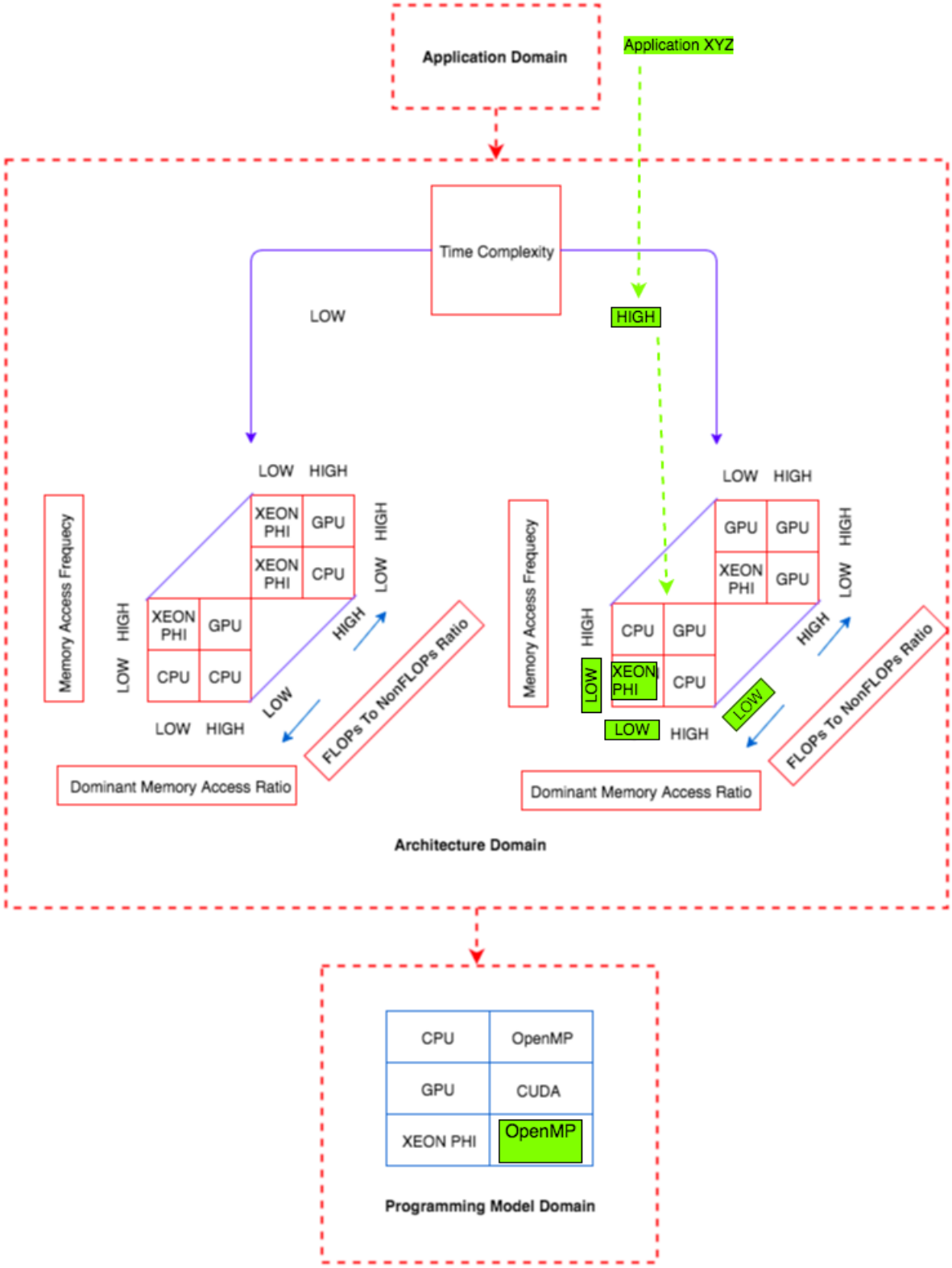

Figure 4.2: Example Mapping between xyz Application to Optimal Architecture and Programming Model

Characteristics	Description	Parameter
Computation Ratio	The amount of time required to perform computation	Time Complexity
FLOPs	The amount of floating point computation instructions	FLOPs to Non-FLOPs Ratio
Non-FLOPS	The amount of Non-floating point instructions	
L1 cache Accesses	L1 cache Hit and Miss Ratio	Dominant Memory Access Ratio
L2 cache Accesses	L2 cache Hit and Miss Ratio	
L3 / shared cache Accesses	L3 cache Hit and Miss Ratio	
Main Memory Accesses and Latency	Time required to access main memory	Memory Access Frequency
Non-uniform Memory Accesses NUMA	The amount of time required for NUMA	
I/O Ratio	The amount of time required to perform I/O operations	Assumptions: The algorithm has already performed network communication And I/O
Communication Ratio	The amount of time required to perform communication	
Network Bandwidth and Latency	Characteristics of Network used for communication	
Network Congestion	The amount of wait time due to network congestion in various runs	

Table 4.1: List of Factors that Affect Performance

1. *Time Complexity*: Time complexity of an application is often represented using Big-O-Notation. Time complexity of an algorithm summarizes how the scaling with input size impacts performance. Asymptotic upper bounds reveal an algorithm's fundamental limits. Although algorithmic complexity is a strong indicator of the performance, it alone can not be used as a deterministic identifier. For application characterization, time complexity parameter is categorized as low for complexity less than O(n log n) and high if otherwise.

   From the architecture perspective, different architectures use different techniques to accomplish good performance for higher complexities. CPU has limited number of threads but a higher core frequency. In addition to zero-overhead context switching, GP-GPU has inherent latency hiding capabilities. Xeon Phis utilize higher core frequency coupled with multi-threaded context switching but a limited thread launch. Although one can not simply rank the architectures for any given complexity, other parameters do play a role in achieving this ranking. For example, GP-GPU and Xeon Phi are better choices for applications with higher time complexity. However, if the memory transfers becomes an issue, CPU would outperform the former two.

2. *FLOPs To Non-FLOPs Ratio*:

FLOPs to Non-FLOPs ratio is a mathematical ratio of the number of FLOP instructions to the number of Non-Flop Instructions. In computational sense, an operation is the effect of an operator on an expression. A floating-point operation is any mathematical operation (such as +, -, *, /) or assignment that involves floating-point numbers. FLOPs refers to the number of FLOP instructions. (Note: This is different from the term FLOPS which is generally used as an abbreviation for floating point operations per second.) Branching and conditional control divergence operations are considered examples of non-FLOP operations. Literature such as "Roofline: An Insightful Visual Performance Model for Multicore Architectures"[28] shows the impact of FLOP operations on compute performance. From the application perspective, the ratio is considered *high* if greater than 2, i.e. if the number of FLOP instructions is at least twice the number of non-FLOP instructions and *low* otherwise.

CPU has lower performance in terms of FLOPS but it handles branching and divergence in control sequence gracefully. While GP-GPU performs very well with FLOPs, non-FLOPs such as branch divergence hamper the GP-GPU performance significantly. Xeon Phi fairs well against the GP-GPU in terms of FLOPS with reduced negative impact of branching, divergence and synchronization.

3. *Dominant Memory Access Ratio*: Dominant Memory Access Ratio is a mathematical ratio of dominant access in Memory Hierarchy, i.e. on-chip to off-chip. The access time for the on-chip cache memory accesses is eminently less than that for an off-chip non-cache memory access. The cost of access time has a direct impact on the total execution time and thus the compute performance. For an application, dominant memory access ratio is categorized as *high* if greater than 1, i.e. if the number of on-chip accesses exceeds the number of off-chip accesses. For higher ratio, GP-GPU with very fast local memory and Xeon Phi with dual memory bank rings achieve good performance. Off-chip access usually affect the co-processor's performance as the memory transfers from host to device and device to host constrain the performance.

CPUs with traditional architecture see minimal benefit from higher on-chip access while not suffering significantly by off-chip accesses.

4. *Memory Access Frequency*: The effect of memory access behaviour depends on the frequency of access. The ratio of dominant memory access can have varying degrees of impact depending on the frequency of these applications. Applications with low memory access frequency may not be significantly affected by higher off-chip memory access behavior. The characterization of application for memory access frequency is determined as *high* or *low* by evaluating the number of memory accesses against the total number of operations. If the memory operations constitute more than 25% of total operations, the memory access frequency is said to be *high* and *low* otherwise.

4.2.2 Programming Model Selection

Programming models discussed in Section 3.3 not only differ in terms of the programming effort required, but application performance also varies significantly for different architectures. Thus a thorough investigation is required to correlate application performance with a combination of architecture and programming model. A methodology adopted in this research is to utilize a representative algorithm for scientific applications and to compare measured run time for implementations of the same with a variety of popular programming models on available architectures.

4.3 Measures

This Section explains the measures used in this research to quantify different characteristics involved in the experimentation. In order to validate the hypothesis explained in Section 4.2, this research uses the following measures:

(1) To validate the architecture to application mapping model used to select the appropriate architecture, the model is tested for applications from a benchmark suite consisting of different computation/communication idioms, i.e. dwarfs called OpenDwarfs.

OpeenDwarfs is implemented in OpenCL, which is a portable language and thus will not favor any architecture in particular. The model is used to predict a matching architecture, given the application characteristics. The prediction is then verified by comparing actual run time by running the code on different architectures. The accuracy is judged based on whether the model predicted the best architecture for the given application. The applications chosen belong to following algorithmic domains/classes:

1. Dense Linear Algebra
2. Sparse Linear Algebra
3. Spectral Methods
4. N-Body Methods
5. Structured Grid
6. Unstructured Grid
7. Dynamic Programming
8. Graphical Models
9. Combinational Logic

(2) To verify the programming model selection strategy of using the native language for the architecture of choice, a representative application (k-means algorithm) is run on multiple architectures in native languages as well as in portable language to see the effect of language variability for a given application.

(3) To establish the effectiveness of the parallelization strategy, a representative algorithm from scientific computing (Lower-Upper decomposition) is selected and is approached with two methodologies, trial and error methods as well as the "Architecture First, Language Later" parallelization strategy. The two methods are quantified based on achieved results, i.e run time of implementations, parallelization efforts in terms of man-hours for

code conversion, amount of necessary accesses to hardware and degree of knowledge and expertise required for the method. The overall results and efforts incurred are contrasted for both methods.

4.4 Setup

Table 4.2, lists the different architectures used in this study for experimentation with different implementations. The architectures include NVIDIA Tesla K20 GPU, Intel Xeon Phi 5110P many core co-processor and Intel Xeon E5-2680 multi-core single node CPU. The table lists the specification hardware characteristics that significantly affects the performance of an application running on the architecture. Each architecture has its own advantages and disadvantages making it more suitable for a particular type of computation/communication pattern. The impact and suitability of hardware capabilities of a given architecture is very complex and thus can not be judged based on specific numbers alone.

Device	**Tesla K20**	**Intel Xeon Phi 5110P**	**Intel Xeon E5-2680**
Single Precision (Gflops/s)	3520	2021.76	665.6
Double Precision (Gflops/s)	1170	1010.88	332.8
Cores (*)	2496	60	8
Memory Bandwidth (GB/s)	208	320	51.2
Memory Size (GB)	5	8	32

Table 4.2: Device Specifications

Intel Xeon-Phi E5110P: The current generation of Xeon-Phi Coprocessors are located on the PCI Express (PCIe2) with 60 processor cores each running at 1.238Ghz and interconnected via a bi-directional ring with a theoretical peak of 330GB/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 8GB DDR5 global Memory. In off-load mode, MPI processes are executed on one of the architectures: either the coprocessors or the host processors. Off-load mode has the benefit of targeting specific computation onto dedicated architectures using the available off-load directives. Furthermore, the compiler manages the

data movement and parallelism with hints from the developer. In the symmetric mode, MPI processes uniformly span the domains of both the host and the coprocessor architectures. The developer must explicitly manage parallelism across the different architectures but with more control over how the application executes. The coprocessor-only mode is a subset of symmetric mode, with all MPI processes being confined to the Xeon-Phi architecture alone.

Nvidia K20 GPU: The Nvidia K20 are Kepler Series GPUs also accessible via the PCIe2 bus with 2496 cores and a core clock speed of 732 MHz. The K20 architecture has 5GB DDR5 global memory available at 208GB/s bandwidth. Each core has L1 cache and L2 data cache available in 3 configurations (16KB L1/48KB L2, 32KB L1/32KB L2, and 48KB L1/16KB L2).

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 a last level cache (L3 CACHE) and, a local integrated memory controller connected via an Intel Quick Path interconnect. The cores share 32GB DDR3 global memory with 32KB data and instruction L1 cache, a 256 KB unied L2 cache, and 8 MB L3 Cache.

4.5 Summary Of Experimental Design

This chapter provided details of the model used for application to architecture mapping via application characterisation and programming model selection via native language mapping to selected architecture. Additionally, the four parameters namely, Time Complexity, FLOPs to Non-FLOPs ratio, Dominant Memory Access Ratio and Memory Access Frequency were also described. This chapter also outlined the validation metrics used to determine the effectiveness of the parallelization strategy proposed in this thesis. The next chapter discusses the details of the mathematical background as well as implementation of Lower Upper Decomposition Algorithm as the case study.

Chapter 5

Case Study: Linear Algebra - Lower Upper Decomposition

In this chapter, we provide background on the Lower Upper Decomposition algorithm as an enhanced Gaussian Elimination technique to solve a system of linear equations. The chapter is structured as follows. Section 5.1 describes the basics of solving linear equation and details the related work in this area. Section 5.2 provides mathematical background for the Gaussian Elimination and Lower Upper (LU) Decomposition. The design modifications are articulated in Section 5.3. The chapter is concluded in Section 5.4 with detailed parallel implementation methodology for trial and error methods as well as the 'Architecture First, Language Later" parallelization strategy.

5.1 Introduction to solving system of linear equations

Given a system $\boldsymbol{Ax} = \boldsymbol{b}$, a solution can be derived via a variety of different methods.Gaussian elimination is usually selected when a unique solution is known to exist, and the coefficient matrix is not sparse. There are several papers that employ various parallel approaches to solving a linear system with Gaussian Elimination [29, 30]. This case study examines the performance of different variations of Lower Upper Decomposition of Gaus-

sian Elimination algorithm when parallelized for multi-core CPU, GPU and CPU+GPU environments with C, MPI, CUDA and MPI+CUDA hybrid implementations. Some work has been done on a parallel algorithm for Gaussian Elimination by S. F. McGinn and R. E. Shaw [29] which implements it in a shared memory environment using OpenMP, and in a distributed memory environment using MPI. Popular Benchmarks such as the Rodinia Benchmark Suite includes a CUDA and OPENCL Implementation of the naïve Gaussian Elimination [30]. The most efficient implementations of Gaussian Elimination are designed using the Basic Linear Algebra (BLAS) package [31] of primitive functions or using numerical libraries Linear Algebra Package (LAPACK) [32], Numerical Algorithms Group (NAG) [33], Math Kernel Library (MKL) [34], or MATLAB [35]. But none of the efforts cater to a heterogeneous environment and also fail to evaluate LU decomposition variation of the algorithm with storage modifications for minimized memory footprint.

5.2 Mathematical Background

5.2.1 Gaussian Elimination

The Gaussian elimination approach is applied to solve a general set of n equations and n unknowns as shown in Equation 5.1

$$\left.\begin{array}{c} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + ... + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + ... + a_{2n}x_n = b_2 \\ ..... \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + ... + a_{nn}x_n = b_n \end{array}\right\} \tag{5.1}$$

Gaussian elimination consists of two steps :

1. *Forward Elimination of Unknowns:* In this step, the unknown is eliminated in each equation, one at a time. The end goal of this step is to have the equations reduced in such a way that we have an equation with only one unknown variable.

2. *Back Substitution:* In this step, we find the value for the unknown in the last equation result of forward elimination. Once this value is found, it is substituted in the equation with two unknowns so as to reduce that equation into one with only one unknown. In each step, one equation is solved. The end result of this step is the solution for all unknowns.

There are pitfalls of the above-mentioned naïve Gauss Elimination Method:

1. *Division by zero:* It is possible for a divide by zero condition to occur during the steps of forward elimination or back substitution as the algorithm in itself does not have any checks against these possibilities. One way to avoid this is to use partial pivoting to swap two rows of coefficients as needed.

2. *Round-off error:* The naïve Gauss elimination method is prone to round-off errors. As we solve equation by equation, the round-off value errors propagate to the next equation. This condition is mainly significant when same equations have large coefficients and others have very small coefficients. Scaling can be employed in such cases to lower the impact of round-off errors.

Example : Consider the system of linear equations given in Equation 5.2

$$\left.\begin{aligned} x - 3y + z &= 4 \\ 2x - 8y + 8z &= -2 \\ -6x + 3y - 15z &= 9 \end{aligned}\right\} \tag{5.2}$$

First, we eliminate x from the second equation by subtracting 2 times the first equation from the second. This yields the equivalent system given in Equation 5.3

$$\left.\begin{aligned} x - 3y + z &= 4 \\ -2y + 6z &= -10 \\ -6x + 3y - 15z &= 9 \end{aligned}\right\} \tag{5.3}$$

Next, we add 6 times the first equation to the third, to eliminate x from the third equation as well, yielding the system in Equation 5.4

$$\left.\begin{aligned} x - 3y + z &= 4 \\ -2y + 6z &= -10 \\ -15y - 9z &= 33 \end{aligned}\right\} \tag{5.4}$$

Then, we eliminate y from the third equation by subtracting 15/2 times the second equation from it, which yields the system in Equation 5.5

$$\left.\begin{aligned} x - 3y + z &= 4 \\ -2y + 6z &= -10 \\ -54z &= 108 \end{aligned}\right\} \tag{5.5}$$

This system is in upper-triangular form, because the third equation depends only on z, and the second equation depends on y and z. This reduction using row operations is called as *forward elimination*. Because the third equation is a linear equation in z, it can easily be solved to obtain z= -2. Then, we can substitute this value into the second equation, which yields -2y - 12 = -10. This equation only depends on y, so we can easily solve it to obtain y = -1. Finally, we substitute the values of y and z into the first equation to obtain $x = 3$. This process of computing the unknowns from a system that is in upper-triangular form is called *back substitution* [36].

5.2.2 Lower-Upper Decomposition

To solve several linear systems with the same **A** with different **b** would involve repetitions of steps when using Gaussian elimination for each linear system. A more efficient and accurate way to solve such systems is LU-decomposition, which in effect records the steps of Gaussian elimination. LU decomposition essentially decouples the factorization phase, which is usually compute-intensive, from the actual solving phase. Factorization of

A is done only once and then this factorized A is used with every new vector **b** for actual solving. Thus the factorization efforts need not be duplicated for every new **b**.

Given a matrix **A**, the aim is to build a lower triangular matrix **L** and an upper triangular matrix **U** which has the following property: diagonal elements of **L** are unity and **A**=**LU**. Let **A** be n x n matrix. LU factorization is a process for decomposing **A** into a product of a lower triangular matrix **L** (diagonal elements of **L** are unity) and an upper triangular matrix **U** such as $\boldsymbol{A} = \boldsymbol{LU}$ as given in Equation 5.6.

$$L = \begin{pmatrix} 1 & & & \\ l_{21} & 1 & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \dots & 1 \end{pmatrix} U = \left.\begin{pmatrix} u_{11} & u_{12} & \dots & u_1 n \\ & u_{22} & \dots & u_2 n \\ & & \ddots & u_{n-1,n} \\ & & & u_{nn} \end{pmatrix}\right\} \tag{5.6}$$

For the resolution of linear system : $\boldsymbol{Ax} = \boldsymbol{b}$, the system becomes as in Equation 5.7:

$$LUx = b \Leftrightarrow \left\{ \begin{array}{ll} Ly = b & (1) \\ Ux = y & (2) \end{array} \right\} \tag{5.7}$$

We solve the system (1) to find the vector **y**, then the system (2) to find the vector **x**. The triangular shape of the matrices facilitates the resolution [37].

There are three factorization methods:

1. Crout Method: diagonal (**U**) = 1;
2. Doolittle Method: diagonal (**L**) = 1;
3. Choleski Method: diagonal (**U**) = diagonal (**L**) =1

The cost of factorizing the matrix **A** into **LU** is $O(N^3)$. After **A** is factorized, the cost of solving i.e. the cost of solving **LUx** = **b** is $O(N^2)$, since the cost of solving a triangular system scales as $O(N^2)$. (Note that to solve **LUx** = **b**, **Ly**=**b** is solved first and then **Ux**=**y**. Solving **Ly**=**b** and **Ux**=**y** each costs $O(N^2)$.) Hence, if you have '**r**' right

hand side vectors **b1,b2,...,br**, once you have the LU factorization of the matrix **A**, the total cost to solve **Ax1=b1,Ax2=b2,...,Axr=br** scales as $O(N^3 + rN^2)$. On the other hand, the total cost for repeating Gauss elimination for each right hand side vector **b** scales as $O(rN^3)$ as each Gauss elimination independently costs $O(N^3)$.

Example: Consider the system of linear equations (same as in Subsection 5.2.1 Equation 5.2)

$$\left.\begin{aligned} x - 3y + z &= 4 \\ 2x - 8y + 8z &= -2 \\ -6x + 3y - 15z &= 9 \end{aligned}\right\} \tag{5.8}$$

This can be represented as given in Equation 5.9:

$$\left.\begin{bmatrix} 1 & -3 & 1 \\ 2 & -8 & 8 \\ -6 & 3 & -15 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ -2 \\ 9 \end{bmatrix}\right\} \tag{5.9}$$

First, we decompose the **A** matrix into a product of two matrices **L** and **U** where **L** is a lower triangular matrix and **U** is an upper triangular matrix. This yields the equivalent system given by Equation 5.10:

$$\left.A = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -6 & -15/2 & 1 \end{bmatrix} \begin{bmatrix} 1 & -3 & 1 \\ 0 & -2 & 6 \\ 0 & 0 & -54 \end{bmatrix} = LU\right\} \tag{5.10}$$

LUx =b can be represented as given in Equation 5.11:

$$\left.\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -6 & -15/2 & 1 \end{bmatrix} \begin{bmatrix} 1 & -3 & 1 \\ 0 & -2 & 6 \\ 0 & 0 & -54 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ -2 \\ 9 \end{bmatrix}\right\} \tag{5.11}$$

$$Y = \begin{bmatrix} Y1 \\ Y2 \\ Y3 \end{bmatrix} = \begin{bmatrix} 1 & -3 & 1 \\ 0 & -2 & 6 \\ -6 & -15/2 & 1 \end{bmatrix} \quad \left.\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right\} \tag{5.12}$$

Next,We solve the system in Equation 5.7 (1) **LY=b** to find the vector **Y** as given by Equation 5.13.

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -6 & -15/2 & 1 \end{bmatrix} \quad \begin{bmatrix} Y1 \\ Y2 \\ Y3 \end{bmatrix} = \begin{bmatrix} 4 \\ -2 \\ 9 \end{bmatrix}$$

$$\left.\begin{bmatrix} Y1 \\ Y2 \\ Y3 \end{bmatrix} = \begin{bmatrix} 4 \\ -10 \\ 108 \end{bmatrix}\right\} \tag{5.13}$$

Then,we solve the system given in Equation 5.7 (2) **Ux=y** to find the vector **x**.

$$\begin{bmatrix} 1 & -3 & 1 \\ 0 & -2 & 6 \\ 0 & 0 & -54 \end{bmatrix} \quad \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ -10 \\ 108 \end{bmatrix}$$

This yields the system given by Equation 5.14,

$$\left.\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 3 \\ -1 \\ -2 \end{bmatrix}\right\} \tag{5.14}$$

5.3 Important variation for LU Decomposition Algorithm

A design decision was made to change the storage structure for the said **L** and **U** matrices but the Doolittle method's conventions were followed to denote the two said matrices. Since the diagonal elements of **L** are 1's and it is assumed to be a known fact,

this information becomes redundant. Additionally, to take advantage of the two triangular matrix structures, the two matrices were appended to form one matrix, which has the same size as matrix **A** i.e. nxn. This variation reduces the memory and communication needs by half as opposed to having two different matrices. A serial C code was developed to transform matrix **A** to **LU** and then solving **Ly**=**b** followed by **Ux**=**y** using forward and backward substitution respectively.

5.4 Parallel Implementation Methodology

The parallelization involves taking the serial code for the LU decomposition algorithm and transforming it into a parallel code (MPI or CUDA or MPI+CUDA) capable of running on a parallel platform Multicore CPU, GP-GPU or XEON PHI.

5.4.1 Trial and Error Method

Trial and error methods involve experimenting with a variety of combinations of parallel platforms and programming model until an optimal performance is achieved. The following algorithm serves as a guideline for these efforts.

```
Result: Parallel code with optimal performance/speedup
Given: Serial Code
Combo[(parallel programming model, parallel platform)]:
 [(MPI,CPU),(CUDA,GPU),(MPI+CUDA, CPU+GPU)]
i ← 0    BestTime ← SerialTime    choice ← -1
while i < 3 do
    Combination for experiment ← combo[ i ]
     Convert code to parallel code for selected programming model
     Run the code on the selected parallel platform
     Measure run time (ParallelRunTime) and calculate speedup
    if CalculatedRunTime < BestTime then
        BestTime = ParallelRunTime
        choice = i
    end
    i ++;
end
```

Algorithm 1: Trial and Error Methods For Optimal Parallelization

As observed in the Algorithm 1, trial and error methods involve development of code in MPI, CUDA and MPI+CUDA hybrid. It also necessitate the access to a Multi-core CPU as well as a GPU platform. In order to successfully conduct this experimentation the programmer/scientist must have knowledge of different programming models and have expertise in coding in the aforementioned languages. Thus trial and error methods can be considered as a resource intensive methodology.

5.4.2 "Architecture First, Language Later" parallelization strategy

The 3 step "Architecture First, Language Later" parallelization strategy is implemented as follows for LU Decomposition:

Step (1) choose a compatible combination of architecture and parallel programming language using the Figure 4.1. Application characterization of LU Decomposition is as follows:

Time complexity : High

FLOPs To NonFLOPs Ratio: Low

Dominant Memory Access Ratio: High

Memory Access Frequency: High

Predicted optimal combination: (CUDA,GP-GPU) by using the Figure 4.1 as explained in Subsection 4.2.1.

Step (2) translate base code/algorithm to a parallel language i.e. transform serial LU Decomposition code into CUDA code

Step (3) optimize and tune the code for GP-GPU using basic GP-GPU optimization techniques such as Memory Optimization, Execution Configuration Optimization, and Instruction Optimization.

As observed in step 2, the "Architecture First, Language Later" parallelization strategy involves converting code to just one language and needs access to one platform as opposed to multiple combinations in trial and error methods. Thus the "Architecture First, Language Later" parallelization strategy can be termed as a resource effective strategy that provides comparable results with significantly reduced parallelization efforts in terms of development, access to hardware and knowledge and expertise.

5.5 Summary Of Case Study: LU Decomposition Algorithm

This chapter explained the mathematical concepts involved in the Lower Upper Decomposi- tion algorithm as an enhanced Gaussian Elimination technique to solve a system of linear equations, along with a numerical example including solution. This chapter also

included guidelines for implementing Trial and Error methods as well as the “Architec- ture First, Language Later” parallelization strategy.

Chapter 6

Results and Analysis

This chapter presents the results for individual steps of the propsed "Architecture First, Language Later" parallelization strategy along with observations from the case study. Section 6.1 details the results for the quantitative comparison of run-times for K-means Algorithm implemented on a variety of architectures in different programming languages. Section 6.2 discusses the accuracy of optimal architecture prediction using the qualitative model for 9 test applications from the OpenDwarfs suite. Section 6.3 presents the case study observations for LU Decomposition algorithm implemented in MPI, CUDA and a hybrid MPI+CUDA. This chapter is concluded with summary in Section 6.4.

6.1 Optimal Language : Quantitative Comparison Results

Figure 6.1 shows performance comparison of different implementations on different parallel platforms for K-means algorithm with 1638400 items and 20 clusters. The Figure 6.1 illustrates that strong performance gain can be obtained by using dedicated programming models. However, the relative performance for architecture remains the same even for various programming models. It is evident that native languages provide significantly better performance for any given architecture. Thus this quantitative comparison demonstrates that once an architecture is selected, the native language is the most appropriate selection

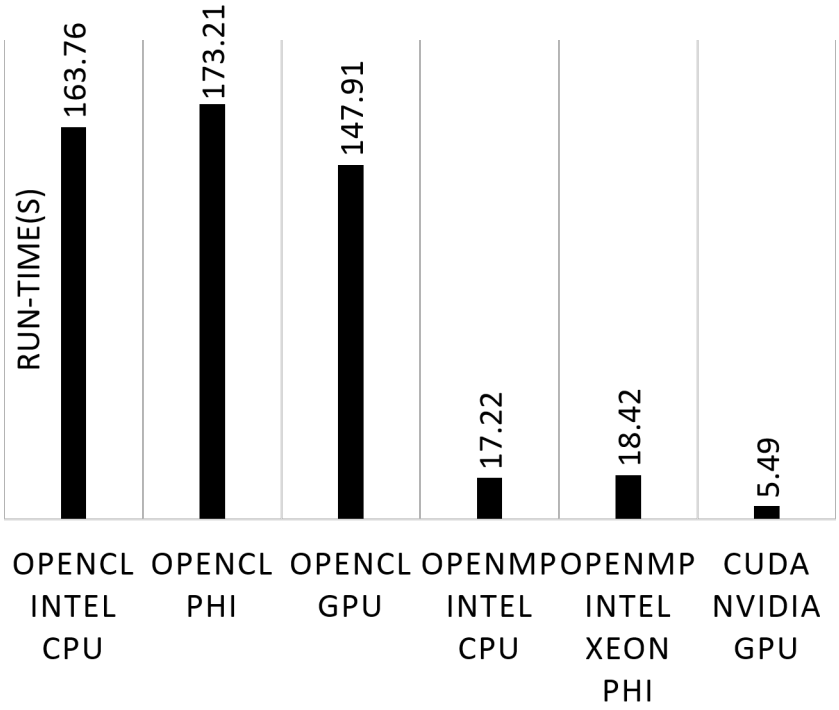

Figure 6.1: Performance Comparison for K-means Implementations

to obtain best achievable performance.

6.2 Optimal Architecture: Verification of Qualitative Model

To verify the accuracy of the qualitative model depicted in Figure 4.1, dwarfs are used instead of traditional benchmarks. A dwarf is an algorithmic method that captures a pattern of computation and communication [2] that are most common in scientific applications. A benchmarking suite, called OpenDwarfs, implements 13 dwarfs in OpenCL language. This benchmark is well suited to evaluate the qualitative model for application to architecture mapping, since the dwarfs represent the most common patterns found in scientific algorithms and thus these results can be extrapolated to real life scientific HPC applications. Additionally, since the benchmark is implemented in OpenCL, it does not favor any architecture resulting in a fair comparison. OpenDwarfs includes applications from many different algorithmic classes such as linear algebra, divide and conquer, combinational

logic, grid, spectral methods etc. Each application is characterized and rated as high or low on the four parameters as mentioned in Section 4.1. The qualitative model is then used to predict the most suitable architecture. This section verifies the accuracy of this prediction by comparing the actual run times. For the purpose of this research, the focus is on seven numerical methods identified by Phillip Colella [3].

1. **Dense Linear Algebra : K-Means Algorithm** - Dense linear algebra involves dense matrix and vector operations. Applications such as data mining use a variety of linear algebra algorithms such as the K-means clustering algorithm. K-means clustering aims to partition a collection of data objects into a finite number of clusters resulting in the data space into clusters containing similar samples. Any data point is classified as a member of the cluster when the distance between the center and the data point is less than center of any other cluster.

   Application Characterization - Time complexity for Lloyds algorithm for K-means clustering is often given as O(n*K*I*d) where n is number of data points, K represents the number of clusters, I is number of iterations, and d stands for number of attributes [38]. As the product of the number of clusters and their dimensions is comparable to number of data points, it can be termed as *High*. As the algorithm has a high ratio of computation to communication operations, the FLOPS to non-FLOPs ratio is considered *high*. The algorithm involves a lot of data dependency among threads, thus a *high* ranking on memory access ratio as well as the memory access frequency.

   Architecture Prediction - All high ranking in application characterization, leads to prediction to be GP-GPU, as depicted in Figure 4.1.

   Prediction Verification - As observed in the following Table 6.1 and Figure 6.2, GP-

K-means,Data Size	Single Node	Single Phi	Single-GPU
500k	88.388174	69.873352	61.45915767
1000k	252	187	145
1500k	488	332	250
2000k	799	542	382

Table 6.1: Dense Linear Algebra : K-Means Algorithm run-time (sec), varying data sizes

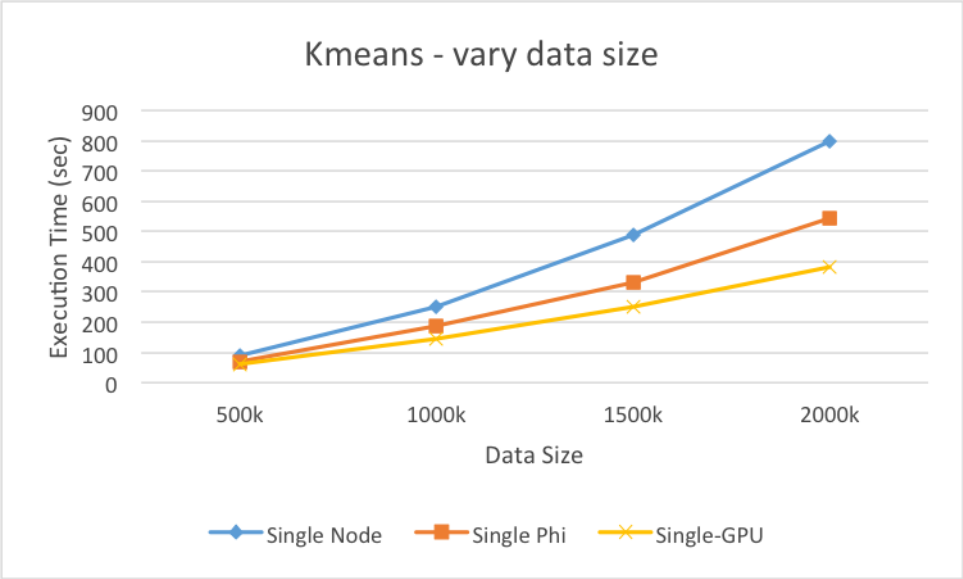

Figure 6.2: Dense Linear Algebra: K-Means Algorithm results, varying data sizes

GPU achieves best performance with the lowest run-time for all data sizes. For smaller data sizes, the algorithm converges quickly and thus all architectures perform comparable but as the algorithm scales, the difference in run-time becomes significant.

2. **Sparse Linear Algebra: Sparse Matrix Vector multiply (SPMV)** - Sparse linear algebra involves similar matrix vector operations such as dense linear algebra but with a matrix with very few non-zero elements. SPMV involves multiplication of sparse matrix [**A**]and a vector [**x**], resulting in a vector [**y**]. As most elements of a sparse matrix are zero, a variety of formats exist for memory/space efficient representation of a sparse matrix. One such format is Compressed Sparse Row (CSR), that

represents the sparse matrix using three dense one dimensional vectors. These vectors contain all non-zero elements, their column indices and pointers to elements in the first array that starts each row [39]. Application kernels such as partial differential solvers and finite element analysis use SPMV as part of standard computation.

Application Characterization - Sparse matrices have very few non-zero elements. SPMV problems are significantly different from dense linear algebra in terms of memory access patterns. Sparse algebra involves a lot of indirect and irregular memory access pattern. Although the SPMV classifies as highly intensive as per time complexity and FLOPs-to-Non-FLOPs ratio, the dominant access ratio is *low* as opposed to the K-means algorithm. The memory access frequency is observed to be *high*.

Architecture Prediction - *Low* ranking on Dominant memory access ratio with *high* rankings on rest of the three parameters in application characterization, results in the GP-GPU being predicted as the most optimal choice, as depicted in Figure 4.1.

SPMV, Data Size	Single Node	Single Phi	Single-GPU
16384	7.447	50.997	5.47
32768	29.739	127.442	18.207
65536	115.856	355.33	69.586
131072	454.111	1246.57	268.913

Table 6.2: Sparse Linear Algebra: Sparse Matrix Vector multiply (SPMV) run-time (sec), varying data sizes

Prediction Verification - Table 6.2 and Figure 6.3, demonstrates the run-times on different architectures. One distinct variation from the SPMV results is that Single node performs better than Xeon Phi. This can be attributed to higher demands fpr synchronization needed in threads for the Xeon Phi as per this implementation of SPMV algorithm. GP-GPU out performs due to less intensive bandwidth requirements of this algorithm.

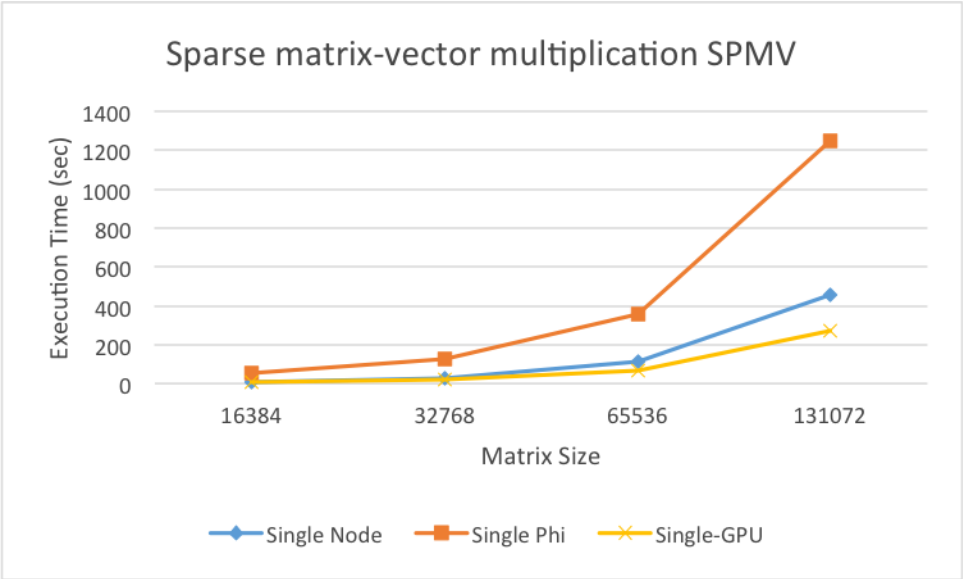

Figure 6.3: Sparse Linear Algebra: Sparse Matrix Vector multiply (SPMV) results, varying data sizes

3. **Spectral Methods: Fast Fourier Transform (FFT)** - Spectral methods are techniques used to numerically solve certain partial equations, which involves transformation of spatial/temporal data. The execution is generally multi-stage in nature, with dependencies between stages forming a butterfly pattern of computation. These methods are used in quantum mechanics and fluid dynamics application. A fast Fourier transform (FFT) algorithm transforms the signal from spatial/temporal domain to a representation in the frequency domain and vice versa.

   Application Characterization - FFT has time complexity of O($n$log$n$) where n is the data size, thus it is termed as *low* on time complexity parameter. FLOPs to Non-FLOPs ratio is *high* for FFT due to computational component of the algorithm. Dominant memory access ratio is relatively *low* with *low* frequent access.

Architecture Prediction - With *high* FLOPs to Non-FLOPs ratio, coupled with *low* rating on other three parameters, Xeon Phi is deemed as the most suitable architecture for FFT application as per the model in Figure 4.1.

FFT, Data Points, Data Size	Single Node	Single Phi	Single-GPU
65536	0.973729	0.919067	0.463484
524288	9.60265	6.11975	3.43868
6291456	144.348	115.651	39.8
16777216	388.722	301.879	106

Table 6.3: Fast Fourier Transform (FFT) run-time (m-sec), varying data sizes

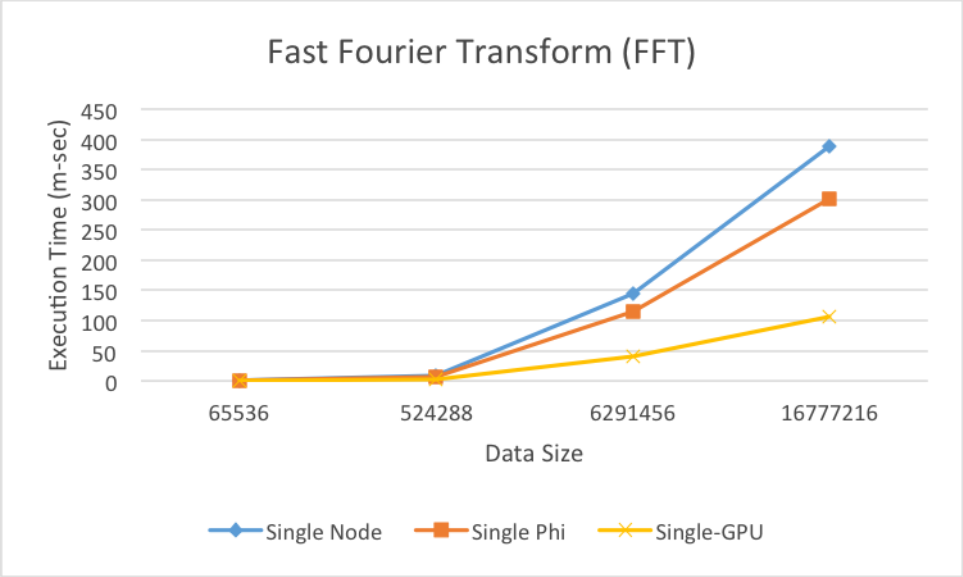

Figure 6.4: Spectral Methods: Fast Fourier Transform (FFT) results, varying data sizes

Prediction Verification - GP-GPU takes approximately half the time for execution as compared to other two architectures, as observed in the Table 6.3 and Figure 6.4. Thus the prediction on Xeon Phi is inaccurate.

4. **N-body Methods: GEM** - Interactions among many discrete points are processed using N-body methods. This processing involves a large numbers of independent cal-

culations at each step, followed by all-to-all communications between steps. This GEM package implementation of N-body methods calculates the charge at each point resulting from the effect of surrounding atoms along the surface of a molecule [40].

Application Characterization - The GEM implementation has O($M * N$) complexity where N is number of atoms and M is number of points along the surface. Hence, this is termed as *high* time complexity. The FLOPs to Non-FLOPs ratio, is also *high* as the algorithm is compute intensive. The Dominant memory access ratio is *low*, as most access are off chip. The memory access frequency is *high*.

Architecture Prediction - With *low* ratings on dominant memory access ratio, while *high* ratings on time complexity and FLOPs to Non-FLOPs ratio as well as memory access frequency, the Xeon Phi is predicted to be the most optimal choice for GEM as observed in Figure 4.1.

GEM, Data Points, Data Size	Single Node	Single Phi	Single-GPU
24, 382	0.302023	1.176967	1.342539
91, 1441	0.412428	1.227809	1.398127
1268, 25086	17.1301	6.480024	3.784541
30780, 476040	662.28605	191.8663	55.54715

Table 6.4: N-body Methods: GEM run-time (sec), varying data sizes

Prediction Verification - Table 6.4 and Figure 6.5 depict that the prediction holds true upto a pivotal data size. Once the algorithm scales beyond this point, the GP-GPU architecture performs better due to its capacity for coalesced memory access of large size. Thus in this case the prediction of the qualitative model is only partially applicable and thus inaccurate.

5. **Structured Grid: Speckle Reducing Anisotropic Diffusion (SRAD)** - SRAD

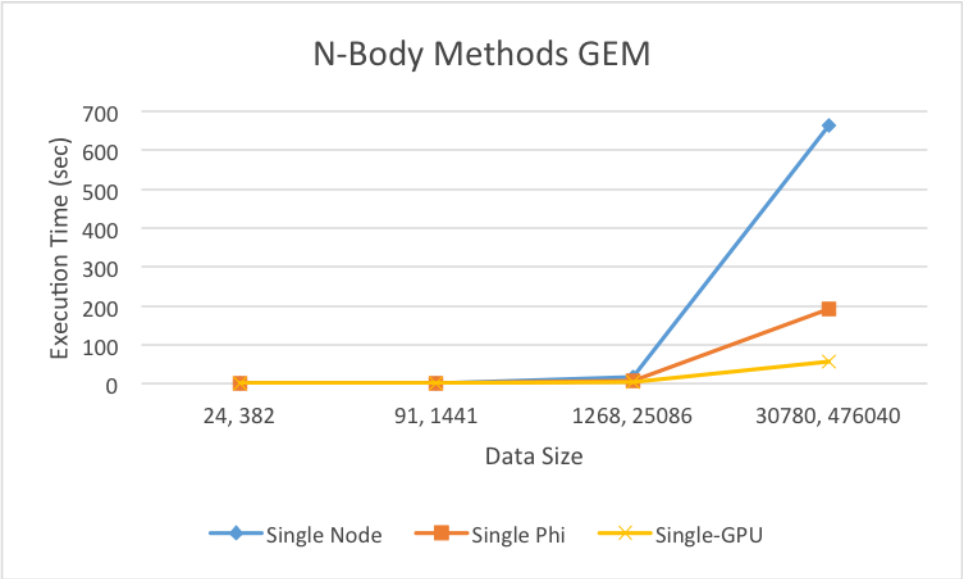

Figure 6.5: N-body Methods: GEM results, varying data sizes

is a diffusion method based on partial differential equations (PDEs). It is commonly used in ultrasonic and radar imaging applications. SRAD aims to filter out the noise, known as speckles, without loosing important image details. SRAD involves various stages such as image extraction, continuous iterations over the image, and image compression [41].

Application Characterization - As the processing occurs at each pixel in the image, SRAD has *high* time complexity. The FLOPs to non-FLOPS ratio is *high*. The dominant memory access ratio is also *high*. The memory access frequency is *low* since although stages depends on the output of previous stage and the individual pixels can be processed independently.

Architecture Prediction - With *low* memory access frequency and *high* ratings on the other three factors, the GP-GPU is predicted as the optimal architecture as observed in Figure 4.1.

SRAD, Data Points, Data Size	Single Node	Single Phi	Single-GPU
1024	1.370475	2.297683	1.658712
2048	4.559999	5.662703	2.559021
4096	18.831808	18.087546	6.566177
8192	72.423335	66.062827	22.687956
12288	163.464418	140.300742	49.911433

Table 6.5: Structured Grids: SRAD run-time (sec), varying data sizes

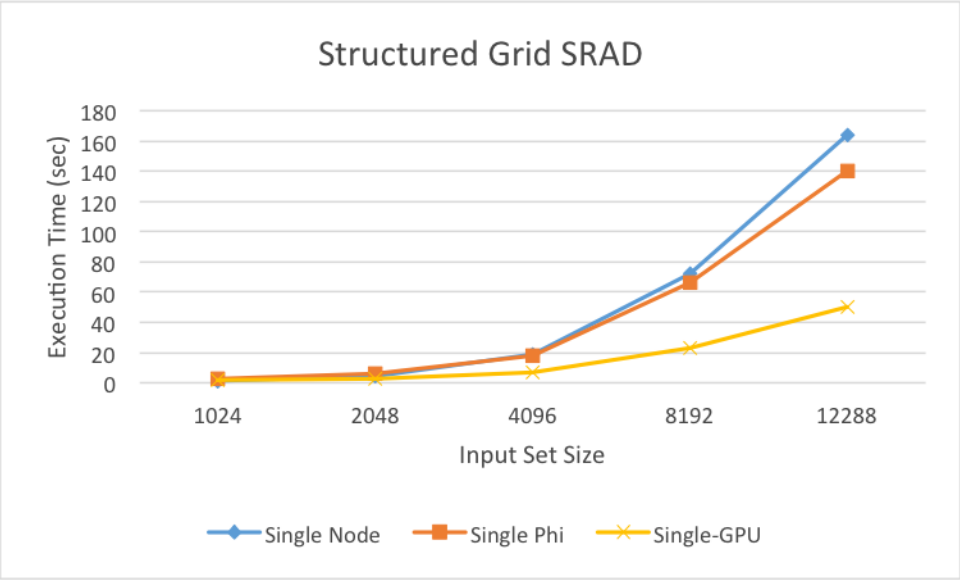

Figure 6.6: Structured Grids: SRAD results, varying data sizes

Prediction Verification - Table 6.5 and Figure 6.6 show that the prediction of the GP-GPU is valid. This algorithm seems to benefit from the coalesced memory accesses utilized in GP-GPU, which aligns perfectly with the data needs of the algorithm.

6. **Unstructured Grid: Computational Fluid Dynamics (CFD)** - Unstructured grids are often used in finite element analysis when the input to be analyzed has an irregular shape. Unlike structured grids, unstructured grids require a list of the connectivity, which specifies the way a given set of data points are arranged to represent

the individual elements. Computation Fluid Dynamics solver applications are based on 3-Dimensional (3D) Euler equations for compressible flow [42].

Application Characterization - The time complexity of this algorithm is *high* as each results depends on its neighbouring data points. The FLOPs to non-FLOPs ratio is *low*. As the access of unstructured data is random, the dominant memory access ratio is *low*. The frequency of memory access is *high* due to its dependency on adjacent data points.

Architecture Prediction - The CPU is predicted as the most suitable architecture as per the evaluation of the four parameters as explained in Figure 4.1.

CFD, Input Data Sizes	Single Node	Single Phi	Single-GPU
fvcorr.097k	6.786569	8.572833	7.138944
fvcorr.193k	13.330695	14.585241	13.00784
missile.0.2M	15.717012	17.572076	15.56318

Table 6.6: Unstructured Grid: Computational Fluid Dynamics (CFD) run-time (sec), varying data sizes

Prediction Verification - Table 6.6 and Figure 6.7 demonstrate that CPU and GP-GPU performance is comparable with CPU run time being the lowest for all data sizes. Additional computing powers of the GP-GPU and Xeon Phi is not advantageous since the communication overhead is high due to unstructured data storage and dependency. The prediction of the CPU is correct.

7. **Dynamic Programming: Needleman-Wunsch (NW)** - Needleman-Wunsch is an optimal matching algorithm or an algorithm that employs the global alignment technique to align two Deoxyribonucleic acid (DNA) sequences. This algorithm involves three steps: Initialization of a 2 dimensional matrix, scoring the alignment and

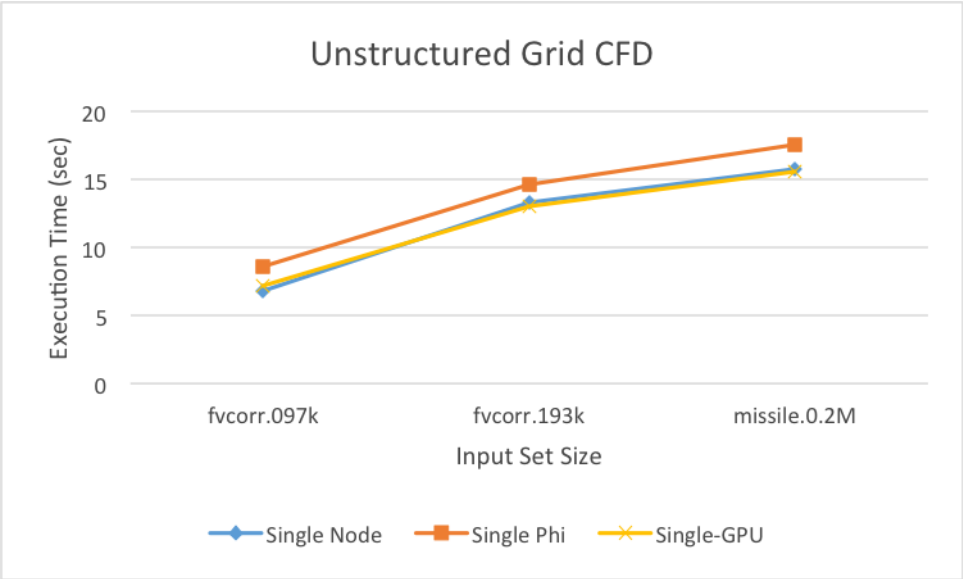

Figure 6.7: Unstructured Grid: Computational Fluid Dynamics (CFD) results, varying data sizes

filling it in the matrix and, backtracking through the array to return optimal alignment [43].

Application Characterization - The time complexity for NW algorithm is O(m*n) where m and n represent the length of the two sequences which leads to a *high* rating for this parameter [44]. The FLOPs to non-FLOPs ratio is also *high*. Due to the nature of the algorithm, the dominant memory access ratio is *low* with a *low* frequency of memory access.

Architecture Prediction - NW has *high* ratings on time complexity coupled with *low* rating on the FLOPs to non-FLOPs ratio. The dominant memory access ratio is *low* with *high* frequency memory. Thus the CPU is predicted as the most optimal architecture as depicted in Figure 4.1.

NW, Sample Sizes	Single Node	Single Phi	Single-GPU
1024	0.349833	0.887091	1.32914
2048	0.441113	1.039292	1.42353
4096	0.846273	1.582725	1.729119
8192	2.241366	3.920679	2.734046
16384	8.211932	12.625132	6.834454

Table 6.7: Dynamic Programming: Needleman-Wunsch (NW) run-time (sec), varying data sizes

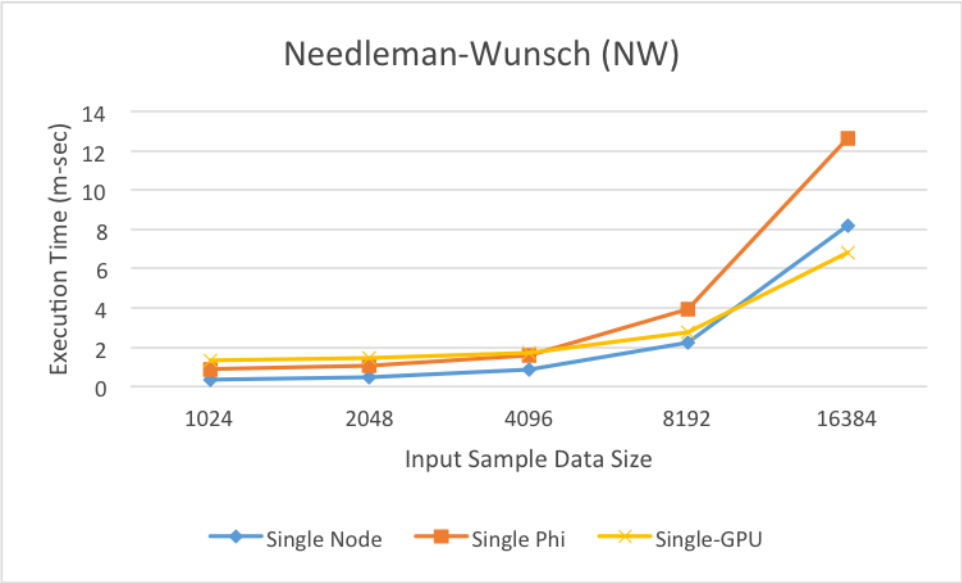

Figure 6.8: Dynamic Programming: Needleman-Wunsch (NW) results, varying data sizes

Prediction Verification - Table 6.7 and Figure 6.8, demonstrate that the CPU achieves the fastest run time except for the highest data size. Thus the prediction can be termed as mostly accurate.

8. **Graphical Models : Baum-Welch hidden Markov model (BW-HMM)** - The Hidden Markov Model (HMM) is a tool for representing probability distribution over sequence of observation in the form of a Bayesian network. The Baum-Welch algorithm employs a forward-backward approach to find the unknown parameters of an HMM model [45]. This algorithm has many applications in the areas of bio-

informatics, cryptanalysis and speech recognition.

Application Characterization - The time complexity for this parallel forward-backward algorithm is O($n$log$n$) and thus can be termed as *low*. Flops-to-Non-FLOPs ratio is *high*. The dominant memory access ratio is *low* with *high* memory access frequency.

Architecture Prediction - As per the ratings described in the application characteristics for this algorithm and Figure 4.1, the Xeon Phi is predicted as the architecture that aligns the best with the needs of the BW-HMM algorithm.

BW-HMM , Sample Sizes	Single Node	Single Phi	Single-GPU
250	1.19276	1.0022858	1.596556
500	2.56302	1.5832326	2.143096
1000	6.401591	2.188918	4.023842
2000	18.505463	3.1340927	11.10542
4000	56.569929	23.281245	43.842075

Table 6.8: Graphical Models : Baum-Welch hidden Markov model (BW-HMM) run-time (sec), varying data sizes

Prediction Verification - Table 6.8 and Figure 6.9 depicts that the Xeon Phi achieves the best run-time at every data size and scales better than the other architectures. This performance gain can be attributed to its bi-directional memory ring that allows it to guard compute performance against the negative effects of the bad memory access pattern with more frequent access .

9. **Combinational Logic: Cyclic Redundancy Check (CRC)** - CRC is based on cyclic error-correcting codes. A redundancy is introduced in the message for the sole purpose of checking the correctness of the data, i.e. to detect the data corruption while transmission. This redundant value is the remainder of polynomial division on the data stream with a predetermined CRC polynomial. At the receiving end, a zero-remainder signals loss-less transmission [46].

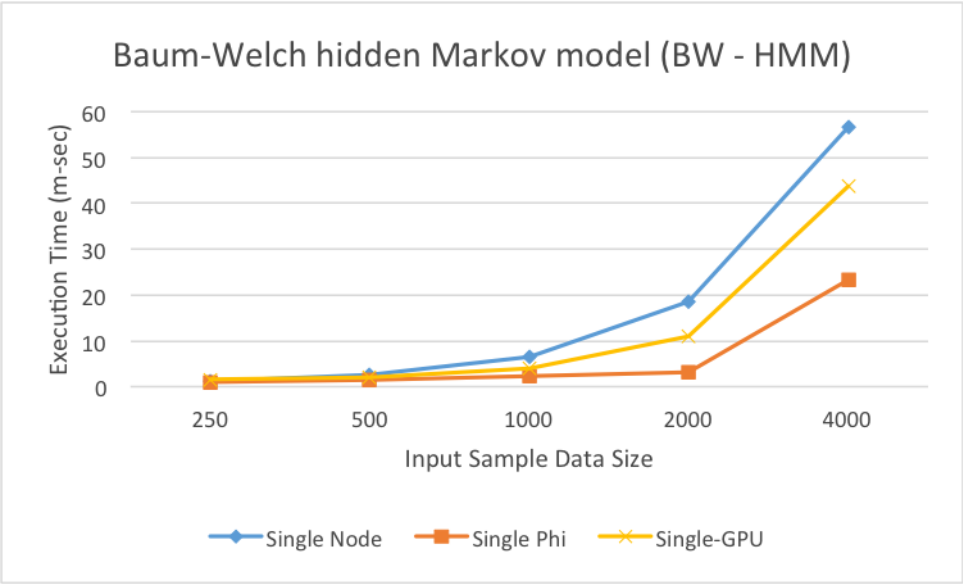

Figure 6.9: Graphical Models : Baum-Welch hidden Markov model (BW-HMM) results, varying data sizes

Application Characterization - The time complexity as well as FLOPs to non-FLOPs ratio is *low* for the CRC slice by 8 algorithm in this implementation. The dominant Memory access ratio is *high* with *lower* frequency of access.

Architecture Prediction - With *high* dominant memory access ratio with *low* ratings on other three parameters, as per mapping in Figure 4.1, the CPU is predicted as the most suitable architecture.

CRC, Sample Sizes	Single Node	Single Phi	Single-GPU
64	0.415	27.162	1.985
512	1.187	132.326	14.327
1024	2.271	283.636	28.4
2048	9.38	610.038	56.659

Table 6.9: Combinational Logic : Cyclic Redundancy Check (CRC) run-time (sec), varying data sizes

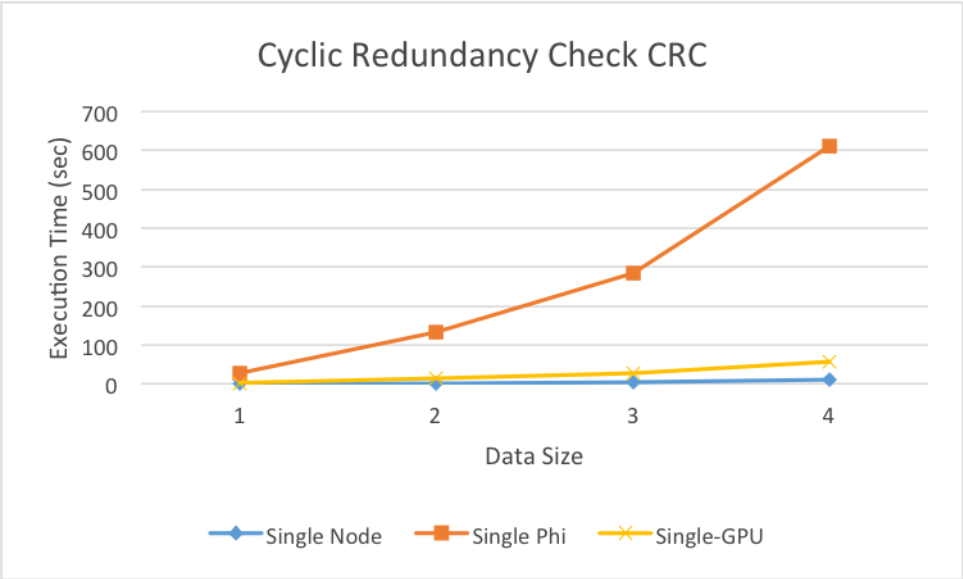

Figure 6.10: Combinational Logic : Cyclic Redundancy Check (CRC) results, varying data sizes

Prediction Verification - Table 6.9 as well as Figure 6.10 validates the prediction that the CPU achieves best performance for among all architectures.

Table 6.10 below summarizes the applications characterized in this study along with their prediction and the observed optimal architecture matches. As observed, optimal architectures for 7 out of 9 applications are predicted accurately, yielding 78% accuracy for the qualitative model proposed in Figure 4.1.

Rows 3 and 4 show that both of the inaccurate predictions incorrectly selected the Xeon Phi whereas the experimental results suggested the GP-GPU. This inaccuracy is due to two main reasons, (1) similarity of OpenCL with CUDA and (2) unfamiliarity with MIC programming techniques as compared to GP-GPU. Even though the application is developed in OpenCL which is a neutral language and it is very similar to CUDA, the developer's implementation has an unconscious bias that favors the GP-GPU architecture. (Note: the application developer in this case was the author of OpenDwarfs).

Rows 6 and 7 demonstrate that when two architectures achieve similar performance, the model favors the CPU over the GP-GPU. This is more logical as, the GP-GPU is an

accelerator and hence always needs a CPU to work with. For cost, availability and data transfer reasons, the CPU over the GP-GPU is a favourable choice unless CPU has other additional tasks.

No.	Algorithmic Domain	Application	Prediction	Observation	result
1	Dense Linear Algebra	K-Means Algorithm	GPU	GPU	✓
2	Sparse Linear Algebra	Sparse Matrix Vector multiply (SPMV)	GPU	GPU	✓
3	Spectral Methods	Fast Fourier Transform (FFT)	Xeon Phi	GPU	×
4	N-body Methods	GEM	Xeon Phi	GPU	×
5	Structured Grid	Speckle Reducing Anisotropic Diffusion(SRAD)	GPU	GPU	✓
6	Unstructured Grid	Computational Fluid Dynamics (CFD)	CPU	CPU, GPU	✓
7	Dynamic Programming	Needleman-Wunsch(NW)	CPU	CPU, GPU	✓
8	Graphical Models	Baum-Welch hidden Markov model (BW-HMM)	Xeon Phi	Xeon Phi	✓
9	Combinational Logic	Cyclic Redundancy Check (CRC)	CPU	CPU	✓

Table 6.10: Qualitative model verification: Summary of results for 9 tests/applications

6.3 Case Study Observations : LU Decomposition

In an MPI-only implementation, the rows of this combined LU matrix were distributed across processes using the similar Row Interleaved Decomposition method. Each process then helps solve $\mathbf{Ly} = \mathbf{b}$ to get the values of vector $\mathbf{y}$. These values are broadcasted so that at the end of this forward substitution stage, each process now has a copy of the entire $\mathbf{y}$ vector. The next stage involves the back substitution stage to solve $\mathbf{Ux} = \mathbf{y}$ to finally get the $\mathbf{x}$ vector. This method is a fairly naïve approach to achieve parallelism for given sub tasks. This approach does result in better performance as compared to the serial implementation owing to the fact that the rows were divided and thus simultaneously processed. Figure 6.11 demonstrates the scaling of the algorithm with increasing data sizes with the number of processes.

In the CUDA-only implementation, the entire original matrix $\mathbf{A}$ along with vector $\mathbf{b}$ is copied to the GP-GPU. A conversion kernel is invoked on this data to transform the matrix $\mathbf{A}$ into a combined $\mathbf{LU}$ matrix. A forward substitution kernel operates on this newly formed matrix and generates the necessary $\mathbf{y}$ vector. A backward substitution kernel is then

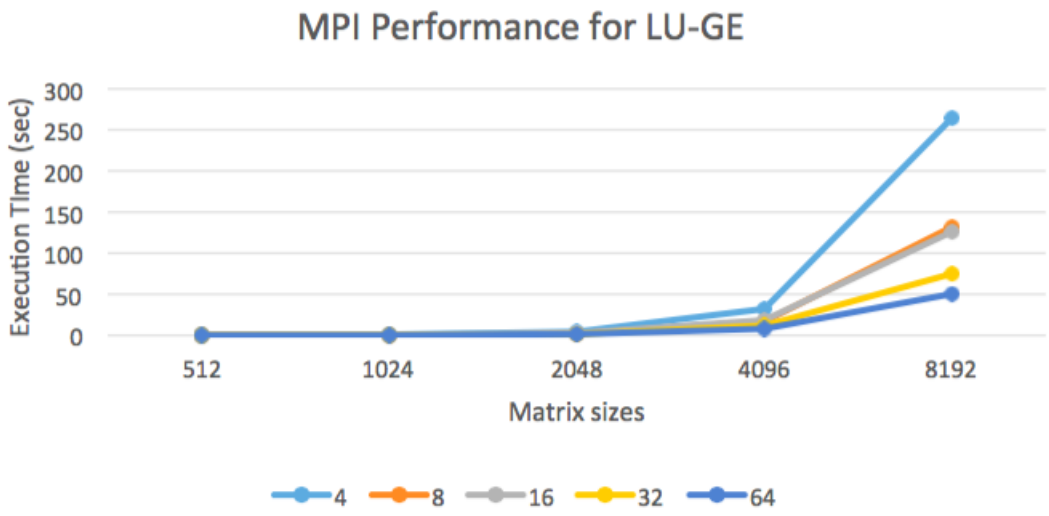

Figure 6.11: MPI Performance for LU

used to calculate the final results. As evident from this algorithm, there is a dependency among the kernels however operations within each individual kernel are highly parallelizable and thus we see a great benefit with reduced computation times for this implementation. Figure 6.12 shows the scaling of the algorithm with increasing data sizes as we vary the block sizes for the algorithm.

In a hybrid MPI+CUDA implementation, the aim is to leverage the heterogeneity and efficiently divide the work between the CPU and GP-GPU so as to achieve a balance and thus get a better performance. The design decisions are fairly intuitive as the possible combinations are limited. As per the structure of the chosen LU decomposition method, the two decoupled sections are the decomposition/transformation of **A** to combined **LU** form and the substitution to solve the equations. The decomposition or transformation part was handled by individual processes and the transformed matrix was copied to the GP-GPU memory row by row. Then the CUDA kernels work on the substitution part forward followed by back operation to achieve the results. The performance in hybrid implementation is mainly impaired by the fact that these stages must wait on the earlier stage to finish. It is important to note that achieving a good balance of both block size and number of processes is essential. The behavior of MPI-only code is different than that of

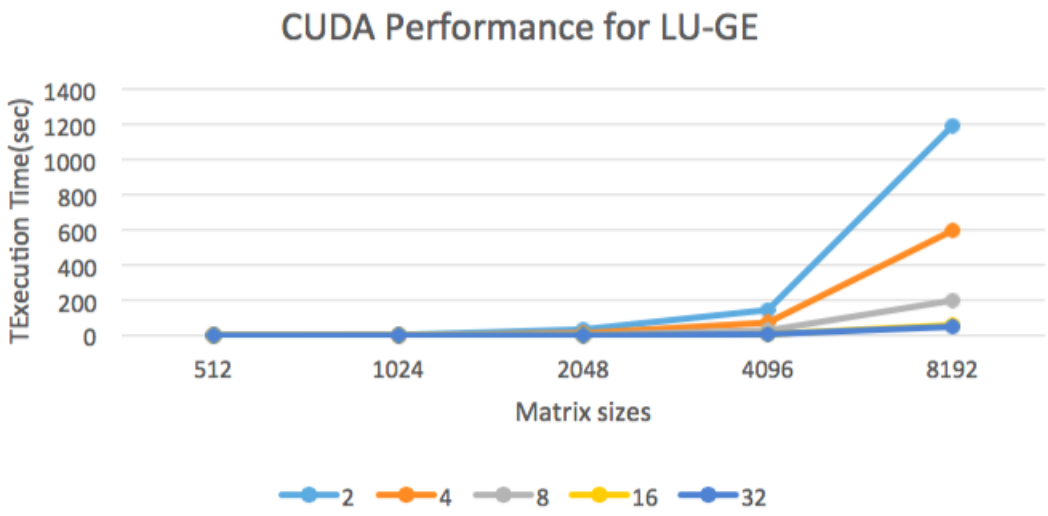

Figure 6.12: CUDA Performance for LU

the MPI part of hybrid code due to reduced computation.

Comparing the performances for all of the LU implementations as depicted in Figure 6.13, we see that CUDA performs the best i.e GP-GPU is the most optimal architecture. However, the MPI-only code is slower by just a minuscule scale. Thus Multicore CPU running MPI or symmetric configuration of CPU and a XEON PHI might have been a comparable optimal choice for this application. Although one would expect utilization of both CPU+GP-GPU concurrently should improve the performance when a CPU running MPI alone or a GP-GPU running CUDA alone achieves good speedup, the hybrid code that combines MPI and CUDA performed just as bad as the serial version. This degradation of performance corresponds to the added overheads of communication involved between CPU and the GP-GPU as well as the dependencies on each other. A better division of labor with larger data sizes might see some benefits from the hybrid approach, but is beyond the scope of this work.

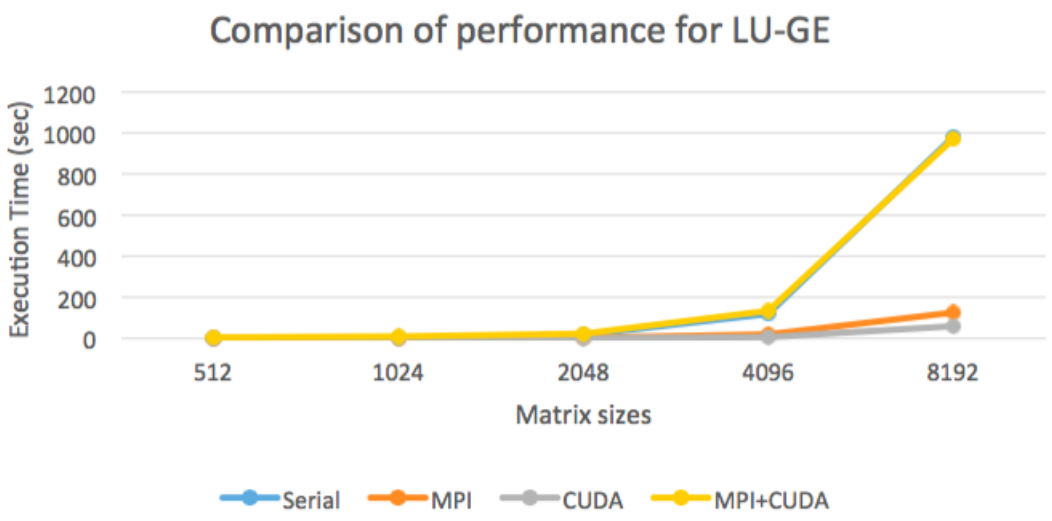

Figure 6.13: Comparison of Performance for LU Implementation

6.3.1 Trial and Error methods - parallelization efforts

As observed in the section 6.3 , the LU Decomposition application had three different implementations, MPI, CUDA and MPI+CUDA Hybrid. The best experimental results are for the 'CUDA 16 block size' implementation as noted in in Table 6.11.

Matrix Sizes	**512**	**1024**	**2048**	**4096**	**8192**
Serial Only	0.23	1.87	15.1633333	118.69	982.16
MPI 16 nodes	0.10211	0.438825	2.566645	18.362238	126.051864
CUDA 16 block size	0.033682495	0.12594397	0.839140686	6.440961914	58.81563672
Hybrid MPI + CUDA	4.749216	8.964449	22.677378	133.456077	970.861978

Table 6.11: LU Decomposition : Run Time comparison of implementations

It is observed that CUDA on the GP-GPU is the combination that achieved the best performance out of all those tried. It is important to note that there is no defined end point for this type of experimentation. The experiment is usually aborted by one of the following scenarios:

(a) Exhausted resources - hardware/software expertise

(b) Time deadline for completion

(c) Reasonable/acceptable amount of speedup achieved

Thus, trial and error methods do not necessarily guarantee best achievable performance, which is a major disadvantage of this approach. Specially for a scientists with limited HPC experience, the resource intensity of the trial and error methods can be daunting.

To quantify the parallelization efforts, the following assumptions are made:

1. Man hours for one code conversion is constant.
2. Cost of access any given type of hardware is constant
3. Efforts involved in acquiring the knowledge and expertise in a particular programming model for a given architecture type are constant.

The best achieved results are for the 'CUDA 16 block size' in Table 6.11. Here the run-times for the 8192 matrix size is selected for comparison. The parallelization efforts are summarized in Table 6.12. The man-hours for code conversion are counted as 1x for MPI, 1x for CUDA, and 0.5x for combining the two approaches in hybrid code. Similarly, the required amount of necessary accesses to hardware is quantified as 1x for multi-core CPU and 1x for GPU. The degree of knowledge and expertise required is also 2x for the programming models used, MPI and CUDA. The overall assessment of trial and error methods can be summarized as a summation of all the other factors. Thus this approach is 6.5x effort for achieving the run time of 58.815637 seconds for matrix size of 8192 x 8192 as shown in Table 6.12.

Best Run Time achieved	**58 seconds for 8192x8192 matrix**
Man-hours for code conversion	**2.5 x**
Amount of necessary H/W access	**2x**
Degree of knowledge and expertise required	**2x**

Table 6.12: Quantified parallelization efforts for trial and error methods

6.3.2 "Architecture First, Language Later" parallelization strategy - parallelization efforts

The main advantage of the "Architecture First, Language Later" parallelization strategy as explained in Section 4.2 is that one combination of architecture and programming model is selected based on the qualitative model's projection using the application/algorithm characterization. This selection lessens the code conversion efforts as one has to convert just once as opposed to multiple times in the trial and error approach.

The man hours for code conversion as well as degree of knowledge and expertise required is assessed to be 1x each since this approach involves parallelization of code into one programming model for one particular type of architecture e.g., CUDA code for the GP-GPU in case of the LU Decomposition algorithm. Since the GP-GPU acts as a co-processor, and it requires a host CPU, therefore the hardware factor is estimated as 1.25 x. The CPU is not counted as 1x since the CPU need not be a powerful multi-core CPU as in the case of considering a multi-core CPU as the only parallel architecture. The best run time achieved is considered to be the same as the CUDA implementation, i.e. 58.815637 seconds for a matrix size of 8192 x 8192. This determination of equivalent performance is based on an assumption that the program implementation is considered fully optimized. This size selection is a conscious decision made for the purpose of easier comparison between two approaches. It is possible to choose the investment in development efforts as a constant, where the remaining 3.25x can be considered to be an investment in fine tuning and optimization of the CUDA code, which will reduce the run-time further, achieving higher speedup as an outcome. As outlined in Table 6.13, the overall parallelization effort is summarized as 3.25x.

Best Run Time achieved	**58 seconds for 8192x8192 matrix**
Man-hours for code conversion	**1 x**
Amount of necessary H/W access	**1.25 x**
Degree of knowledge and expertise required	**1x**

Table 6.13: Quantified parallelization efforts for "Architecture First, Language Later" parallelization strategy

6.4 Summary Of Results and Analysis

To verify the programming model selection strategy of using the native language for the architecture of choice, a representative application (k- means algorithm) was executed on multiple architectures in the native language as well as in portable language(s) to see the effect of language variability for a given application. Results for the K-means algorithm with 1,638,400 items and 20 clusters demonstrated that a certain architecture yields fastest performance for native as well as portable implementation. It also exhibited that native language achieved superior computer performance as compared to the portable language.

In this chapter, the architecture to application mapping model used to select the appropriate architecture was also validated. The model was tested for 9 applications from OpenDwarfs suite consisting of different computation/communication idioms, i.e. dwarfs. The accuracy was validated to be 78 %, with 7 out of 9 applications yielding correct predictions. The 2 incorrect predictions were attributed to two main reasons: (1) similarity of OpenCL with CUDA and (2) unfamiliarity with MIC programming techniques as compared to GPU.

To establish the effectiveness of the parallelization strategy, a representative algorithm from scientific computing (Lower-Upper decomposition) was selected and approached with the two methodologies, trial and error methods as well as the "Architecture First, Language Later" parallelization strategy. The development efforts incurred are contrasted for both methods. As part of the trial and error approach, the algorithm was parallelized and implemented in MPI, CUDA and a hybrid version using MPI+CUDA. The trial and error development required 6.5x effort and achieved the best performance, i.e. 58 seconds for 8192x8192 matrix. The "Architecture First, Language Later" achieved comparable performance at 58 seconds with only 3.5x worth of development effort and thus was proven to achieve 50 % reduction in efforts to achieve similar performance.

The next chapter summarizes the research work and draws conclusions. This chapter also provides suggestions for future work.

Chapter 7

Conclusion and Future Work

7.1 Summary

This thesis presents a novel three step "Architecture First, Language Later" parallelization strategy to accelerate scientific applications, that maximizes the opportunities of achieving speedup while minimizing the development efforts. As mentioned in Chapter 1, development of this strategy was motivated by the limitations and wastage of resources and efforts incurred in trial and error, a methodology commonly employed by scientist when attempting to leverage High Performance Computing to achieve speedup.

Chapter 2 reviewed some of the prominent literature targeting performance modeling approaches. Additionally, it also discussed the trade offs between accuracy and generality, the idea of a fitness matching model and hybrid approach technique that combines qualitative and quantitative models into one. This thesis research extends the the hybrid approach of qualitative modeling for fitness and quantitative modeling for selecting programming methodology, and enhances its generality and robustness while maintaining reasonable accuracy. A few architecture studies that demonstrate that a specific type of processor or co-processor/accelerator is more appropriate for certain type of applications with unique set of computation and communication patterns, were also elaborated.

Chapter 3 provided the necessary background for this thesis including the details

of a few key concepts in parallel programming such as Flynn's Taxonomy, Shared Vs. Distributed Memory and Manycore Vs. Multicore processors. This chapter also introduced the computing accelerators namely, General Purpose Graphics Processing Unit (GP-GPU) and Many Integrated Core (MIC) architecture. Additionally, various programming models such as MPI, CUDA, OpenCL were detailed. Benefits and costs of Code Portability was also analyzed in the 3rd Chapter.

Chapter 4 detailed the experimental design and the claims regarding the effectiveness of the "Architecture First, Language Later" parallelization strategy. A qualitative model used for application-to-architecture mapping was explained in detail. This model is based on four parameters namely, Time Complexity, FLOPs to Non-FLOPs ratio, Dominant Memory Access Ratio and Memory Access Frequency. The chapter also outlines the programming model selection using a quantitative model based on experimental benchmarks. This chapter also outlines the validation metrics used to determine the effectiveness of the parallelization strategy proposed in this thesis.

In Chapter 5, the mathematical concepts involved in the Lower Upper Decomposition algorithm as an enhanced Gaussian Elimination technique to solve a system of linear equations was explained along with a numerical example including solution. This chapter also includes guidelines for implementing Trial and Error methods as well as the "Architecture First, Language Later" parallelization strategy.

As mentioned in Chapter 6, to verify the programming model selection strategy of using the native language for the architecture of choice, a representative application (k-means algorithm) was executed on multiple architectures in the native language as well as in portable language(s) to see the effect of language variability for a given application. Results for the K-means algorithm with 1,638,400 items and 20 clusters demonstrates two key things:

1. On a given architecture, the native language achieves the fastest run-time, i.e. CUDA on GPU implementation takes less time than OpenCL on GPU

2. For a given application, a specific architecture achieves better performance for the same portable implementation, i.e. OpenCL implementation runs fastest on GPU as compared to CPU and Xeon Phi.

Chapter 6 also validated the architecture to application mapping model used to select the appropriate architecture. The model was tested for applications from a benchmark suite consisting of different computation/communication idioms, i.e. dwarfs called OpenDwarfs. The prediction was then verified by comparing actual run-time by running the code on different architectures. As observed, the accuracy was validated to be 78 %, with 7 out of 9 applications yielding correct predictions. The 2 incorrect predictions were attributed to two main reasons: (1) similarity of OpenCL with CUDA and (2) unfamiliarity with MIC programming techniques as compared to GPU. Thus this can be further improved by better understanding the Xeon Phi architecture in more depth in order to achieve better alignment of software structure with hardware blocks and to develop a less biased portable program with OpenCL despite its similarities with CUDA.

To establish the effectiveness of the parallelization strategy, a representative algorithm from scientific computing (Lower-Upper decomposition) was selected and approached with the two methodologies, trial and error methods as well as the "Architecture First, Language Later" parallelization strategy. As part of the trial and error approach, the algorithm was parallelized and implemented in MPI, CUDA and a hybrid version using MPI+CUDA. The trial and error development required 6.5x effort and achieved the best performance, i.e. 58 seconds for 8192x8192 matrix. The "Architecture First, Language Later" achieved comparable performance at 58 seconds with only 3.5x worth of development effort and thus was proven to achieve 50 % reduction in efforts to achieve similar performance.

7.2 Conclusion

We draw the following conclusions:

1. For any given scientific application, one architecture performs better than other avail-

able architectures, even for a common portable application.

2. For a given architecture, the same algorithm has varying run-times for implementations in different languages.

3. For a given algorithm coupled with a given architecture, the implementation in its native language generally achieves the best performance.

4. A qualitative model used in this thesis predicts an optimal architecture for a given application from scientific domain with 78 % accuracy.

5. The proposed "Architecture First, Language Later" parallelization strategy achieves 50 % reduction in development efforts while achieving results comparable to that of the trial and error methods.

Thus, it is concluded that the proposed "Architecture First, Language Later" parallelization strategy achieves best performance with least development efforts with the help of a hybrid combination of quantitative model for programming model/language selection coupled with a qualitative model for optimal architecture prediction. Hence, the proposed strategy is a performance focused, development friendly and model aided parallelization strategy.

7.3 Future Work

This thesis presents a parallelization strategy to accelerate scientific applications that maximizes the opportunity to achieve speedup while minimizing the development efforts. This strategy has been validated using one end-to-end case study scenario and 9 application tests. Additional case studies with other scientific algorithms and full applications should be carried out employing the "Architecture First, Language Later" strategy, which will help understand the applicability for a wider domain. Such observations can also be helpful in designing iterations for the underlying application to architecture model.

The qualitative model used in this thesis, can be further extended to include more parallel architectures such as FPGAs, in addition to the three architectures - CPU, GPU and Xeon Phi. Additionally, run-times should be measured again on the second generation Xeon Phi known as Knights Landing and the more advanced version of NVIDIA GPU Tesla K40. These results will help understand the differences between the capacities of two architectures. The architectures like CPU and Xeon Phi, can run MPI as well OpenMP, and both can be considered native. The current model can be further enhanced to include characterization for multiple "native" programming languages in such scenarios.

The proposed "Architecture First, Language Later" strategy is based on a model with 78 % accuracy. As explained in the summary and in Chapter 6, the accuracy can be further improved by studying the Xeon Phi architecture in more depth an fine tuning the model to take this enhanced knowledge into consideration.

This thesis pioneers the way for qualitative characterization of application, which when mapped to hardware capabilities, yields optimal match and thus best performance. A quantitative characterization and application to architecture mapping would be the next logical step in the right direction necessary to further this research. The application characterization can be automated by developing a code scanner which could take away the burden on the user to characterize and thus make it more user friendly. Such tool would allow the naive users to adapt this parallelization strategy with ease.

Additionally, this thesis involved a hybrid approach of using MPI + CUDA implementation of LU Decomposition to utilize CPU as well as accelerator GP-GPU. This implementation was motivated by a line of thought that utilizing host CPU for not only control but for compute work means more compute resources and thus likely will result in an increase in performance. However, the hybrid code did not achieve the desired gain in performance. This unexpected result is attributed to the fact that data communication slows the code down. There are no defined patterns of data or task decompositions for the hybrid approach. The hybrid approach needs to be further investigated in order to unlock the possible benefits of utilizing CPU along with the accelerator for computation.

Bibliography

[1] K. Krommydas, W.-c. Feng, C. D. Antonopoulos, and N. Bellas, "Opendwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures," *Journal of Signal Processing Systems*, pp. 1–20, 2015.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[3] P. Colella, "Defining software requirements for scientific computing," 2004.

[4] What is gpu computing? [Online]. Available: http://www.nvidia.com/object/what-is-gpu-computing.html

[5] NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210. [Online]. Available: http://images.nvidia.com/content/pdf/tesla/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

[6] J. Reinders, "An overview of programming for intel xeon processors and intel xeon phi coprocessors," 2012.

[7] K. Sapra, "Framework for lifecycle enrichment of hpc applications on exascale heterogeneous architecture."

[8] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso, "A survey of performance modeling and simulation techniques for accelerator-based computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 1, pp. 272–281, 2015.

[9] V. V. Pallipuram Krishnamani, "Exploring multiple levels of performance modeling for heterogeneous systems," Ph.D. dissertation, Clemson University, 2013.

[10] M. Bhuiyan, "Performance analysis and fitness of gpgpu and multicore architectures for scientific applications," Ph.D. dissertation, Clemson University, 2011.

[11] E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Performance evaluation of sparse matrix multiplication kernels on intel xeon phi," in *Parallel Processing and Applied Mathematics*. Springer, 2013, pp. 559–570.

[12] A. Abdullah, K. kok Yong, E. K. Karuppiah, and P. K. Chong, "Multi keyword range search in gpu and mic: A comparison study," in *Open Systems (ICOS), 2014 IEEE Conference on*. IEEE, 2014, pp. 117–122.

[13] R. Bordawekar, U. Bondhugula, and R. Rao, "Believe it or not!: mult-core cpus can match gpu performance for a flop-intensive application!" in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 537–538.

[14] G. Teodoro, T. Kurc, J. Kong, L. Cooper, and J. Saltz, "Comparative performance analysis of intel xeon phi, gpu, and cpu," *arXiv preprint arXiv:1311.0378*, 2013.

[15] M. Snir, *MPI–the Complete Reference: The MPI core*. MIT press, 1998, vol. 1.

[16] L. Dagum and R. Enon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[17] NVIDIA Corporation, *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2015.

[18] K. O. W. Group *et al.*, "The opencl specification, version 1.2, 15 november 2011," *Cited on pages*, vol. 18, no. 7, p. 30.

[19] M. Wolfe, "The openacc application programming interface," 2013.

[20] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson, "Comparing hardware accelerators in scientific applications: A case study," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 58–68, 2011.

[21] R. Dolbeau, F. Bodin, and G. C. de Verdiere, "One opencl to rule them all?" in *Multi-/Many-core Computing Systems (MuCoCoS), 2013 IEEE 6th International Workshop on*. IEEE, 2013, pp. 1–6.

[22] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of cuda and opencl," *arXiv preprint arXiv:1005.2581*, 2010.

[23] A. J. Van der Steen, *Overview of recent supercomputers*. Citeseer.

[24] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE transactions on computers*, vol. 100, no. 9, pp. 948–960, 1972.

[25] M. Ilg, J. Rogers, and M. Costello, "Projectile monte-carlo trajectory analysis using a graphics processing unit," in *2011 AIAA Atmospheric Flight Mechanics Conference, Portland, OR, Aug*, 2011, pp. 7–10.

[26] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[27] B. Barney, "Introduction to parallel computing. lawrence livermore national laboratory," *Available on https://computing. llnl. gov/tutorials/parallel_comp*, 2012.

[28] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[29] S. McGinn and R. E. Shaw, "Parallel gaussian elimination using openmp and mpi," in *null*. IEEE, 2002, p. 169.

[30] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron, "Rodinia: Accelerating compute-intensive applications with accelerators." IISWC, 2009.

[31] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[32] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' guide*. Siam, 1999, vol. 9.

[33] J. Phillips, *The NAG library*. Clarendon Press, 1986.

[34] M. Intel, "Intel math kernel library," 2007.

[35] M. U. Guide, "The mathworks," *Inc., Natick, MA*, vol. 5, p. 333, 1998.

[36] J. Lambers, "Summer session 2009-10 lecture 4 notes mat610," Lecture Notes, 2009-10.

[37] J. F. Grcar, "Mathematicians of gaussian elimination," *Notices of the AMS*, vol. 58, no. 6, pp. 782–792, 2011.

[38] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 24, no. 7, pp. 881–892, 2002.

[39] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills, "Vectorized sparse matrix multiply for compressed row storage format," in *International Conference on Computational Science*. Springer, 2005, pp. 99–106.

[40] W.-c. Feng, H. Lin, T. Scogland, and J. Zhang, "Opencl and the 13 dwarfs: a work in progress," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ACM, 2012, pp. 291–294.

[41] Y. Yu and S. T. Acton, "Speckle reducing anisotropic diffusion," *IEEE Transactions on image processing*, vol. 11, no. 11, pp. 1260–1270, 2002.

[42] A. Corrigan, F. Camelli, R. Löhner, and J. Wallin, "Running unstructured grid based cfd solvers on modern graphics hardware," *AIAA paper*, vol. 4001, p. 2009, 2009.

[43] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," in *Application Specific Processors, 2008. SASP 2008. Symposium on.* IEEE, 2008, pp. 101–107.

[44] M. Fatimah Noni, "Reducing the search space and time complexity of needleman-wunsch algorithm (global alignment) and smith waterman algorithm (local alignment) for dna sequence alignment," Ph.D. dissertation, Universiti Malaysia Perlis (UniMAP), 2011.

[45] L. R. Welch, "Hidden markov models and the baum-welch algorithm," *IEEE Information Theory Society Newsletter*, vol. 53, no. 4, pp. 10–13, 2003.

[46] G. Campobello, G. Patane, and M. Russo, "Parallel crc realization," *IEEE Transactions on Computers*, vol. 52, no. 10, pp. 1312–1319, 2003.