2015-01-01

A Case Study Of Accelerator Performance

Esthela Gallardo
University of Texas at El Paso, egallardo5@miners.utep.edu

A CASE STUDY OF ACCELERATOR PERFORMANCE

ESTHELA GALLARDO

Department of Computer Science

APPROVED:

Patricia J. Teller, Ph.D., Chair

Michael P. McGarry, Ph.D.

Shirley Moore, Ph.D.

Charles Ambler, Ph.D.
Dean of the Graduate School

A CASE STUDY OF ACCELERATOR PERFORMANCE

by

Esthela Gallardo, B.S.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

December 2015

Acknowledgements

This work was partially funded by the National Science Foundation Stampede Grant through the University of Texas at Austin Grant #UTA13-000072, and by the Army High Performance Computing Research Center (AHPCRC). The University acknowledges the Texas Advanced Computing Center (TACC) at the University of Texas, Lawrence Livermore National Laboratory (LLNL), and the University of Tennessee, Knoxville for providing HPC resources that have contributed to the research results reported within this thesis.

Abstract

In recent years the designs of High Performance Computing (HPC) clusters have become more complex. This is due to the emergence of new processing elements, in particular Graphics Processing Units (GPUs) and other many-core processors that can be combined with multi-core processors to enhance application performance. The design of a cluster includes processing elements that meet the needs of the applications that will run on the system. Unfortunately, it has become increasingly difficult to compare the performance of novel many-core processing elements due to the differences in their architectures. This work describes an attempt to develop a methodology for comparing the architectures of three many-core processing elements, which are called accelerators, that are used in several existing HPC systems, i.e., the Fermi, Kepler, and MIC architectures. Using the LULESH 1.0 proxy application, which has been ported to processing elements with different architectures and programming models, we compared the number of instructions executed per cycle (IPC), memory behavior, vectorization capacity, and power energy consumption of these three architectures, as well as of the multi-core Sandy Bridge processor, the performance of which was used as a baseline for comparison. This study showed that (1) the Kepler architecture achieved the best execution-time performance, while consuming the least power/energy; and the Kepler's superior execution-time performance is due to LULESH's vectorization usage and high IPCs.

Table of Contents

List of Tables

List of Figures

List of Illustrations

Chapter 1: Introduction

The value of a supercomputer depends on the problems that are solved on it. Over the years, there has been a turnover of the technologies, architectures, and usage of these systems. The beginning of modern supercomputing was marked by the introduction of the vector computer in the 1970s. These systems offered a performance advantage by being able to compute multiple arithmetic operations simultaneously on elements of arrays. By the following decade, vector systems were integrated in conventional computing environments. Performance was subsequently enhanced by the introduction of shared-memory multiprocessors, which consist of multiple processors that share the same memory and have equal access to input/output devices. By the end of the 1980s scalable parallel computing was achieved through the use of distributed memory. Distributed-memory systems consist of multiple interconnected computers. It is through these systems that supercomputers were realized [1].

While the parallelism and scalability of a supercomputer can be expanded by simply adding more processors, there exist considerable drawbacks to this approach. These systems are very expensive to maintain due to the amount of power they consume. The expandability of these systems is also limited by the physical space they require. Finally, the processors that comprise these systems lose their value over time, as the needs of applications require more computing power than is achievable by these systems. Although processors on a system can also be upgraded, interest arose in the use of graphics processing units (GPUs) to aid in computation due to the limited parallelism of conventional superscalar processors. A GPU is a computer chip with a highly parallel architecture consisting of multiple simple cores that are designed for handling multiple tasks simultaneously. Consequently, GPUs are much more effective than general-purpose processors at performing single operations on large blocks of data. However, the architecture of a GPU is vastly different than that of general-purpose processor, and they are difficult to program.

In order to alleviate the issues associated with GPU programming, NVIDIA introduced the CUDA API. The API was the industry's first effective programming model that followed a C-

based development environment. Additionally, NVIDIA introduced the Fermi architecture in 2010, which was designed for general-purpose computing. Drawbacks of the Fermi architecture were addressed in the Kepler architecture, which was introduced in 2012. Specifically, the Kepler architecture was designed to improve performance, but at the same time be power-efficient.

Despite the fact that exploiting GPU architectures is beneficial for algorithms that process large amounts of data and that efforts were made to improve the programmability of these devices, there were still drawbacks to GPU programming. Even though a familiar programming model was adopted, one must understand the constructs in the API and the design of the GPU architecture to make appropriate use of the device's available parallelism. Additionally, there is a significant overhead associated with offloading data to the GPU.

Due to the increased use of GPUs in HPC clusters, Intel introduced the Many Integrated Core (MIC) architecture with the Xeon Phi co-processor. The MIC architecture is designed to be similar to a stand-alone processor, but with a large number of simpler cores. Furthermore, it supports programming models commonly used in parallel computing. Finally, it is able to launch programs independently, as opposed to a GPU, which requires a host processor.

Because of the emergence of these new architectures, commonly called accelerators, HPC systems are becoming increasingly complex. Designing these systems is not simple due to the fact that the performance of these accelerators are not directly comparable, and it is unlikely that a particular configuration will meet the needs of all the applications that will run on a system. As a developer, it is also difficult to determine whether an application should be launched solely on a general-purpose processor or should make use of an accelerator. Although various tools exist to measure the performance of an application on a given processor, it is not clear which metrics can be used to compare performance across architectures.

In order to address this concern, the LULESH proxy application was developed by the Lawrence Livermore National Laboratory (LLNL). LULESH is a mini-app that has been ported to several different programming models in order to identify optimization techniques that are portable across the models. Additionally, LULESH contains algorithms commonly used in

hydrodynamics, which is modeled in a variety of computer simulations of science and engineering problems. As a result, lessons learned from LULESH regarding its performance can be applied to larger applications that make use of the algorithms that are found in LULESH. Hence, it is an ideal candidate to compare the performance of different architectures.

However, the issue remains that there is no standard way to compare different architectures. Several tools do exist to evaluate the performance of an application, but the differences between the exposed metrics of distinct computing components make it difficult to compare them. Furthermore, different tools may specialize in providing different types of measurements and metrics, and the number of measurements that can be collected is based on the hardware of the architecture of interest. This work presents an attempt to develop a methodology for comparing commonly used accelerators. Specifically, this work compares the performance of the Xeon Phi co-processor and Kepler and Fermi GPUs through the use of the LULESH application and a number of performance tools. A general-purpose processor based on the Sandy Bridge architecture is used as a baseline for performance.

The organization of this thesis is as follows: Chapter 2 describes work that has been previously done to compare the performance of an application on different architectures. Chapter 3 describes the metrics used to compare application performance across accelerators and the systems used for experimentation, and provides details concerning the organization of the LULESH application. Chapter 4 describes the performance of LULESH on the architectures of interest and provides the performance data utilized to evaluate its runtime behavior. Finally, Chapter 5 summarizes the work and presents the conclusions of this study.

Chapter 2: Related Research

As new computing devices materialize, it is not uncommon for a variety of studies to emerge that exploit the features offered by the novel hardware and architectural designs. Considering that two of the architectures explored in this study, the Kepler and the MIC, are relatively new to the High Performance Computing (HPC) community, there already exist several case studies that explore the performance improvements that can be obtained from employing these devices. In addition, efforts have been made to compare these architectures with each other and with other computing devices to provide information regarding the performance advantages and disadvantages of each. Moreover, LULESH, the application used in this comparative study, has previously been used to analyze traditional and emerging parallel programming models.

This chapter reviews literature that is related to this study and compares the approaches used to evaluate application/architecture performance with the one used in this study. The chapter is partitioned into three sections, which reflect the main foci of this study. The first, Section 2.1, presents work that concentrates on providing information about the benefits of porting applications to specific computer architectures. Section 2.2 briefly describes performance evaluation techniques that have been applied to compare different architectures. And, finally, Section 2.3 discusses how LULESH has been used by Lawrence Livermore National Laboratory to compare programming models.

2.1 Performance Evaluation: Architecture-specific Analysis

The evaluation of the performance improvements that can be gained from porting an application to a specific computer architecture has been approached in many different ways. Some researchers focus on a new architecture and present a case study that involves the use of the architecture to achieve execution-time speedup. This is the case for the work described in [2], where Rosales, et al. explore several aspects of the Intel Xeon Phi co-processor, which utilizes the MIC architecture. An application can be executed using the Xeon Phi in three different ways: (1) Native mode: The application is executed solely on the Xeon Phi. (2) Offload mode: The

application is executed by both the Xeon Phi and a host processor, which executes parts of the application but does not execute concurrently with the Xeon Phi, which also executes parts of the application; usually, in this mode the CPU sends data to the Xeon Phi. (3) Symmetric mode: The CPU and Xeon Phi concurrently execute parts of the application. In [2], the authors present an analysis of these modes of execution to illustrate the challenges inherent in porting applications to this architecture, and techniques that can be applied to enhance performance. Execution time is the main metric used to measure the speedup gained by porting to the Xeon Phi from an Intel Xeon processor. The efforts to enhance application performance, which were related to vectorizing the code, improving the parallelism of the most time-consuming functions, and reducing the data movement between the Xeon Phi and Xeon processors, can be applied to other applications that are being tuned for the Xeon Phi. However, a deeper analysis is required to understand why those changes in the application resulted in a faster execution time.

While case studies provide valuable insights into the speedups that can be attained by executing a real application on a specific architecture, the algorithms explored are limited to the applications studied. In contrast, benchmarks can be used to stress different parts of a system. In particular, the Rodinia benchmarks were created, with this purpose in mind, for accelerators. These benchmarks are partitioned into mini-applications that allow testing of different algorithm domains. In [3] Che, et al. demonstrate the behavior of the benchmarks executed on an NVIDIA GeForce GTX 280 GPU and compare this behavior with that experienced on a Quad-core Intel Core 2 Extreme CPU. The devices are compared in terms of power consumption, synchronization, execution time, and communication overhead (for data transfers between the GPU and CPU). The CPU experiments were performed using one thread and four threads to determine the differences between serial CPU versions of the benchmarks, parallel CPU versions of the benchmarks, and parallel GPU versions of the benchmarks. Porting the benchmarks to the GPU resulted in significant speedups for each of the benchmarks, which ranged from a little over 5x as compared to the serial CPU K-Means benchmark up to 80x as compared to the serial CPU Leukocyte benchmark, and from 1.6x to 26.3x for the parallel CPU versions of the same benchmarks. Despite

the fact that the GPU provided a considerable performance improvement as compared to the serial code executed on the CPU, for the Similarity Scores benchmark the parallel GPU version consumed up to 1.3x more power than the serial CPU version and for the Leukocyte benchmark the parallel CPU version consumed up to 2x more power than the serial CPU version. Since neither device provided the lowest power consumption for all of the benchmarks that were evaluated, one cannot say that either the CPU or GPU is more power efficient than the other. For example, although both the parallel GPU and CPU versions of the Needleman-Wunsch benchmark consumed the same amount of power, the parallel GPU version of Leukocyte consumed 1.286x more power than the parallel CPU version. However, the CPU consumed 1.182x more power than the GPU for the parallel K-Means benchmark. The main drawback of the CPU was the overhead related to parallelization, which was as high as 30% for four of the benchmarks. In contrast, the GPU's performance was heavily affected by the time related to memory transfers.

The challenges associated with porting the benchmarks to accelerators are also discussed in [3]. The authors note that performance is more dependent on the implementation, as opposed to the architecture. Also, due to the architecture's design, optimizations are not intuitive and those employed in one application may not provide performance improvements in another. Recently, the Rodinia benchmarks were also used to evaluate the performance of the Xeon Phi in [4]. Unlike the previous study, the benchmarks were not tuned for this architecture, but nonetheless a speedup was achieved. Both offload and native modes were employed, and it was determined that the Rodinia benchmarks scaled better when native mode was used. The time for transferring data between the CPU and the Phi was found to be very expensive. The study suggests that optimization of the benchmarks that refine the memory behavior of an application and vectorize the code could result in greater performance and an improvement in the scalability of the codes.

In order to get an isolated view of particular aspects of performance, microbenchmarks may need to be applied. This approach was implemented by Jonson, Playne, and Hawick in [5] to obtain various performance tradeoffs of different GPUs. The popularity of and improvements to GPUs has rapidly increased over the last 10 years. In order to understand the range of performance

across different NVIDIA GPU models (from the GeForce GTX 260 to the GeForce GTX 680, and from the Tesla M2050 to the Tesla M2090), the devices are compared in terms of their integer and double-precision floating-point computation and global memory accesses (both random and coalesced accesses). This was achieved by writing simple microbenchmarks that stress these factors independently. Regarding the integer computation rate, the number of cores available on the device seems to be the deciding factor for the best achievable performance, followed by the clock speed of the cores. As a result, the GeForce GTX 680 (which follows the Kepler architecture) had the best integer computation performance. However, this GPU did not perform as well for double-precision floating-point computations. The GTX 680 had the lowest double-precision floating-point rate, followed by the GeForce GTX 200 series. The Tesla cards, which were designed for general-purpose processing, had the best double-precision floating-point computation rates. The GTX 680 also showed negative results for random memory accesses. It is suspected that this is due to the fact that the GTX 680 has a smaller number of multiprocessors than its predecessors (GeForce GTX 590, 580, and 480), which are responsible for handling memory operations for the cores. Nevertheless, the kernel execution time with random memory accesses was 1.35x, 1.43x, and 1.52x faster on the GeForce GTX 680 than on the Tesla M2090, Tesla M2050, and Tesla M2075, respectively. Regarding, coalesced memory accesses, the GeForce GTX 680 exhibited the best performance, while the Tesla cards exhibited the worst. Additionally, it was noted that while the GFLOPSs per GPU has been increasing, the computational power per core has decreased.

2.2 Comparison of Architectures

Although several studies in the literature focus on a detailed analysis of the performance of an accelerator using a standard CPU processor as a baseline for performance, there are not as many studies that compare the performance of different accelerators and processors. Additionally, due to the differences in the designs of accelerators and multi-core processors, a methodology to compare the performance of these devices has not been standardized.

Carabaño, et al. address this issue in a conceptual manner [6]. In [6], the authors list technical features of the i7 Sandy Bridge, Xeon Sandy Bridge, GeForce GTX 680 GPU, Tesla K20x GPU (which employs the Kepler architecture), Xeon Phi, and a standard FPGA. Based on the known features and drawbacks of each device, the achievable productivity, energy efficiency, and performance for scalar-complex and parallel-simple codes were estimated. In this study, productivity is not a quantifiable metric, since it is defined in terms of the complexity of employing a specific type of processor with respect to how much a developer must be aware of the underlying architecture in the device, and if a high-level programming language with a sophisticated compiler can be used to develop applications to run on the device. The CPUs were predicted to have the best productivity overall and the best performance with scalar-complex codes due to the fact that they have high clock rates, complex processing units, and sophisticated memory hierarchies. The GPUs were predicted to have the best performance with parallel-simple codes; this predication was based on the argument that GPUs are designed with these applications in mind. FPGAs were predicted to have the highest energy efficiency since they are designed to have low power consumption. The Xeon Phi was not determined to outperform in any of the categories. However, the Xeon Phi was predicted to be a close successor of the CPUs in term of productivity and performance achieved with scalar-complex codes. Although this work can serve as a quick reference for the design features of the devices included in the study, it lacks experimental data to support the predictions – the arguments are qualitative rather than quantitative in nature.

A stronger comparison between architectures is found in [7], where the Xeon Phi is compared to a Sandy Bridge Xeon E5-2620 processor and the Tesla c2050 GPU (which employs the Fermi architecture). The SHOC benchmarks are used to compare the Xeon Phi with the Tesla in terms of power consumption and execution time, and the Rodinia benchmarks are used to compare the Xeon Phi to the Sandy Bridge processor in terms of execution time. Additionally, native mode was used to compare the Xeon Phi with the Sandy Bridge processor and offload mode was used to compare the Phi with the Tesla GPU. The Sandy Bridge processor outperformed the Xeon Phi for memory-bound benchmarks, but the Phi fared better for the compute-bound

benchmarks. However, the Xeon Phi exhibited better performance than the Tesla GPU for both the compute-bound and memory-bound benchmarks. Nonetheless, it generally consumed more power than the GPU. The Xeon Phi also was analyzed independently to show that its performance and power characteristics vary with the computation and the memory characteristics of an application. Specifically, the SHOC and Rodinia benchmarks were run using 2 to 240 cores to determine how well the application scaled to the available threads on the Xeon Phi. Note that these experiments were not accompanied by changes to the default affinity setting. The benchmarks varied in their memory-to-computation ratio. The majority of the benchmarks were found to not scale well after 120 threads due to a sudden increase in L2-cache load misses. However, some of the benchmarks experienced a reduction in L2-cache load misses as the threads were increased from 2 to 120 threads, while others exhibited a constant number of cache misses up to 80 threads, followed by a significant drop once 120 threads were reached, and a sudden increase afterwards. The energy consumption of the benchmarks followed a similar pattern to that of execution time, with the amount of energy consumed decreasing as the number of threads were increased up to 120 threads. However, the distribution of the energy consumed between data transfers and computation varied with each benchmark. Even though both the Reduction and FFT benchmarks are memory bound, for the FFT benchmark 80% of energy consumption was due to data transfers, while the Reduction benchmark devoted only 10% of its energy consumption data transfers. Similar experiments to establish the scalability of the algorithms on the Fermi and Xeon processor were not performed.

Another property that has been used to compare different processing units is programmability, specifically, a quantification of the complexity of using the programming model associated with the device of interest. Determining the complexity of employing any given device is a difficult task since ease-of-use is not a quantifiable metric. This is due to the fact that usability is subjective. Despite this disadvantage, Krommydas, Scogland, and Feng attempt to assess the programmability of the Kepler architecture (represented by the Kepler K20c) and the MIC architecture (represented by the Xeon Phi) in [8]. Programmability of the Sandy Bridge

architecture (represented by the Xeon E5-2680) is used as a baseline. To perform this assessment, an n-body molecular modeling application was ported from its serial version to a parallel version using OpenMP – this code was targeted for the Sandy Bridge and Xeon Phi. OpenACC and CUDA were both utilized to parallelize the application for execution on the Kepler GPU. Programmability was measured by counting the number of source lines of code introduced to develop each of parallel code. Additionally, common optimization techniques were applied and their effect on application performance was measured by calculating the speedup over the serial code. The highest speedup, 1,020.88x, was achieved through the use of the Kepler, followed by the Xeon Phi with an 885.18x speedup. However, the Kepler required much more manual tuning than did the Xeon Phi. This was because many of the optimizations applied to the code for the Sandy Bridge processor were found to be effective on the Xeon Phi. Despite the tuning applied to the code for the Kepler, it reached only 54.34% of its theoretical peak performance. At 86.35%, the Xeon Phi achieved the highest performance relative to its theoretical peak.

The case studies discussed above mainly consider a single device. However, for large-scale applications, it is common for the workload to be distributed over multiple devices to improve execution-time performance. Considering this usage, in [9], Bernaschi, Bisson, and Salvadore evaluate the performance of GPU and Xeon Phi (MIC) architectures not only as stand-alone systems, but also in a cluster environment. Parallel efficiency and execution time are the main metrics considered in this work. (Parallel efficiency is defined as the serial execution time of the application divided by the product of the number of parallel devices used, *P*, and the execution time of the application running on *P* devices.) The performance of the Intel Xeon E5-2687 processor was used as a baseline. The Kepler K20 accelerator and the Xeon Phi 5110P co-processor were included in this work to compare the GPU and MIC architectures. In order to fully parallelize the application on the CPU and Xeon Phi, vectorization had to be incorporated into the application, along with thread parallelism. The Xeon Phi required significant tuning to achieve the speedup that was achieved in this study. An in-depth analysis of the Xeon Phi implementation revealed that the application was experiencing a large number of L2-TLB misses. Padding was

introduced to avoid this loss in performance. Also, individual experiments were performed to determine the appropriate number of threads and the affinity setting to use. On the other hand, the Kepler recycled code that was originally written for the Fermi architecture, which is the predecessor of the Kepler. As stand-alone systems the Xeon Phi and the GPU exhibited comparable performance. Larger differences between the devices were recorded in multi-system configurations with the GPU's parallel efficiency being much higher than that of the Xeon Phi.

2.3 Portable Performance: LULESH and Programming Models

Accelerators are known for following very strict programming models. And, the programming model used to develop an application has a direct effect on the performance it obtains on a computing platform. LULESH, the application of interest in this study, has been used to determine the effect of programming models on performance. In [10] LULESH was used to determine the performance of different optimizations implemented in eight programming models: Serial, OpenMP, MPI, CUDA, Chapel, CHARM++, Liszt, and Loci. The authors explored the portability of the following optimizations: loop fusion, vectorization, global allocation of data, data structure transformations, blocking, and communication and computation overlap. The study presents the strengths and weaknesses of each programming model evaluated, emphasizing the features of each model that allowed each optimization to be implemented, if in fact it could be implemented. For example, Chapel's domain maps can be used to facilitate implementation of the blocking optimization, and its asynchronous communication constructs makes it possible to overlap communication and communication. Meanwhile, communication and communication overlap occurs naturally with CHARM++, and any optimization that is applicable in C++ can also be implemented to this programming model. The LULESH ports were used to compare the scalability of the emerging programming models with the applied optimizations. The Sandy Bridge and Blue Gene/Q architectures were utilized to perform these experiments. It was found that Chapel achieved more than 80% efficiency on 16 cores of the Sandy Bridge, but had worse single-

core performance than OpenMP, while CHARM++'s performance was found to be comparable to the MPI implementation of LULESH.

The authors of [11] included LULESH in a study that examines different arrangements for the data layouts used in LULESH. LULESH represents a 3D hexahedral mesh structure that is characterized by 19 element-centered arrays, 13 node-centered arrays, and 3 symmetry arrays that represent the outside surfaces of the domain. Experiments were conducted to determine if LULESH's performance would benefit from different interleavings of the arrays. The experimental results showed that the modifications to the data layouts provided a speedup in execution time of up to 1.02x for the Sandy Bridge, 1.61x on the AMD APU, and 1.82x for the IBM Power 7 as compared to the base case (which is the runtime of the original code using the same number of threads). The data layout changes were not as beneficial on the Sandy Bridge, because this optimization reduced data motion in memory-bound sections of the code, but prevented the compiler from generating some of the SIMD instructions that had been included in the base version of LULESH in compute-bound sections of the code. Nevertheless, the findings of this work demonstrate that data layout changes can provide performance enhancements when porting to a different architecture.

Since LULESH is a proxy application, the tuning techniques applied to this code can potentially be ported to large-scale applications that include code sections similar to those that comprise LULESH. In [12] Karlin, McGraw, Keasler, and Still measure the impact of optimizations applied to LULESH on the Sandy Bridge architecture ported to the BlueGene Q architecture, and then the impact of these same optimizations applied to a large-scale application, i.e., a subset of ALE3D, also ported to the BlueGene Q. The optimizations were loop fusion, global allocation, increased vectorization, and NUMA-aware allocation. However, only loop fusion, increased vectorization, and global allocation were ported to the BlueGene Q architecture.

Loop fusion was found to reduce the runtime of LULESH on both machines, and increase its scalability. Global allocation significantly reduced the number of mallocs and frees placed inside LULESH. As a result, less than 0.1% of the total runtime was devoted to the serial portions

of LULESH through global allocation. Increased vectorization proved to be advantageous on the Sandy Bridge processor; it reduced the runtime by 25-27%. In contrast, this optimization negatively affected the code's runtime on the BlueGene Q architecture because the IBM compiler was not able to vectorize the tuned code. NUMA-aware allocation was also only applicable to the Sandy Bridge architecture.

Porting the optimized ALE3D subset to BlueGene Q was challenging. Only the loop fusion optimization was portable. This is because global allocation would have substantially increased the memory footprint of the application, and applying the techniques that resulted in increased vectorization would have required substantial changes to the structure of the code. Nevertheless, the loop fusion optimization was able to reduce the runtime of the ALE3D subset by up to 20%.

The work in [12] was further expanded by porting the optimizations to an NVIDIA GPU in [13]. Both the data allocation and loop fusion were ported to the CUDA version of LULESH for the Fermi architecture. The data allocation optimization was able to reduce the runtime of LULESH by 13%. The loop fusion allocation was applied to one of the most compute-intensive functions of LULESH. However, the introduction of the optimization did not provide a speedup. In fact, the loop fusion optimization increased the execution time of LULESH on the GPU.

2.4 Comparison of Related Work with Thesis Research

Similar to the work described above, the study presented in this thesis also compares the performance of different architectures. The devices explored in this study are representative of the Kepler, Fermi, Sandy Bridge, and Xeon Phi (MIC) architectures. An exploration of the literature highlights that work has been done to demonstrate the performance that can be obtained from each of these devices [2-4]. Each device has been the target of efforts to parallelize different types of scientific applications, and benchmarks have been used to stress different components of each system [3-5, 7]. The programming model of each has also been analyzed. Specifically, it has been shown that several optimizations make efficient use of the resources of each device [2, 3, 8, 11-13]. For example, it is known that reducing branch statements is beneficial to GPU architectures

because it avoids stalls in the execution pipeline. It is also known that greater performance can be achieved on the Xeon Phi by vectorizing the code as much as possible, which takes advantage of the large width of the vector unit and exploits the parallelism in the application. Although the comparative studies explored in this literature review [6-9] compare performance in terms of execution time, power consumption, and programmability, they do not delve into the cause of the performance of each architecture studied. This comparative study uses the same algorithms (LULESH code segments) to study each of the aforementioned architectures, determines the performance of the algorithms on each architecture, and provides an analysis of the performance of each code segment that explains the performance provided by the architecture.

It can be argued that comparative studies are not necessarily new. In particular, there already exist studies that compare the Kepler or Fermi GPU architectures with that of the Xeon Phi co-processor (i.e., the MIC architecture). Many of these studies use the Sandy Bridge architecture as a baseline for comparison [2, 4, 7-13]. Nevertheless, the approaches that have been taken to compare these devices are very different. All studies, including this one, provide some focus on the execution time due to the fact that it is the main metric used to measure performance, but there are peculiarities to each study that have been overlooked. For example, some studies have explored the performance of these devices under a multi-system configuration. Instead, this study compares the performance of a single-node configuration. Studies that have observed these processors and accelerators in such a configuration [2-5, 7, 8, 10-13] have considered fewer metrics for the comparison. One study focused on programmability [8], while another focused on power consumption [7]. In contrast, this study concentrates not only on power consumption and execution time, but also on memory performance, IPC, and utilization of vectorization. In addition, this study does not restrict itself to one type of GPU architecture as other studies have done. And, finally, most of the results used as a basis for comparison were collected through profiling, which is a process that seems to have not been employed in other studies [2-6, 8, 10]. Specifically, in [9] profiling is mainly used to explain a performance bottleneck for only the device that is experiencing inferior performance. Thus, this may be the only study that analyzes these

architectures in such detail, while using the same application for each device. Note that this is not always the case for all studies. In particular, the study described in [6] compares the architectures via an architectural overview and the study discussed in [7] uses the Rodinia benchmarks to compare the Xeon Phi to the Sandy Bridge, while using the SHOC benchmarks to compare the Xeon Phi to the Fermi GPU.

LULESH, the application of interest in this study, has been used for different types of comparative analyses [10-13]. However, the foci of these research endeavors are very different from the focus of this study. LULESH has been used exclusively to compare tuning techniques across programming models. In other words, the algorithms and optimizations inherent in LULESH have been ported across programming models to evaluate each model in terms of programmability and also to establish the portability of each optimization. While this study does delve into these topics, it does not have such a concentrated focus on programming models nor the programmability of the devices. Instead, the main focus of this study is to understand the performance that each architecture provides to the LULESH proxy application. Since no code changes were introduced to the existing versions of LULESH, programmability is not a concern of this study.

While this study has limitations that other studies did not encounter, e.g., power monitoring is not as fine-grained as that of [7] and experiments are restricted to a single-node configuration as opposed to the multi-system explored in [9], it collects several event counts on each architecture to understand its performance. In both [7] and [9] this is only done for the Xeon Phi architecture. The devices included in this study are still widely used by the HPC community, and any information regarding their performance can be used to help make a decision on whether it would be beneficial to offload an application to any of these devices. Moreover, LULESH is an application that is modeled after large scientific applications that consist of a significant portion of the workloads that are run on HPC systems. Therefore, the information gained from the analysis of the performance of this code on various architectures can be directly applicable to those codes that use the same algorithms as LULESH.

Chapter 3: Experimental Methodology

For this comparative study, power, energy, execution time, overhead due to parallelism, and additional metrics were studied for LULESH 1.0 executed on the Sandy Bridge, Xeon Phi, Kepler, and Fermi architectures. The methodology to obtain and analyze each of these measurements varied due to the differences in the target architectures. Section 3.1 describes LULESH 1.0 and explains why it is an appropriate code for this study. The platforms used to study its execution-time behavior are detailed in Section 3.2, and Section 3.3 presents the experimental plan for collection of each metric.

3.1 LULESH 1.0

The Department of Energy (DOE) initiated the Fast Forward project to accelerate the research and development necessary to shift to extreme-scale computing. One of the contributions of the Lawrence Livermore National Laboratory (LLNL) to this project was the development of the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) proxy application. LULESH solves the Sedov blast problem using Lagrangian hydrodynamics. A large number of scientific applications solve equations of hydrodynamics and/or model them. Hydrodynamics equations describe the motion of materials relative to each other when subject to forces, and the algorithms used to solve these equations present unique computational issues. In particular, many simulation problems involve complex multi-material systems that experience considerable deformations. As a proxy application, LULESH is a simplified code that is larger than a number of benchmarks, but is smaller than full-fledged scientific applications. Nonetheless, it represents the numerical algorithms, data motion, and programming style typical in scientific C or C++ based applications. Furthermore, its analytic solution can be scaled to large problem sizes. Consequently, the lessons learned from applying different optimizations to this code translate well to larger applications that either solve hydrodynamics equations or apply similar algorithmic strategies. Given the potential of this application, it has been ported to several programming models and tuned by experts for multiple computer architectures. This has allowed the use of

LULESH for various studies that compare programming models in terms of performance, ease of tunability, and usability.

Although LULESH has been used mainly to compare programming models, little work has been done to investigate how or why the performance of this application varies across platforms. Nevertheless, LULESH is an appropriate code for such a study because its translation to different programming models translates to its execution on different architectures (e.g., NVIDIA GPUs with the CUDA programming model). Table 3.1 lists the code versions of LULESH that were used for this study.

Table 3.1: Versions of LULESH 1.0 used in study.

Code Name	**Description**
LULESH.cc	Serial reference code
LULESH_OPTIM_OMP_ALLOC.cc	OpenMP code optimized for Sandy Bridge architecture
LULESH_OPTIM_OMP_ALLOC_DL.cc	Optimized OpenMP code with data layout changes
Luleshfermi.cu	Fermi code
lulesh.cu	Kepler code

LULESH.cc is the best serial version of LULESH. This program runs on both the Sandy Bridge and Xeon Phi architectures. However, for this study, this program is used solely on the Sandy Bridge architecture to obtain a baseline for performance. LULESH_OPTIM_OMP_ALLOC.cc was tuned for the Sandy Bridge architecture. The optimizations implemented in this program consist of loop fusion (the combination of multiple loops into a single loop), global allocation (allocation and deallocation of all temporary variables outside of the main time-step loop), and vectorization. Note that loop fusion also resulted in array contraction, since temporary arrays, which were needed to store data computed in between loops

that were fused, were no longer required. Vectorization was enabled by unrolling loops and in-lining function calls. This strategy takes into account the vector length of the architecture, and it depends on the compiler's ability to vectorize the code. LULESH_OPTIM_OMP_ALLOC_DL.cc differs from LULESH_OPTIM_OMP_ALLOC.cc in terms of the layout and access of data. Instead of storing the mesh in a struct of arrays, it is stored in arrays of structs. In addition, data that was accessed consecutively is now interleaved to improve memory performance. The LULESH programs for the GPU architectures, Luleshfermi.cu and lulesh.cu, were implemented using LULESH_OPTIM_OMP_ALLOC.cc as a template. Minimal tuning techniques were applied to make better use of the Kepler and Fermi architectures; these techniques involve the arrangement of data in a way that is favorable to the GPU architecture and the mapping of sequential loops to threads [10]. For simplicity the LULESH_OPTIM_OMP_ALLOC.cc will be referred as the Optimized OpenMP code (or Opt code), and LULESH_OPTIM_OMP_ALLOC_DL.cc will be referred as the DL Optimized OpenMP code (or DL code) in the remainder of the thesis.

Intermediate I/O does not affect LULESH's execution time. Additionally, for this study, the initialization and termination steps are not included for the comparative analysis. This is due to the fact that these code sections do not map well to real applications; for example, they were not developed using coding techniques that are used in real applications. Nevertheless, the initialization and termination code sections consume a minimal portion of the execution time of all LULESH versions used in this study. The hydrodynamics equations are approximated in LULESH by partitioning the problem domain into a collection of elements. A mesh defines the collection of elements, and its size is used to scale LULESH. For this study, only mesh sizes of 50^3, 70^3, and 90^3 elements were considered due to the fact that these problem sizes are representative of the workload that would be allocated to a single node when a system executes a large-scale application.

For this study, the time-stepping algorithm used in LULESH was broken down in the following four phases:

1. Calc Volume Force: computation of the forces of each element based on the values of the mesh variables at the given time-step; validation of the computation is also included in this phase
2. Calc & Apply Accel: computation of the acceleration of the nodes and the application of boundary conditions to the acceleration
3. Lagrange: advancement of element quantities, in particular, the pressure, internal energy, and relative volume are updated
4. Time Constraints: calculation of the courant and hydro constraints controlling the main computation loop

The identification of these phases was required in order to be able to compare the performance of the different code versions used in this study. The exact mapping applied to each code version is included in Appendix A.1.

Illustration 3.1 presents a high-level chart that indicates the flow of execution of LULESH. The four phases of interest in this study are highlighted in the figure. The grayed out portions of LULESH are part of the execution, but are excluded from the execution time and profiling data associated with this study. Note that LULESH is characterized by a main computation loop, in which the phases of interest are computed, with the exception of the Time Increment step, where the time variable is increased. Emphasis was not placed on the Time Increment step because the computation regarding the constraints placed on the time variable is performed in the Time Constraints phase.

The execution of LULESH is deterministic. The main computation loop executes for a defined number of iterations, which is based on the specified problem size. As mentioned before, only three problem sizes were considered in this study. The number of iterations the loop executes for the problem sizes of interest are the following: 1,566 for a mesh size of 50^3, 1,816 for a mesh size of 70^3, and 2,026 for a mesh size of 90^3.

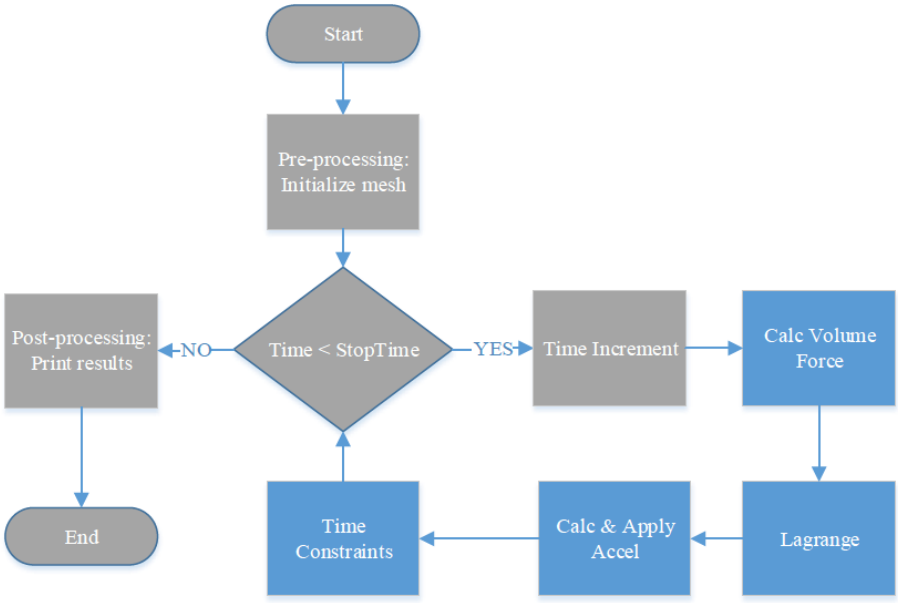


Illustration 3.1: LULESH 1.0 flowchart.

3.2 Experimental Platform

An HPC cluster is composed of a large number of compute nodes that can be used concurrently to solve large parallel applications. A single node in such a system can consist of a number of processors. It is also becoming more common for such systems to have heterogeneous nodes, i.e., ones that include different computer architectures in a single node (e.g., a host processor with a GPU). This study focuses on a node configuration that consists of a host processor paired with an accelerator. A node configured in such a manner has up to four different modes that can be used to execute a program or a part of a program; (1) CPU-only mode: use CPU(s) only; (2) native mode: use CPU(s) only for setup and termination, and use accelerator(s) only for the main computation phase; (3) offload mode: use the accelerator(s) only to compute specified sections of a program; and (4) symmetric mode: use both CPU(s) and accelerator(s) simultaneously to execute a program by partitioning data among the devices. In this study, we evaluate the Sandy Bridge, Many Integrated Core (MIC), CUDA Fermi, and CUDA Kepler architectures. Only the first two modes of execution are used in this study to compare the performance that can be achieved using

the three different accelerators and the host processor. The devices used for this study are described in Table 3.2.

Table 3.2: Architectures of accelerators and host processor.

Accelerator	Speed	# Cores	Max. Threads per Core	L1 Cache	L2 Cache	L3 Cache	RAM	Memory Bandwidth	Memory Type
Intel Xeon E5-2680: Sandy Bridge, SB	2.7GHz	8	2	64KB/core (32KB L1D, 32KB L1I)	256KB/core	20MB shared	750GB	51.2GB/s	DDR3, 1600MHz
Intel Xeon Phi SE10P: Phi	1.1GHz	61	4	64KB/core (32KB L1D, 32KB L1I)	512KB/core	N/A	8GB	352GB/s	GDDR5
NVIDIA Tesla M2050: Fermi	1.15GHz	14 (SM)	1,536/SM	Up to 48KB L1D/SM	768KB shared	N/A	3GB	148GB/s	GDDR5
NVIDIA Tesla K20: Kepler	705MHz	13 (SM)	2,048/SM	Up to 48KB L1D/SM	1,536KB shared	N/A	5GB	208GB/s ECC off	GDDR5

TACC's Stampede cluster was used to perform runtime and profiling experiments on the Sandy Bridge, Xeon Phi, and Kepler, while Fermi experiments were executed on LLNL's Edge cluster. This is due to Stampede compute nodes being composed of only Sandy Bridge processors paired with either Xeon Phi and/or Kepler architectures. Unfortunately, neither of these systems allowed collection of all the data required for this study. Thus, stand-alone servers at the University of Tennessee, Knoxville (UTK) were employed as well. The specifications of the UTK systems' accelerators are the same as those of the Xeon Phi, Fermi, and Kepler devices used in Stampede. The Sandy Bridge processor of the UTK system is similar to, but not identical to the one in Stampede. Stampede is composed of Intel Xeon E5-2680 processors, while the UTK system consists of an Intel Xeon E5-2690 processor, which has a slightly faster clock rate of 2.9GHz.

3.3 Experimental Design

As mentioned earlier, this study concentrated on a number of metrics to compare the performance of LULESH 1.0 executed on the accelerator architectures described in the previous section. The performance metrics considered for each architecture were execution time, memory performance, instructions executed per clock cycle, vectorization efficiency, power consumption, and energy consumption. Note that, only the solve time of LULESH is being considered since the pre- and post- processing (initialization and termination) steps do not reflect those of typical hydrodynamics codes used at LLNL. Due to the differences in the design of the architectures under study, a process specific to the architectures had to be employed in order to collect each metric. The following sections detail the experimental design used in this study to obtain each metric.

3.3.1 Compiling LULESH

In order to compile LULESH for the Sandy Bridge and Xeon Phi architectures, the Intel compiler was used. Currently, only the Intel compiler can create a binary that is executable on the Xeon Phi. Furthermore, execution-time comparisons of LULESH compiled with the GNU compiler versus the Intel compiler indicated that the Intel compiler performed more aggressive optimizations than the GNU compiler, and resulted in better performance for the Optimized OpenMP, DL Optimized OpenMP, and serial versions of LULESH. The version of the Intel compiler used for this study was 13.1.0.146. LLNL's technical reports on LULESH [14] indicate the flags required to obtain the optimal execution time for LULESH executed on the Sandy Bridge architecture. To compile LULESH for the Kepler and GPU architectures, the NVIDIA CUDA Compiler driver (NVCC) was required. In this study, the version of the driver used was V0.2.1221 from the 5.0 release of CUDA. The version of LULESH developed for the Kepler can be obtained with an accompanying Makefile at https://codesign.llnl.gov/lulesh.php. Table 3.3 shows how each version of LULESH was compiled.

Table 3.3: Compiler commands.

Code Name	Compiler Command
LULESH.cc (serial code)	`icc LULESH.cc -O3 -o LULESH`
LULESH_OPTIM_OMP_ALLOC.cc	SB: `icc LULESH_OPTIM_OMP_ALLOC_VEC3.cc -openmp –mavx -O3 -o LULESH_OPTIM_OMP_ALLOC_VEC3` Phi: `icc LULESH_OPTIM_OMP_ALLOC_VEC3.cc -openmp -O2 -mmic -o LULESH_OPTIM_OMP_ALLOC_VEC3`
LULESH_OPTIM_OMP_ALLOC_DL.cc	SB: `icc LULESH_OPTIM_OMP_ALLOC_VEC3_DL.cc -openmp -O3 -mavx -o LULESH_OPTIM_OMP_ALLOC_VEC3_DL` Phi: `icc LULESH_OPTIM_OMP_ALLOC_VEC3_DL.cc -openmp -O2 –mmic -o LULESH_OPTIM_OMP_ALLOC_VEC3_DL`
Luleshfermi.cu	`nvcc "-arch=sm_20" -O3 -c stopwatch.c` `nvcc "-arch=sm_20" lulesh.o stopwatch.o -O3 -o lulesh`
lulesh.cu	`nvcc "-arch=sm_35" lulesh.o allocator.o -O3 –o lulesh`

Since the OpenMP codes, i.e., (Optimized) LULESH_OPTIM_OMP_ALLOC.cc and (DL Optimized) LULESH_OPTIM_OMP_ALLOC_DL.cc, were optimized for the Sandy Bridge, the codes had to be slightly modified to execute properly on the Xeon Phi. Specifically, the VEC_LEN variable in the Optimized and DL Optimized codes, which was set to four for the Sandy Bridge, was changed to eight for the Xeon Phi. This variable, which indicates the vector length of the processor, is used to aid in vectorization of the loops in LULESH. In addition, neither of the OpenMP codes was compiled for the Xeon Phi with the same flags that were used to compile them for the Sandy Bridge. The `–mavx` flag was not supported by the MIC architecture of the Xeon Phi, and execution-time comparisons of the codes compiled with the O2 and O3 levels of optimization indicated that better performance could be obtained by using O2 on the Xeon Phi, while the Sandy

Bridge experienced better performance with the O3 optimization level. Results of these experiments are included in Chapter 4.

3.3.2 Running LULESH

Almost all versions of LULESH were executed in the same fashion, i.e., by using the following command:

```
./<executable> <problemSize>,
```

where <executable> refers to the binary of LULESH and <problemSize> indicates the mesh size to be used in the computation. As mentioned earlier, the problem sizes used in this study are 50^3, 70^3, and 90^3. To specify these values to the executable, we simply use the inputs 50, 70, and 90, respectively. The program cubes the input size to create the mesh. Only the Fermi code hard wires the problem size using the MESH_RESOLUTION variable and, thus, does not expect the size to be an input parameter. The Fermi code also allows for the use of different block sizes, however, the preset block size is 256, which was used in this study.

Regarding the OpenMP versions of LULESH, to avoid leaving cores idle, the maximum number of threads that can be used on the two Sandy Bridge processors and one Xeon Phi of a Stampede node were employed. Since Stampede does not support hyperthreading on the Sandy Bridge processors, the number of threads used was set equal to the number of cores available on the two Sandy Bridge processors of a node, i.e., 16, and the maximum number of hardware threads available on the Xeon Phi, which is equal to four threads per core times 61 cores, i.e., 244. For the Xeon Phi, experiments with different number of threads and affinity settings (which control how the threads are mapped to the cores of the processor) were performed to identify the best runtime environment for both of the optimized OpenMP versions. Both versions were executed on the Xeon Phi using 244, 240, 122, and 120 threads with `scatter`, `compact`, `balanced`, and `no affinity` for all problem sizes. The results of these experiments are presented in Appendix A.3; they indicate that LULESH would achieve best performance by employing all available hardware threads. Also, both the optimized code and the DL optimized code achieved optimal performance

with `compact` affinity and 244 threads. A summary of the environment parameters that were used to run the five versions of LULESH on the four architectures is presented in Table 3.4.

Table 3.4: Runtime environment parameters.

Code Name	OMP_NUM_THREADS	KMP_AFFINITY	Block Size
LULESH.cc (serial code)	n/a	n/a	n/a
LULESH_OPTIM_OMP_ALLOC.cc	SB:16; Phi: 244	SB: n/a; Phi: `compact`	n/a
LULESH_OPTIM_OMP_ALLOC_DL.cc	SB: 16; Phi: 244	SB: n/a; Phi: `compact`	n/a
Luleshfermi.cu	n/a	n/a	256
lulesh.cu	n/a	n/a	dynamic

3.3.3 Execution Time

In this study execution time is defined as the period of time (in seconds) a code segment executes. The measurement of this metric initiates at the execution of the first instruction of the code segment and terminates at the execution of the final instruction. In order to make a proper comparison of the different phases of LULESH, the time consumed by each phase was recorded as well as the overall solve time of the entire program. This data made it possible to determine the percentage of time that each of the architectures devoted to the identified phases of LULESH. Additionally, for the Sandy Bridge and Xeon Phi, the overhead introduced by parallel constructs was calculated to establish how much of LULESH's solve time was consumed by the overhead. The subsections below describe how the execution time of LULESH was collected for the architectures under study. In addition, the process employed to resolve how the execution time is distributed among the phases of LULESH is also described.

Overall Solve Time of LULESH

Assuming that the best runtime environment for LULESH on each platform was established, experiments to collect the execution time of the application were performed. In this study, each binary was launched in the best runtime environment to collect the execution time of LULESH on each of the architectures of interest using three different problem sizes (50^3, 70^3, 90^3). A total of 10 trials of each execution were run to obtain the maximum, minimum, and average execution times, along with the standard deviation. For each problem size, the average execution

time was used to compute the speedup with respect to that of the best serial version of LULESH execution on one core of the Sandy Bridge processor.

All versions of LULESH include the timing constructs that are required to collect the execution time of the application. For the Intel Sandy Bridge and Xeon Phi devices, the C function `gettimeofday()`, which collects the current time expressed in seconds and microseconds, was employed. On the other hand, the Fermi and Kepler require the use of CUDA Events. CUDA Events are objects created for the purpose of collecting a performance metric of interest. `gettimeofday()` cannot be utilized to measure execution time on a GPU since the clock of the host processor is not necessarily synchronized with that of the accelerator. CUDA Events are recorded in the CUDA call streams and, thus, can be used to obtain timestamps from the device. A CUDA event is required to obtain a timestamp before the execution of the region of interest begins, and another is required to obtain a timestamp after the execution of the region ends. The timestamps are collected through calls to `cudaEventRecord`. Finally, a subsequent call to `cudaEventElapsedTime()` reports the elapsed time between the recorded timestamps of the two CUDA events.

Distribution and Percentage of Execution Time

As mentioned earlier in Section 3.1, an analysis of LULESH revealed that the application can be partitioned into four phases of execution, which allow comparison of execution times across the studied architectures. Both the Optimized and DL Optimized OpenMP versions of LULESH have timing constructs that surround the 12 major loops in LULESH but these are commented out. These constructs can be and were used to obtain the distribution of time among the four phases of the execution of the program. A sample output of the optimized OpenMP code with the timing constructs activated is shown in Figure 3.1.

```
Elapsed time = 1.379130e+01

Run completed:
   Problem size        =  50
   Iteration count     =  1566
   Final Origin Energy = 4.052502e+05
   Testing Plane 0 of Energy Array:
        MaxAbsDiff   = 8.003553e-11
        TotalAbsDiff = 3.133088e-09
        MaxRelDiff   = 1.142118e-12


Elapsed time = 2.364429e+00

Elapsed time = 1.572556e+00

Elapsed time = 3.728118e+00

Elapsed time = 1.254355e+00

Elapsed time = 2.826410e-01

Elapsed time = 1.364800e-02

Elapsed time = 3.321620e-01

Elapsed time = 2.822255e+00

Elapsed time = 1.140152e+00

Elapsed time = 3.371500e-02

Elapsed time = 1.865570e-01

Elapsed time = 5.008900e-02
```

Figure 3.1: Optimized OpenMP LULESH Output.

The output of LULESH first reports the solve time, followed by the problem size, iteration count, and the result of the computation, i.e., the Final Origin Energy. With the additional timing constructs included, the output also reports how long each of the 12 parallel loops took to execute. Given this data, we mapped the execution times of these parallel loops to the four major execution phases of LULESH. The mapping that we defined is shown in Illustration 3.2. First we identified the function in which each loop was executed and then mapped the 10 functions and, thus, the 12 loops to the four major execution phases of LULESH. Considering that both the Optimized and DL Optimized OpenMP versions of LULESH have a minimal amount of serial code (mainly associated with pre- and post-processing), the execution times associated with each loop in the

same phase were added to estimate how much of the execution time of LULESH is attributed to a given phase.

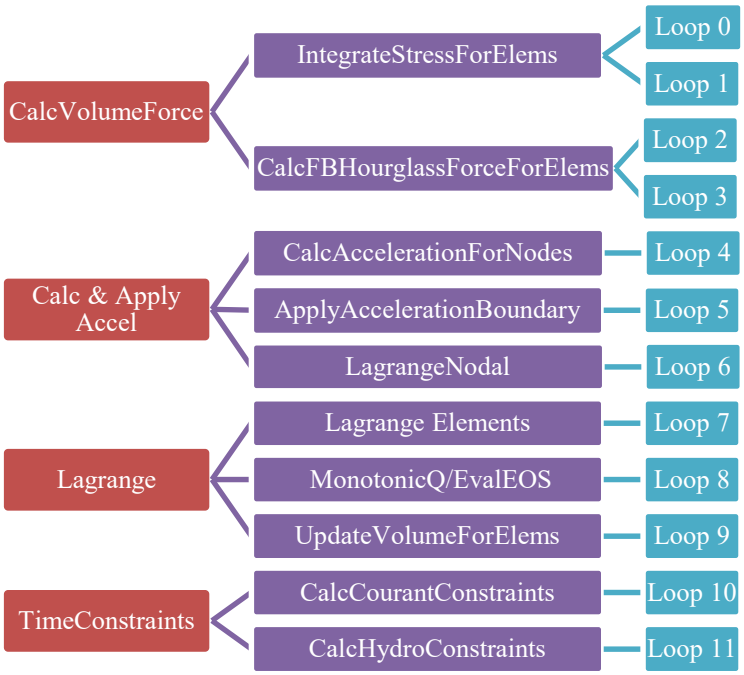


Illustration 3.2: Mapping of the 12 parallel loops in LULESH to the four phases of execution.

The version of LULESH associated with the Fermi architecture includes timing constructs for all the functions in LULESH. Using the same mapping of functions to phases, the distribution of time in LULESH was determined by adding the execution times of the functions mapped to the same phase.

Unfortunately, the version of LULESH associated with the Kepler did not include any timing constructs. Furthermore, the Kepler's overall execution time increased considerably when additional CUDA Events were introduced to record the execution time of each function. Specifically, timing constructs were originally going to be used to measure the overhead of synchronizing threads on the Kepler. At the largest problem size, the solve time of LULESH on the Kepler is less than 20 seconds, but with the introduction of timing constructs the Kepler took over 24 hours to complete its execution. Although it was not possible to instrument the code

without introducing significant perturbation, the information required to use the established function-to-phase mapping was collected through the use of the NVIDIA Visual Profiler (NVVP). NVVP was developed by NVIDIA to collect performance metrics and events available on the device. It requires that the application be executed at least once to provide a timeline, which it presents visually; the timeline indicates the frequency of calls to each CUDA kernel. The measurements made by the device regarding execution time and the performance metrics of each function are provided to the user. For this study those measurements were used to extract how much time was spent in each function during the execution of LULESH. Once the execution times were collected, the same methodology used for the Fermi was used for the Kepler to determine the distribution of execution time and the percentage associated with each of the four phases of LULESH.

Parallel Overhead

The experiments for measuring the parallel overhead were designed using the methodology employed by the EPCC OpenMP microbenchmarks [15]. For those benchmarks, the overhead is defined as:

$$O_p = T_p - T_s / p,$$

Equation 1: Parallel overhead.

where O_p is the parallel overhead associated with the execution of a parallel program and p is the number of processes used to execute it. T_p is the execution time of the parallel (OpenMP) program, while T_s is the execution time of the parallel program compiled without the `-openmp` flag. By removing this flag from the compilation command, the OpenMP pragmas are ignored by the compiler, forcing the program to be executed in serial mode. For the OpenMP versions of LULESH, removing the `-openmp` flag also requires a change to the value of `threads`, i.e., it must be changed from `omp_get_max_threads()` to `Index_t(1)` in the `CalcCourantConstraintForElems()` and `CalcHydroConstraintForElems()` functions. Once these changes were made to both the Optimized and DL Optimized OpenMP

versions of LULESH, the codes were compiled with the following commands for the Sandy Bridge (SB) and Xeon Phi (Phi):

```
SB: icc  <codeName>.cc –mavx -O3 -o <executable>
Phi: icc  <codeName>.cc -O2 –mmic -o <executable>
```

The serial execution time of the two codes, for all three problem sizes of interest, were collected for the Sandy Bridge and Xeon Phi. For each, the serial execution time of the overall application was recorded along with the serial execution time of each of the 12 loops described in the previous section. To collect the parallel execution times, the codes were compiled as described in Section 3.3.1. On the Sandy Bridge, these experiments were conducted using 2, 4, 8, and 16 threads in order to measure the growth of the overhead as the number of threads increases. On the Xeon Phi, 60, 120, and 240 threads were employed. In order to ensure that the all of the cores of the Xeon Phi were utilized and that the number of threads per core was the same, `balanced affinity` was used. This affinity setting causes the threads to be assigned to cores with consecutive ids and threads on the same core to be scheduled in a round-robin fashion. Once the overall serial and parallel execution times of each code as well as the serial and parallel execution times of each of the 12 loops of LULESH were obtained, Equation 1 was employed to compute the parallel overhead. Then for each code, the overheads associated with the execution by different numbers of threads were compared.

3.3.4 Power/Energy

Due to the differences among the devices used in the study, it was difficult to find a tool that could be used to obtain the power draw of all the architectures of interest. As a result, obtaining this information required the use of different tools. Nevertheless, the goal of these experiments was to extract the power and energy consumption of each device while it is executing LULESH.

Power Draw of GPUs

Both of the GPU architectures used in this study were designed by NVIDIA. Along with its release of architectures and accelerator devices, NVIDIA released a set of tools that can be used

to collect performance metrics on its GPU devices. One such tool is the NVIDIA System Management Interface program (NVIDIA-SMI), which can be used to monitor and manage NVIDIA GPU devices. The utility is included with the installation of the NVIDIA Linux graphics driver and contains a vast range of options to view the system state of and diagnostic information about a GPU. The query used to attain the power consumption of both the Kepler and Fermi architectures is:

```
nvidia-smi -i 0 -q -d POWER | grep "Power Draw"
```

This query reads and records the power consumption (`-d POWER`) of the GPU with id zero (`-i 0`). The reading has a +/- 5% accuracy. In order to utilize this utility to obtain the power consumption of executing LULESH on the Fermi and Kepler GPUs, the following code was included in a script:

```
while true
do
    nvidia-smi -i 0 -q -d POWER | grep "Power Draw"
    sleep .1
```

The script is meant to run along with LULESH to query the power consumption every .1 seconds. Once LULESH terminates, the script terminates as well. The values collected in the script are subsequently analyzed to collect the maximum, minimum, and average power consumed.

Power Draw of Intel Xeon Phi

The Xeon Phi allows power usage to be read natively. Intel's System Management Controller (SMC), one of the primary subsystems of the Xeon Phi, is responsible for collecting thermal status information and communicating with the device for power state control. In other words, the SMC monitors the temperature and the power consumption of the device to signal when thermal or power consumption limits are being reached. The SMC samples power every 500 milliseconds and exports its readings via `sysfs` (a virtual file system) to `/sys/class/micras/power`. Consequently, power consumption can be read by viewing the

data in the `power` file. For this device, a script also was written to run simultaneously with LULESH, which is comprised of the following code:

```
while true
do
    cat /sys/class/micras/power
    sleep .1
done
```

The script reads the power consumption of the Xeon Phi every .1 seconds by querying the data in the `power` file. The collected readings are then used to compute the maximum, minimum, and average power consumed by the Xeon Phi.

Energy Consumption of Accelerators

Although the process followed to obtain the power consumption of the accelerator devices varied, the calculation of energy was consistent across the architectures. In physics, power is defined as the amount of work done per unit time. The amount of work completed also can be defined as the amount of energy consumed by a process. Since we collect the execution time and power draw of LULESH on each of the accelerators of interest, we can easily calculate the amount of energy consumed by LULESH with the following formula:

$$E = P \times t,$$

Equation 2: Energy Consumption

where P is the average power required by a given architecture to execute LULESH and t is the solve time of LULESH on the given architecture. In other words, the energy consumed by an accelerator can be computed by attaining the product of the average power required by the device to execute LULESH and the time it takes to complete its execution.

Power and Energy Draw of Intel Sandy Bridge

Although the Sandy Bridge and Xeon Phi are both Intel devices, the considerable differences in their architectures do not permit the use of the same methodology to collect power consumption. Like the Xeon Phi, one can query the power consumption of the Sandy Bridge by

using the Linux `perf` tool. The tool is a profiler that allows the collection of numerous processor metrics, including power consumption. However, this tool requires root privileges, which were not provided. As a result, it was decided to use the PAPI (Performance Application Programming Interface) [16] RAPL (Running Average Power Limit) component. PAPI provides a consistent methodology to access performance counter hardware across architectures. The RAPL component Model Specific Register (MSR) can be read to collect the current energy consumption of the Sandy Bridge. Unlike the previous methodologies, separate scripts to monitor power information are not required. Instead, code was introduced to both the Optimized Open MP and DL Optimized OpenMP versions of LULESH to measure the power and energy consumption of the Sandy Bridge processor.

3.3.5 Profiling

Different performance metrics were collected in order to observe how LULESH takes advantage of the resources available in the architectures. Unfortunately, the difference in the architectures does not permit the use of the same tool to obtain all of the metrics required for analysis. In particular, few tools exist that support the MIC architecture of the Xeon Phi.

All code versions were profiled in terms of:

- Instructions per cycle (IPC)
- Memory performance
- Vectorization efficiency

IPC was collected to gain insight into the parallel efficiency of LULESH. Vectorization efficiency was also considered because the size of the Xeon Phi's vector unit allows for more parallelism.

Profiling on NVIDIA GPUs

The NVIDIA Visual Profiler is a tool that is part of the CUDA Toolkit. It is designed specifically to provide feedback on an application's use of the resources of an NVIDIA GPU. The tool runs the application of interest and generates a base profile that indicates the execution time

of the application and the time distribution among the different functions of the application. The profiler also provides separate metrics and events to gain a better understanding of an application's performance. Accordingly, this tool was used to obtain execution profiles on both the Fermi and Kepler. The tool allows for configuration of metrics and events to be collected, which are categorized as follows: memory, instruction, multiprocessor, cache, texture, and profile trigger. All metrics and events in these categories were collected, with the exclusion of the profile trigger. Since the profiler gives feedback on individual functions, the results were mapped to the execution phases of LULESH presented in Section 3.1.

Profiling on Intel Sandy Bridge and Xeon Phi

Although the Intel Sandy Bridge is a standard processor and there are several tools available to collect information from this device, attempts were made to use the same tool on both the Sandy Bridge and Xeon Phi. This proved to be a difficult task due to the novelty of the MIC architecture of the Xeon Phi. However, PAPI was updated to allow developers to access the counters on the Xeon Phi. As a result, both the Optimized and DL Optimized codes were instrumented using PAPI to collect event counts related to memory performance and IPC.

PAPI distinguishes between preset and native events. Preset events are predefined by the API and are built upon native events, while native events are platform specific. The amount of preset events was limited for both the Sandy Bridge and Xeon Phi. On both the Sandy Bridge and Xeon Phi the preset counters `PAPI_TOT_CYC` (cycle count) and `PAPI_TOT_INS` (instruction count) were collected to calculate IPC. On the Sandy Bridge the preset events `PAPI_L2_DCA` (L2 data cache accesses) and `PAPI_L2_DCH` (L2 data cache hits) were collected, while on the Xeon Phi the preset events collected were `PAPI_L1_DCA` (L1 data cache accesses) and `PAPI_L1_DCM` (L1 data cache misses). Native events were used to obtain information on the performance of the L1/L3 cache on the Sandy Bridge, the L2 cache on the Xeon Phi, and of the TLBs of both architectures.

The specific native event counters used on the Sandy Bridge were `perf::PERF_COUNT_HW_CACHE_LL:MISS` (L3 data cache misses), `perf::PERF_COUNT_HW_CACHE_LL:ACCESS` (L3 data cache accesses), `perf::PERF_COUNT_HW_CACHE_DTLB:MISS` (data TLB misses), and `perf::PERF_COUNT_HW_CACHE_DTLB:ACCESS` (data TLB accesses), `perf::PERF_COUNT_HW_CACHE_L1D:MISS` (L1 data cache misses) and `perf::PERF_COUNT_HW_CACHE_L1D:ACCESS` (L1 data cache accesses). And, the specific native event counters used on the Xeon Phi were `perf::PERF_COUNT_HW_CACHE_LL:MISS` (L2 data cache misses), `perf::PERF_COUNT_HW_CACHE_DTLB:MISS` (data TLB misses), and `perf::PERF_COUNT_HW_CACHE_DTLB:ACCESS` (data TLB accesses).

The Sandy Bridge could collect eight counters simultaneously, but the Xeon Phi used in this study only permitted two counters to be collected at a time without multiplexing. Therefore, the Xeon Phi required multiple experiments to obtain all of the required data. Although the codes under study are parallel codes, only the events for a single thread were collected. This is due to the fact that four threads of the Xeon Phi use the same core. Due to limitations in the number of hardware counters available, this made it impossible to collect the event data for each thread on the Xeon Phi. However, OpenMP tries to assign work evenly among the number of threads being used, but there is no certainty that a thread will be assigned the same piece of work every time a parallel region is entered. As a result, we restricted the collection of event counts to a single thread. To maintain consistency in the profiling methodology between the Sandy Bridge and Xeon Phi processors, the same restrictions used on the Xeon Phi were applied to the Sandy Bridge. Code samples of how the codes were instrumented are included in Appendix A.2.

Chapter 4: Results

This chapter presents the results of the experiments that compare the execution time, speedup, power and energy consumption of different versions of LULESH 1.0 on the accelerators under study, i.e., the Intel Xeon Phi and the NVIDIA Kepler and Fermi GPGPUs. The performance of LULESH executed on a dual Intel Sandy Bridge is used as a baseline to conduct a performance comparison with the Xeon Phi. In addition, in an effort to explain the execution time and speedup data of the accelerators, the performance of LULESH on the architectures is further studied in terms of parallel efficiency, memory performance, and use of vectorization.

Prior to comparing the accelerators under study in Sections 4.2 and 4.3, the performance of the Optimized (Opt) and DL Optimized (DL) OpenMP codes executed on the Sandy Bridge and Xeon Phi is discussed in Section 4.1. This is done to explain why only the DL code is used in the comparison of the accelerators.

In this chapter we refer to the architectures and codes under study as architecture/code pairs. As mentioned in Section 3.1, with the exception of the Sandy Bridge and Xeon Phi, each architecture under study is paired with a particular code. The Fermi and Kepler GPUs are paired with parallel codes tuned for these particular platforms. In this chapter we refer to these architecture/code pairs as FGPU/F and KGPU/K, respectively. In contrast, two different codes, i.e., the optimized OpenMP code (with a 3D representation of the mesh) and the DL code (with a 1D representation of the mesh), were optimized for only the Sandy Bridge, but were executed on both the Sandy Bridge and Xeon Phi. Here these architecture/code pairs are referred to as SB/Opt and SB/DL, and Phi/Opt and Phi/DL, respectively.

In addition, as described in Section 3.3.3, the execution times reported and studied in this chapter include only the solve time of LULESH 1.0. Again, this is because the initialization and termination sections of the code do not use strategies and algorithms representative of real hydrodynamics codes run at the DoE Labs.

4.1 Sandy Bridge vs. Xeon Phi

As discussed in Section 3.1, while there is only one version of the LULESH code for each of the Kepler and Fermi GPUs, there exist two OpenMP versions, the DL and Opt codes that can execute on both the Sandy Bridge and Xeon Phi. The DL and Opt codes, which were both ported to the Sandy Bridge, are based on the same algorithm, but the DL code employs a different data layout for the mesh. As demonstrated in this section, the DL code, which is meant to optimize the performance of LULESH 1.0, performs better than the Opt code on both the Sandy Bridge and Xeon Phi; and Phi/DL outperforms SB/DL except for the 50^3 problem size. As a result, to ensure a fair comparison with the Kepler and Fermi, in Sections 4.2 and 4.3 we only compare the GPU architecture/code pairs with Phi/DL. The architecture/code pairs are compared in terms of execution time, and their performance is mapped to the IPC, vectorization, and memory behavior of LULESH on the GPUs and Xeon Phi.

4.1.1 Execution Time

As shown in Figure 4.1, both the Sandy Bridge (with 16 threads) and Xeon Phi (with 244 threads) execute the DL code in less time than is required to execute the Opt code. Referring to the figure, you can see that: (1) the execution time of SB/DL is 16.5%, 9%, and 10% smaller than that of SB/Opt for the 50^3, 70^3, and 90^3 problem sizes, respectively; and (2) Phi/DL's execution time is 43%, 45%, and 60% smaller than that of Phi/Opt for the same problem sizes. These data indicate that both the Sandy Bridge and Xeon Phi take better advantage of the 1D representation of the mesh in the DL code, as compared to the original 3D representation in the Opt code. Also, since, with the exception of the 50^3 problem size, Phi/DL outperforms SB/DL (albeit not by much, i.e., by 5.17% and 4.30% for the 70^3 and 90^3 problem sizes, respectively), most comparisons in Sections 4.2 and 4.3 are made only among the accelerators, i.e., Phi/DL and the GPU architecture/code pairs, KGPU/K and FGPU/F, which employ codes that were ported specifically to these architectures. Again, note that neither the Opt code nor the DL code was ported specifically

to the Xeon Phi, thus, it is likely that additional tuning could result in better performance on the Xeon Phi.

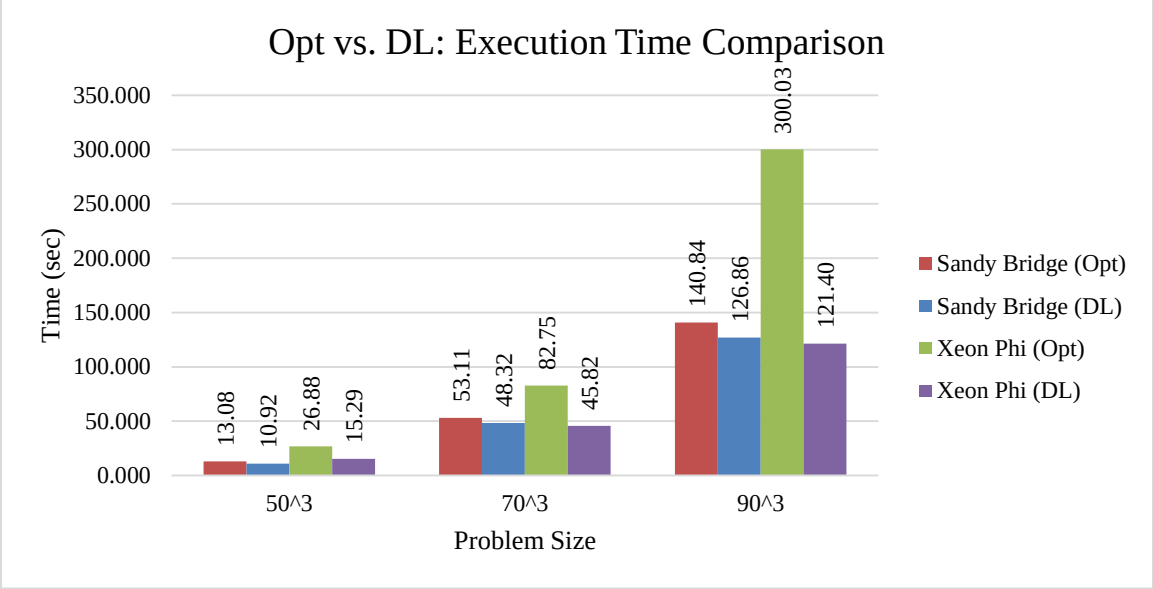


Figure 4.1: Execution Time Comparison of SB/ Opt, SB/ DL, Phi/ Opt, and Phi/DL.

Nevertheless, these results make it clear that the DL code is better suited, as compared to the Opt code, for both the Sandy Bridge and Xeon Phi. Note that even though these codes were not specifically ported to the Xeon Phi, the resultant increase in performance is much more significant for the Xeon Phi: for the three problem sizes, the increase in performance of Phi/DL over Phi/Opt is 26.5%, 36%, and 35% higher than that of SB/DL over SB/Opt.

The parts of LULESH 1.0 that benefit most from the optimizations incorporated in the DL code differ depending upon the architecture/code pair. This is shown in Tables 4.1-4.4, which present the distribution of execution times among the four major phases of LULESH 1.0. As can be seen, across the three problem sizes, for both the Sandy Bridge and Xeon Phi the improvement in performance is most evident in: (1) Calc & Apply Accel, for which the optimizations in the DL code decreased the execution time on the Sandy Bridge (Xeon Phi) by between 46.06% (9.4%) and 63.09% (49.51%), and (2) Lagrange, for which the optimizations decreased the execution time on the Sandy Bridge (Xeon Phi) by between 25.80% (19.38%) and 35.60% (70.58%). In addition,

for Phi/DL there was a significant decrease in the execution time of Calc Volume Force across the three problem sizes, i.e., it executed in 58.00%-68.51% less time than Phi/Opt.

Table 4.1: SB/Opt – Distribution of Time Spent on Phases (sec).

Phase	Problem Size		
	50^3	**70^3**	**90^3**
Calc Volume Force	8.994	36.292	97.692
Calc & Apply Accel	0.617	2.121	8.163
Lagrange	3.803	13.434	36.474
Time Constraints	0.228	0.701	1.755

Table 4.2: SB/DL – Distribution of Time Spent on Phases (sec).

Phase	Problem Size		
	50^3	**70^3**	**90^3**
Calc Volume Force	8.802	37.852	100.498
Calc & Apply Accel	0.320	1.144	3.013
Lagrange	2.822	8.953	23.488
Time Constraints	0.231	0.699	1.761

Table 4.3: Phi/Opt – Distribution of Time Spent on Phases (sec).

Phase	Problem Size		
	50^3	**70^3**	**90^3**
Calc Volume Force	18.694	57.584	170.766
Calc & Apply Accel	1.112	2.781	21.685
Lagrange	6.763	20.806	103.639
Time Constraints	0.456	0.938	2.224

Table 4.4: SB/Opt – Distribution of Time Spent on Phases (sec).

Phase	Problem Size		
	50^3	**70^3**	**90^3**
Calc Volume Force	7.852	23.301	53.781
Calc & Apply Accel	0.845	2.242	6.380
Lagrange	6.127	19.426	52.330
Time Constraints	0.415	0.979	2.296

4.1.2 Parallel Overhead and Computation Time

When a program is parallelized, code is introduced to create a team of parallel threads, distribute the work between them, and synchronize and terminate the threads once they have

reached the end of the parallel region. The execution time consumed by this code is called the parallel overhead. We quantify this overhead for SB/Opt, SB/DL, Phi/Opt, and Phi/DL in order to understand the differences in their execution times and, later in this chapter, to evaluate their scalability, both in terms of problem size and the number of threads employed in the execution of LULESH 1.0. Accordingly, relevant performance data were collected for the execution of the three problem sizes by the following thread counts: 2, 4, 8, and 16 for the SB, and 60, 120, and 240 for the Phi.

First, for each architecture/code pair and each problem size, we decompose the total execution time into the parallel overhead and computation time, i.e., execution time without the parallel overhead. These measurements, which are listed in Tables 4.5 and 4.6, are used to: (1) compare the four architecture/code pairs, (2) quantify how much of the differences in execution times are accounted for by the overhead associated with the parallel constructs in LULESH (which includes the OpenMP overhead), and (3) quantify the percentage of the execution time that is consumed by the overhead. Next, we evaluate the scalability of total execution time, computation time, and overhead in terms of problem size. Finally, we provide and analyze data that is used to understand how execution time grows with the number of threads employed in the execution of LULESH 1.0.

Computation Time

Through inspection the computation data presented in Table 4.5, three main observations can be made:

1. SB/DL performs better than SB/Opt: For the three problem sizes SB/DL's computation times are 27%, 28%, and 17% *smaller* than those of SB/Opt.
2. Phi/DL performs better than Phi/Opt: Similar to SB/Opt vs. SB/DL, Phi/DL's computation times are 45%, 44%, and 66% *smaller* than those of Phi/Opt.
3. Phi/DL performs better than SB/DL: For all three problem sizes the computation times of Phi/DL are *smaller* than SB/DL's, with Phi/DL's computation times being 19%, 17%, and 18% *smaller* than SB/DL's.

Accordingly, for LULESH 1.0 executed on the Sandy Bridge and Xeon Phi, the optimizations in the DL code significantly reduce the computation time.

Table 4.5: Computation Time (sec): SB/Opt vs. SB/DL and Phi/Opt vs. Phi/DL.

	Architecture/Code Pair			
Problem Size	**SB/Opt**	**SB/DL**	**Phi/Opt**	**Phi/DL**
50^3 (a)	9.546 (b)	6.948 (c)	10.199 (d)	5.660 (e)
70^3 (2.7a)	30.792 (3.2b)	22.071 (3.2c)	32.508 (3.1d)	18.277 (3.2e)
90^3 (5.8a)	80.717 (8.5b)	66.737 (9.6c)	160.589 (15.8d)	54.813 (9.7e)

Parallel Overhead

Through inspection of the parallel overhead data presented in Table 4.6, three main observations in terms of computation time can be made:

1. SB/DL's overhead is larger (i.e., 12%, 18%, and 27% larger) than that of SB/Opt, while its computation time is smaller (i.e., 27%, 28%, and 17% smaller).
2. Phi/DL's overhead is smaller than that of Phi/Opt (i.e., 42%, 45%, and 52% smaller for the three problem sizes).
3. Phi/DL's overhead decreases with the problem size, while SB/DL's increases.
4. Accordingly, these data indicate that for LULESH 1.0 executed on the Sandy Bridge and Xeon Phi, in addition to the optimizations in the DL code significantly reducing the computation time, they significantly reduce the parallel overhead.

Table 4.6: Parallel Overhead (sec): SB/Opt vs. SB/DL and Phi/Opt vs. Phi/DL.

	Architecture/Code Pair			
Problem Size	**SB/Opt**	**SB/DL**	**Phi/Opt**	**Phi/DL**
50^3	3.534	3.972	16.681	9.630
(a)	(b)	(c)	(d)	(e)
70^3	22.318	26.249	50.242	27.543
(2.7a)	(6.3b)	(6.6c)	(3d)	(2.9e)
90^3	60.123	76.280	139.441	66.587
(5.8a)	(17.0b)	(19.2c)	(8.4d)	(6.9e)

SB/Opt vs. SB/DL: As discussed in Section 4.1.1, in terms of total execution time, SB/DL outperforms SB/Opt for all three problem sizes, and Phi/DL outperforms SB/DL for the two larger problem sizes. Let us examine why. Referring to Tables 4.5 and 4.6, across the three problem sizes, SB/DL's computation time is 27%, 28%, and 17% *smaller* than that SB/Opt, while SB/DL's parallel overhead is 12%, 18%, and 27% *larger* than that of SB/Opt. As can be seen, the difference between their computation times decreases as the problem size increases, while the difference between their parallel overheads increases with the problem size. This causes the differences between their total execution times to decrease (by 18%, 9%, and 2%) as the problem size increases, with SB/DL still performing better than SB/Opt at the 90^3 problem size. However, given these trends, it is not clear that SB/Opt would not perform better for larger problem sizes.

Phi/Opt vs. Phi/DL: For Phi/Opt vs. Phi/DL, the situation is just the opposite. The difference between their computation times increases with the problem size (staying relatively constant from the 50^3 to 70^3 problem size) – that of Phi/DL always being *smaller* (i.e., 45%, 44%, and 66% smaller). And, the difference between their parallel overheads decreases as the problem size increases – that of Phi/DL always being *smaller* (i.e., 42%, 45%, and 52% smaller). Thus, Phi/DL's total execution time is significantly smaller for all problem sizes.

Sandy Bridge vs. Xeon Phi: For SB/DL vs. Phi/DL the situation is similar to SB/Opt vs. SB/DL, with SB/DL replacing SB/Opt and Phi/DL replacing SB/DL, however, the differences in

parallel overhead are more extreme. Across the three problem sizes, Phi/DL's computation times are 19%, 17%, and 18% *smaller* than SB/DL's, while Phi/DL's parallel overhead is 142% and 5% *larger* than SB/DL's for the two smaller problem sizes and then 13% *smaller* for the largest problem size. In this case, the differences between the computation times of SB/DL and Phi/DL remain relatively constant, i.e., across the three problem sizes the computation time of SB/DL is 21-23% greater. In contrast the differences between the parallel overhead of SB/DL and Phi/DL decrease dramatically in favor of Phi/DL. The total execution time of SB/DL is smallest for the 50^3 problem size, while that of Phi/DL is smallest for the two larger problem sizes.

Difference in Execution Time Accounted for by Parallel Overhead

Having observed the differences in the computation times and the parallel overhead of the four architecture/code pairs under study, we now quantify how much of the differences between the execution times is attributable to parallel overhead. We begin with SB/DL vs. Phi/DL. For the 50^3 problem size the absolute difference between the total execution times of SB/DL and Phi/DL is 4.37 seconds, while the difference in overhead is 5.658 seconds (higher for Phi/DL) and the difference in computation time is 1.288 seconds (higher for SB/DL). For the 70^3 and 90^3 problem sizes, the differences in total execution times are 2.5 seconds and 21.617 seconds, respectively, while the differences in overhead are 1.294 (higher for Phi/DL) and 9.693 seconds (higher for SB/DL). And, unlike for the smallest problem size, the absolute differences in computation times are 3.794 (higher for SB/DL) and 11.924 seconds (higher for SB/DL).

In terms of SB/Opt vs. SB/DL, for the 50^3 problem size, the absolute difference between their total execution times is 2.16 seconds, while the difference in overhead is .438 seconds (higher for SB/DL) and the difference in computation time is 2.598 seconds (higher for SB/Opt). For the 70^3 and 90^3 problem sizes, the differences in total execution times are 4.790 seconds and 2.177 seconds, respectively, while the differences in overhead are 3.931 (higher for SB/DL) and 16.157 seconds (higher for SB/DL).

Finally, for Phi/Opt vs. Phi/DL the absolute difference in total execution times for the 50^3 problem size is 11.590 seconds, while the difference in overhead is 7.051 seconds, and the difference in computation time is 4.539 seconds (all higher for Phi/Opt). Regarding the larger problem sizes, the differences in total execution times are 36.930 seconds for the 70^3 problem size and 178.630 seconds for the 90^3 problem size. For these problem sizes, the differences in overhead are 22.699 and 72.854 seconds for the 70^3 and 90^3 problem sizes, respectively. In these cases, both the total execution time and overhead are higher for Phi/Opt. Finally, the difference in computation time is 14.231 seconds for the 70^3 problem size and 105.776 seconds for the 90^3 problem size. For all the problem sizes, the total execution time, the overhead, and the computation time is higher for Phi/Opt.

In summary, for the 50^3 problem size, the difference in parallel overhead accounts for the majority of the difference between the total execution times of SB/DL and Phi/DL, and for the 70^3 and 90^3 problem sizes, it is due to both the differences in computation time and parallel overhead. In addition, the difference between the total execution times of SB/DL and Phi/DL across the problem sizes follows the pattern of the differences between the overheads. The difference decreases from the 50^3 to 70^3 problem sizes and increases from the 70^3 to 90^3 problem size. In the case of SB/Opt and SB/DL, the differences in the total execution times are mainly due to differences in the computation time for all problem sizes except the 90^3 problem size. Phi/Opt and Phi/DL follow a similar pattern as SB/Opt and SB/DL except that the differences in execution time are mainly due to the overhead for all problem sizes except the 90^3 problem size, where the difference in computation time is more prominent.

Percentage of Execution Time Consumed by Parallel Overhead

Regardless of the cause of the differences in total execution times, as shown in Table 4.7, the overhead associated with parallelism consumes a significant portion of the execution time of LULESH 1.0. The percentage of Sandy Bridge execution time consumed by the overhead ranges from almost 27% to 63%, and that of the Xeon Phi ranges from almost 46% to 63%. However,

again, it should be noted that for the Xeon Phi the percentage of execution time consumed by the overhead decreases with the problem size, especially going from the 70^3 to the 90^3 problem size; while for the Sandy Bridge it increases.

Also note that:

1. Comparing SB/DL with SB/Opt, for all three problem sizes, the percentage of execution time consumed by the overhead is much *larger* for SB/DL (i.e., 34.63%, 29.72%, and 40.85% larger);
2. Comparing Phi/Opt with Phi/DL, the percentage is similar except for the 90^3 problem size, where it is 18% *larger* for Phi/DL; and
3. Comparing SB/DL with Phi/DL, the percentage of execution time consumed by the overhead for SB/DL is *smaller* for the 50^3 and 70^3 problem sizes but (10%) larger at the 90^3 problem size.

Table 4.7: Parallel Overhead (% of Total Execution Time/Execution Time w/o Overhead): SB/Opt vs. SB/DL and Phi/Opt vs. Phi/DL.

	Architecture/Code Pair			
Problem Size	**SB/Opt**	**SB/DL**	**Phi/Opt**	**Phi/DL**
50^3	27.018	36.373	62.057	62.982
(a)	(b)	(c)	(d)	(e)
70^3	42.022	54.323	60.715	60.111
(2.7a)	(1.6b)	(1.5c)	(.98d)	(.95e)
90^3	42.689	60.129	46.476	54.849
(5.8a)	(1.6b)	(1.65c)	(.75d)	(.87e)

4.1.3 Scalability

In this section we explore how the total execution time, computation time, and parallel overhead of SB/Opt, SB/DL, Phi/Opt, and Phi/DL grow with the problem size. In addition, we investigate how the total execution time and parallel overhead grow with the number of threads employed in the execution of LULESH 1.0.

Scalability w.r.t Problem Size

This section focuses on the scalability of the total execution time, computation time, and parallel overhead of LULESH 1.0 executed on the Sandy Bridge and Xeon Phi. Both the scalability of the full application and its four major phases of execution are addressed.

Scalability of Total Execution Time w.r.t. Problem Size: As the problem size is increased, an application will generally take longer to execute due to the fact that it is processing more data. This behavior is expressed in the total execution time of each architecture/code pair utilized in this study. Table 4.8 indicates the total execution time of each architecture/code pair and includes the growth of total execution time w.r.t. the problem size. From the 50^3 to the 70^3 problem size, the total execution times of Phi/Opt and Phi/DL only grow 10-15% *faster* than the problem size. The growths of SB/Opt and SB/DL, however, are much more significant. The growth of the total execution time is 50% *faster* for SB/Opt and 64% *faster* for SB/DL than the problem size. The difference between the growth in problem size and total execution time is more prominent when going from the 50^3 to the 90^3 problem size. The growth of the total execution time is 37%, 86%, 92%, and 126% faster than the problem size for Phi/DL, SB/Opt, Phi/Opt, and SB/DL, respectively. Although, SB/DL had the fastest growth, Phi/Opt had the largest execution time for each problem size, but this is mainly due to the fact that at the 50^3 problem size, the total execution time of Phi/Opt is up to 2.46x larger than that of the other architecture/code pairs. Nevertheless, the fast growth of SB/DL does end up affecting its performance, when compared to SB/Opt and Phi/DL. At the 50^3 problem size, SB/DL starts off with the shortest execution time, i.e., 10.92 seconds, but by the 90^3 problem size its execution time is 1.02x greater than SB/Opt's and 1.18x greater than Phi/DL's.

Table 4.8: Total Execution Time (sec): SB/Opt vs. SB/DL and Phi/Opt vs. Phi/DL.

	Architecture/Code Pair			
Problem Size	**SB/Opt**	**SB/DL**	**Phi/Opt**	**Phi/DL**
50^3	13.080	10.920	26.880	15.290
(a)	(b)	(c)	(d)	(e)
70^3	53.110	48.320	82.750	45.820
(2.7a)	(4.06b)	(4.42c)	(3.08d)	(3.00e)
90^3	140.840	143.017	300.030	121.400
(5.8a)	(10.77b)	(13.10c)	(11.16d)	(7.94e)

Scalability of Computation Time w.r.t. Problem Size: As expected, for each of the architecture/code pairs of interest, both the computation time and parallel overhead increase with the problem size. However, as shown in Table 4.5, in each case going from the 50^3 to the 70^3 problem size, the computation time grows relatively similar w.r.t. the problem size (15-19% *faster*), while going from the 50^3 to the 90^3 problem size it grows 47%, 66%, and 67% *faster* than the problem size for SB/Opt, SB/DL, and Phi/DL, and 172% *faster* for Phi/Opt. As a result, Phi/Opt's computation time at the 90^3 problem size is triple that of Phi/DL (with the lowest value), double that of SB/Opt (with values for the two smallest problem sizes being 5.6-7% smaller than those of Phi/Opt), and 1.8 that of SB/DL. Thus, Phi/Opt's computation time does not scale as well as that of SB/Opt, SB/DL, and Phi/DL as the problem size increases. Comparing SB/Opt, SB/DL, and Phi/DL, even though SB/Opt's computation time scales best with the problem size, its execution time for the smallest problem size is 37.4% and 69% *larger* than that of SB/DL and Phi/DL, respectively, causing SB/Opt to perform worse than SB/DL and Phi/DL across the three problem sizes. SB/DL's and Phi/DL's computation times grow almost identically with the problem size, however, SB/DL's computation time for the smallest problem size is 18.5% larger than that of Phi/DL.

Scalability of Parallel Overhead w.r.t. Problem Size: In terms of parallel overhead, as shown in Table 4.6, the overhead of SB/Opt and SB/DL grow much *faster* than the problem size (by 133% and 144%, respectively, from the 50^3 to the 70^3 problem size, and by 193% and 231% from the 70^3 to the 90^3 problem size), and 100% *faster* than the computation time in both cases. In addition, SB/DL's overhead, which is 12% larger for the 50^3 problem size, grows *faster* than that of SB/Opt, causing it to be 27% larger at the 90^3 problem size. Even though SB/Opt's parallel overhead and its computation time scale better than SB/DL's, SB/DL performs better because its computation time is 27% smaller than SB/Opt's for the smallest problem size.

In contrast, for Phi/Opt and Phi/DL, the growth of the parallel overhead from the 50^3 to the 70^3 problem size is similar to the growth of the problem size. However, from the 70^3 to the 90^3 problem size it grows *faster* than the problem size (by 45% and 19%, respectively, but *slower* than the computation time (by 47% and 29%). In addition, Phi/Opt's overhead, which is 73% larger for the 50^3 problem size, grows *faster* than that of Phi/DL's, causing it to be 109% larger at the 90^3 problem size. As a result, Phi/DL's parallel overhead and its computation time, as well as its total execution time, scale better than Phi/Opt's in terms of problem size.

Although Phi/DL's parallel overhead scales better than that of SB/Opt or SB/DL, at the 50^3 problem size it is 142%-172% larger. However, because of the faster rate at which the overhead of SB/Opt and SB/DL grow, at the 90^3 problem size the overhead of each of the three architecture/code pairs are within 10% of one another. Given this and the fact that Phi/DL's computation times are 21%-23% smaller than those of SB/DL, the total execution times of Phi/DL are smaller than those of SB/DL.

Scalability of Parallel Overhead w.r.t. Number of Threads

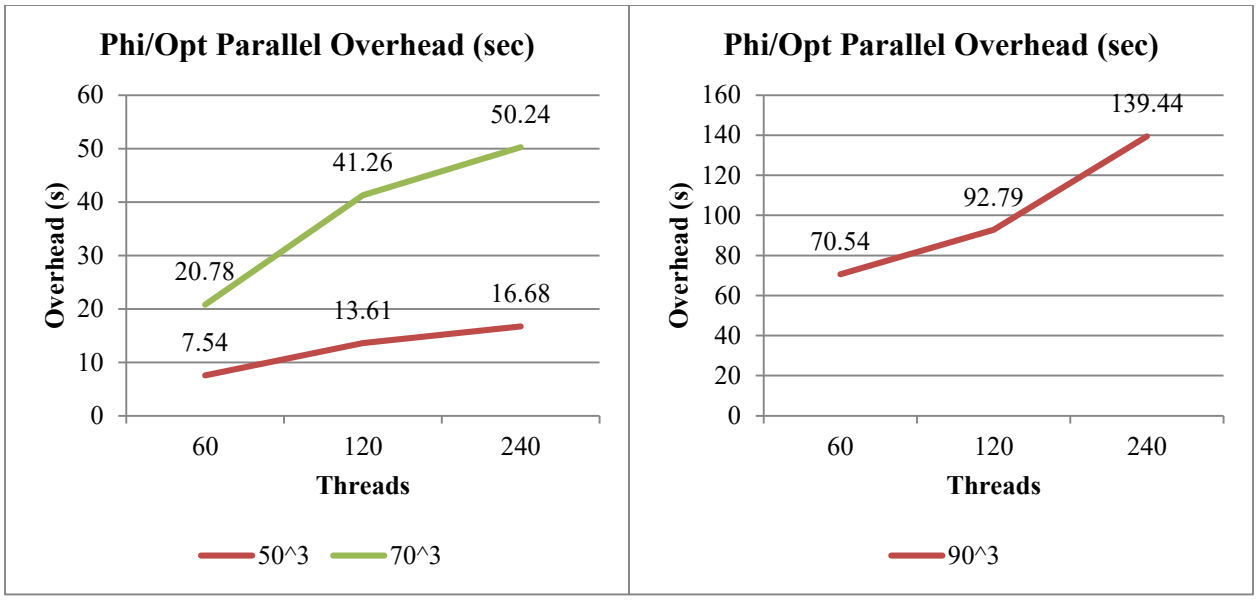
Due to the fact that the parallel overhead contributed significantly to the execution time of Phi/Opt, Phi/DL, SB/Opt, and SB/DL, separate experiments were conducted to measure, across the three problem sizes, the growth of the parallel overhead with the number of threads used to execute LULESH 1.0. This was done in terms of the full application as well as for each of the four

major phases of its execution. First, we focus on the scalability of the parallel overhead w.r.t. the number of threads employed to execute the full application.

Phi/Opt vs. Phi/DL: As shown in Figure 4.2a, as the number of threads increases from 60, to 120, to 240, the parallel overhead of Phi/Opt for the 50^3 and 70^3 problem sizes grows relatively slowly as compared to that for the 90^3 problem size.

As mentioned earlier, the data layout change introduced in the DL code significantly reduces the parallel overhead associated with executing LULESH on the Xeon Phi. This improvement in performance is manifested not only across all three problem sizes, but, as discussed below, across all three different numbers of threads used to execute the application. Figure 4.3 indicates that, similar to the Opt code, for all three problem sizes execution of the DL code results in a similar growth of the overhead on the Xeon Phi as the number of threads increases.

It is unclear if for larger problem sizes, the growth of the overhead of the DL code will behave similarly to that of the Opt code as the number of threads increases. Nonetheless, in comparison to the execution of the Opt code on the Xeon Phi, it is clear that execution of the DL code for the three problem sizes results in: (1) a reduction of the amount of time LULESH 1.0 devotes to parallel overhead activities and (2) improvement of the scalability of the parallel overhead with respect to the number of threads employed to execute the application.



(a) (b)

Figure 4.2: Phi/Opt Parallel Overhead: (a) 50^3, and 70^3 problem sizes and (b) 90^3 problem size.

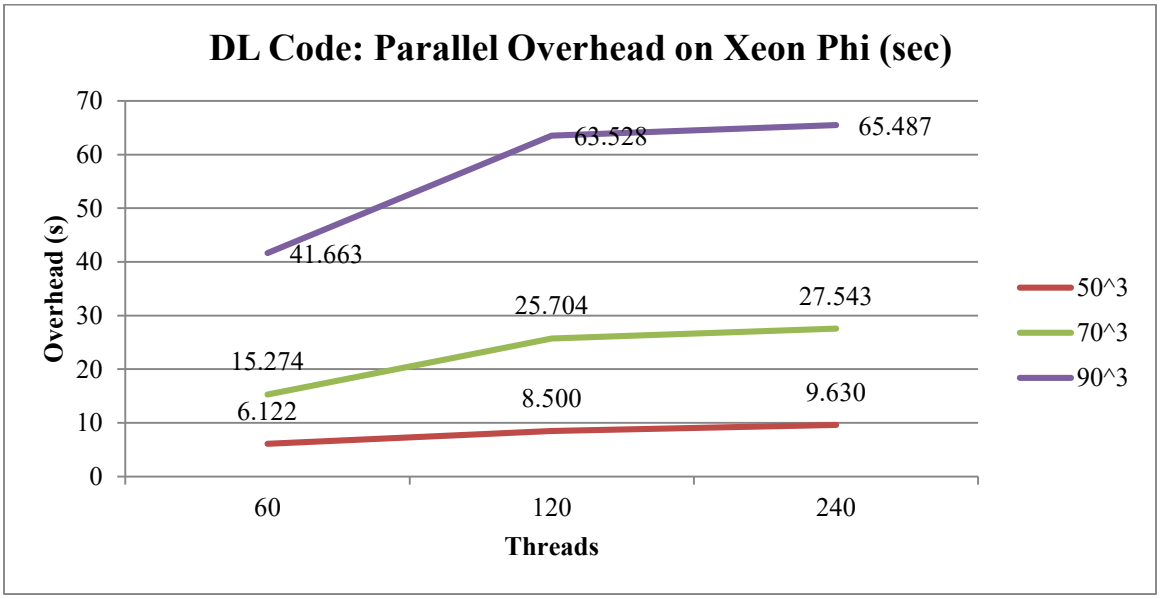


Figure 4.3: Phi/DL Parallel Overhead.

SB/Opt vs. SB/DL: In contrast to the positive impact on the parallel overhead of the DL code executed on the Xeon Phi, across all three problem sizes and all the numbers of threads, the parallel overhead of SB/DL is *larger* than that of SB/Opt (whereas, Phi/DL's overhead was *smaller*

than Phi/Opt's). In terms of the growth of the overhead w.r.t. the number of threads used to execute the application, as shown in Figures 4.4 and 4.5, like Phi/Opt and Phi/DL, the overhead of both SB/Opt and SB/DL increases with both the problem size and the number of threads, save for the executions by two threads. However, (1) akin to Phi/DL, the growth of SB/Opt's and SB/DL's overhead w.r.t. the number of threads is similar across the three problem sizes; but, in contrast to Phi/DL, the growth is not similar across the numbers of threads; and (2) for SB/DL, versus SB/Opt, the growth of the overhead increases more sharply from 8 to 16 threads. As a result, unlike the behavior of the DL code executed on the Xeon Phi, for all three problem sizes, the change in the data layout neither provides: (1) a reduction of the amount of time LULESH 1.0 devotes to parallel overhead activities on the Sandy Bridge nor (2) improvement of the scalability of the parallel overhead on the Sandy Bridge with respect to the number of threads employed to execute the application.

Sandy Bridge vs. Xeon Phi: Comparing the growth of the parallel overhead w.r.t. the number of threads employed to execute LULESH 1.0 on the Sandy Bridge with that on the Xeon Phi, there are several observations that can be made by inspecting Figures 4.2, 4.3, 4.4, and 4.5:

1. Whether executed on the Sandy Bridge or the Xeon Phi, the parallel overhead increases with both the problem size and the number of threads employed to execute the application, regardless of which code is executed.
2. The growth of the overhead w.r.t. the number of threads is similar across the three problem sizes for Phi/DL; for Phi/Opt it grows much faster for the 90^3 problem size than it does for the 50^3 and 70^3 problem sizes. For SB/Opt and SB/DL, excluding the two-thread executions, the overhead grows much faster than it does for Phi/Opt and Phi/DL.

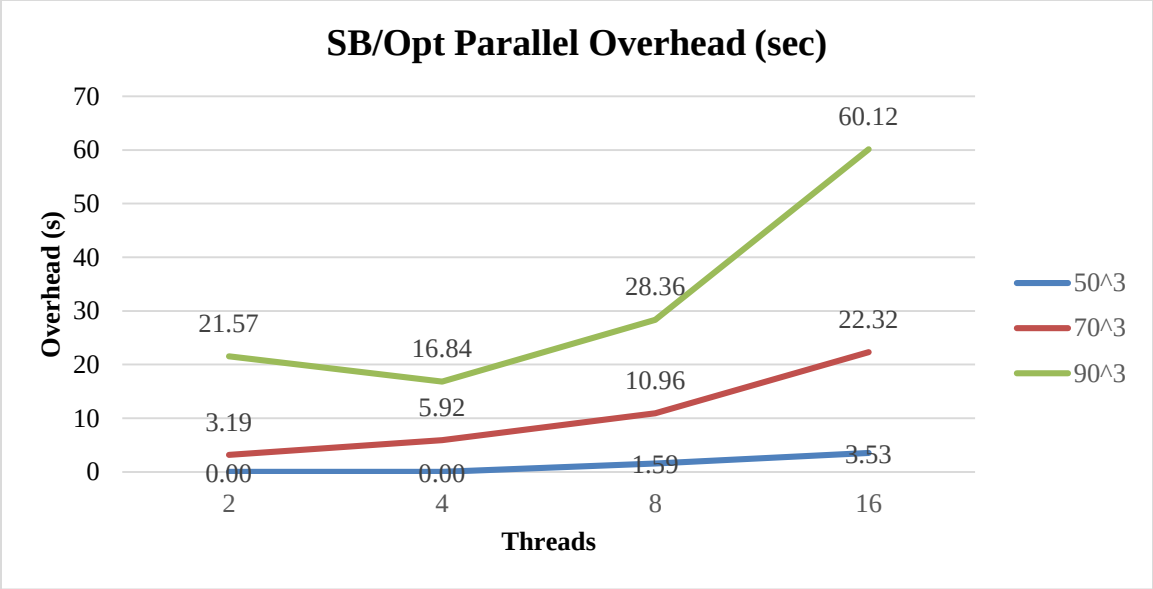


Figure 4.4: SB/Opt Parallel Overhead.

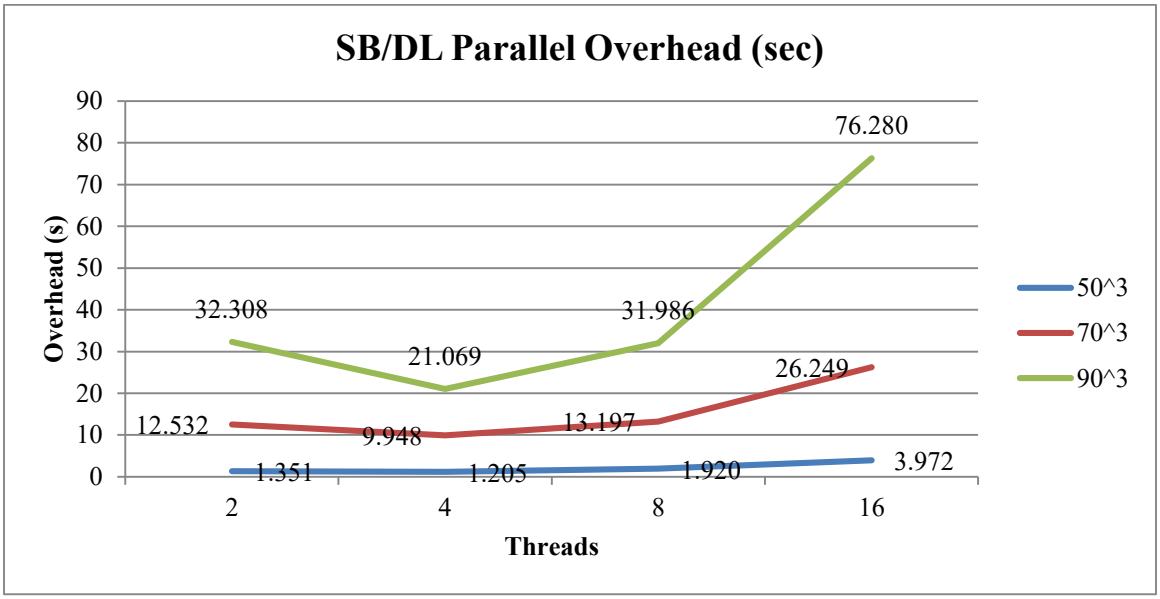


Figure 4.5: SB/DL Parallel Overhead.

Next, we focus on the scalability of the parallel overhead w.r.t. the number of threads employed to execute the four major phases of LULESH 1.0. This provides data that further explains the effect of the DL code on the parallel overhead. First, we consider Phi/Opt vs. Phi/DL.

Phi/Opt vs. Phi/DL: Tables 4.9 - 4.14 quantify the parallel overhead associated with each phase of the execution of Phi/Opt and Phi/DL. For both architecture/code pairs, the three problem

sizes, and the three thread counts, the largest portion of the overhead, i.e., ranging from 52%-74%, as well as the largest part of the execution time, is associated with the execution of Calc Volume Force. And, it is in this phase that is associated with the majority of the reduction of the overhead from executing the DL code (rather than the Opt code) on the Xeon Phi. Generally, the parallel overhead of each phase decreases when the DL code is executed. There are only eight exceptions to this behavior, which mainly occur during the Lagrange and Time Constraints phases. For example, when Lagrange and Time Constraints are executed as part the DL code, the overhead is from 1.03x larger for the 50^3 problem size to 1.22x larger for the 90^3 problem size, using 240 and 120 threads, respectively.

Table 4.9: Parallel Overhead by Phase (sec): Phi/Opt, 50^3 problem size.

Phase	Number of Threads		
	60	**120**	**240**
Calc Volume Force	4.617	9.051	12.208
Calc & Apply Accel	0.556	0.635	0.651
Lagrange	1.954	3.527	3.531
Time Constraints	0.416	0.401	0.291

Table 4.10: Parallel Overhead by Phase (sec): Phi/DL 50^3 problem size.

Phase	Number of Threads		
	60	**120**	**240**
Calc Volume Force	3.561	4.701	5.210
Calc & Apply Accel	0.359	0.550	0.470
Lagrange	1.734	2.928	3.658
Time Constraints	0.468	0.322	0.291

Table 4.11: Parallel Overhead by Phase (sec): Phi/Opt, 70^3 problem size.

Phase	Number of Threads		
	60	**120**	**240**
Calc Volume Force	12.768	26.894	37.275
Calc & Apply Accel	0.841	1.164	1.208
Lagrange	6.648	12.583	11.147
Time Constraints	0.526	0.623	0.611

Table 4.12: Parallel Overhead by Phase (sec): Phi/DL 70^3 problem size.

Phase	Number of Threads		
	60	**120**	**240**
Calc Volume Force	9.688	14.375	14.807
Calc & Apply Accel	0.962	1.157	1.073
Lagrange	4.117	9.555	11.066
Time Constraints	0.507	0.618	0.597

Table 4.13: Parallel Overhead by Phase (sec): Phi/Opt, 90^3 problem size.

Phase	Number of Threads		
	60	**120**	**240**
Calc Volume Force	38.100	58.233	103.263
Calc & Apply Accel	11.745	9.941	11.727
Lagrange	19.326	23.559	22.917
Time Constraints	1.374	1.053	1.533

Table 4.14: Parallel Overhead by Phase (sec): Phi/DL 90^3 problem size.

Phase	Number of Threads		
	60	**120**	**240**
Calc Volume Force	24.786	34.534	33.906
Calc & Apply Accel	2.129	2.560	2.294
Lagrange	13.836	25.148	27.663
Time Constraints	0.913	1.286	1.624

SB/Opt vs. SB/DL: Next we compare SB/Opt's and SB/DL's phases of execution in terms of the growth of the parallel overhead w.r.t. the number of threads, which is shown in Tables 4.15-4.20. Like the Xeon Phi, for both SB/Opt and SB/DL, the three problem sizes, and the thread counts, the largest portion of the overhead is associated with the execution of Calc Volume Force. For SB/Opt and SB/DL it accounts for 12%-97% of the total overhead. And, like on the Xeon Phi, it is in this phase that we see the majority of the reduction of the overhead that results from executing the DL code (rather than the Opt code) on the Sandy Bridge. For Phi/Opt the parallel overhead is from 1.30x (for the 50^3 problem size, using 60 threads) to 3.05x (for the 90^3 problem size, using 240 threads) larger.

Sandy Bridge vs. Xeon Phi: For all three problem sizes, the parallel overhead associated with execution of the Opt code is larger on the Xeon Phi than it is on the Sandy Bridge. It ranges

from 1.05x larger for Calc & Apply Accel at the 70^3 problem size to 20.8x larger for Lagrange at the 50^3 problem size. However, it is important to note that this may be due to the fact that the maximum number of threads that were employed on the Sandy Bridge was 16 (equal to the number of available cores), which is much smaller than the number employed on the Xeon Phi, i.e., 240 (executed on 60 cores).

In contrast, when the DL code is executed on either the Sandy Bridge or Xeon Phi, the majority of the parallel overhead is associated with the execution of Calc Volume Force. It accounts for 75%-84% of the overhead for the 70^3 and 90^3 problem sizes depending on the number of threads used.

Execution of LULESH 1.0 on the Xeon Phi with 60 or 120 threads results in a parallel overhead that is larger than that which results from executing it on the Sandy Bridge with two, four, or eight threads. However, when SB/DL is executed with 16 threads, the parallel overhead exceeds that of the Phi/DL executed with 60, 120, or 240 threads. This is due to the larger parallel overhead associated with the execution of Calc & Apply Accel, Lagrange, and Time Constraints on the Sandy Bridge. Like the Xeon Phi, when the DL code, in comparison to the Opt code, is executed on the Sandy Bridge, the parallel overhead associated with the execution of Calc Volume Force is reduced (for the Sandy Bridge from 4% to 87% dependent on the problem size and number of threads). However, the overhead of the remaining phases increases and offsets the reduction in the overhead associated with Calc Volume Force. Additionally, as the problem size and the number of threads increases, the reduction in the overhead associated with Calc Volume Force decreases until at the 90^3 problem size there is no reduction in the overhead when 16 threads are employed.

Table 4.15: Parallel Overhead by Phase (sec): SB/Opt, 50^3 problem size.

Phase	Number of Threads			
	2	**4**	**8**	**16**
Calc Volume Force	1.209	1.139	1.995	3.381
Calc & Apply Accel	0.112	0.148	0.174	0.282
Lagrange	< 0.001	< 0.001	< 0.001	< 0.001
Time Constraints	0.034	0.038	0.036	0.040

Table 4.16: Parallel Overhead by Phase (sec): SB/DL, 50^3 problem size.

Phase	Number of Threads			
	2	**4**	**8**	**16**
Calc Volume Force	0.154	0.469	1.392	3.215
Calc & Apply Accel	0.297	0.194	0.177	0.339
Lagrange	0.846	0.498	0.313	0.379
Time Constraints	0.045	0.044	0.410	0.038

Table 4.17: Parallel Overhead by Phase (sec): SB/Opt, 70^3 problem size.

Phase	Number of Threads			
	2	**4**	**8**	**16**
Calc Volume Force	11.979	9.866	12.305	20.358
Calc & Apply Accel	0.394	0.434	0.365	1.149
Lagrange	< 0.001	< 0.001	< 0.001	0.715
Time Constraints	0.029	0.060	0.056	0.096

Table 4.18: Parallel Overhead by Phase (sec): SB/DL, 70^3 problem size.

Phase	Number of Threads			
	2	**4**	**8**	**16**
Calc Volume Force	9.352	7.972	11.363	22.036
Calc & Apply Accel	0.613	0.425	0.541	1.315
Lagrange	2.489	1.469	1.225	2.792
Time Constraints	0.077	0.082	0.067	0.106

Table 4.19: Parallel Overhead by Phase (sec): SB/Opt 90^3 problem size.

Phase	Number of Threads			
	2	**4**	**8**	**16**
Calc Volume Force	30.993	20.506	27.827	53.734
Calc & Apply Accel	3.846	2.768	2.341	3.920
Lagrange	< 0.001	< 0.001	< 0.001	2.127
Time Constraints	0.048	0.092	0.097	0.342

Table 4.20: Parallel Overhead by Phase (sec): SB/DL, 90^3 problem size.

Phase	Number of Threads			
	2	**4**	**8**	**16**
Calc Volume Force	24.625	16.924	26.226	62.217
Calc & Apply Accel	1.094	1.231	3.218	5.417
Lagrange	6.475	2.791	2.431	8.213
Time Constraints	0.114	0.124	0.111	0.434

4.2 Accelerator Comparison

Of all of the accelerators under comparison, i.e., the Intel Xeon Phi and the NVIDIA Kepler and Fermi GPGPUs, the Kepler is argued to be the fastest and most efficient for use in HPC systems. However, the GPU architecture has some drawbacks that prevent it from being the processing unit of choice for some applications. Specifically, both the Kepler and Fermi require that data be transmitted to and from the device and host processor, while the Xeon Phi (which is also a many-core architecture) does not necessarily require such data transfers. Furthermore, the memory hierarchy of the GPUs is much simpler than that of a standard processor. For example, the Sandy Bridge architecture has three levels of cache, while both the Fermi and Kepler only offer two levels of cache, with the real estate of the level-1 cache being shared by another memory resource, i.e., the texture cache, which may not necessarily be used by an application. Although it is reasonable to expect that the Kepler's performance would exceed that of the Fermi, the Fermi is included in this study because it can be helpful to quantify the performance differences between an architecture and its predecessor. Such information can be useful to help determine if the performance gained through the use of the Kepler, instead of the Fermi, will offset the cost of upgrading to the new architecture.

With this in mind, referring to Figure 4.6, the accelerator/code pair that provides best performance for the three problem sizes of LULESH 1.0 is the NVIDIA Kepler GPU and the code optimized for it (KGPU/K). The Kepler execution times for the 50^3, 70^3, and 90^3 problem sizes are 2.420, 7.302, and 17.022 seconds, respectively. In comparison, the NVIDIA Fermi GPU and the code optimized for it (FGPU/F) is about seven times slower for the three problem sizes.

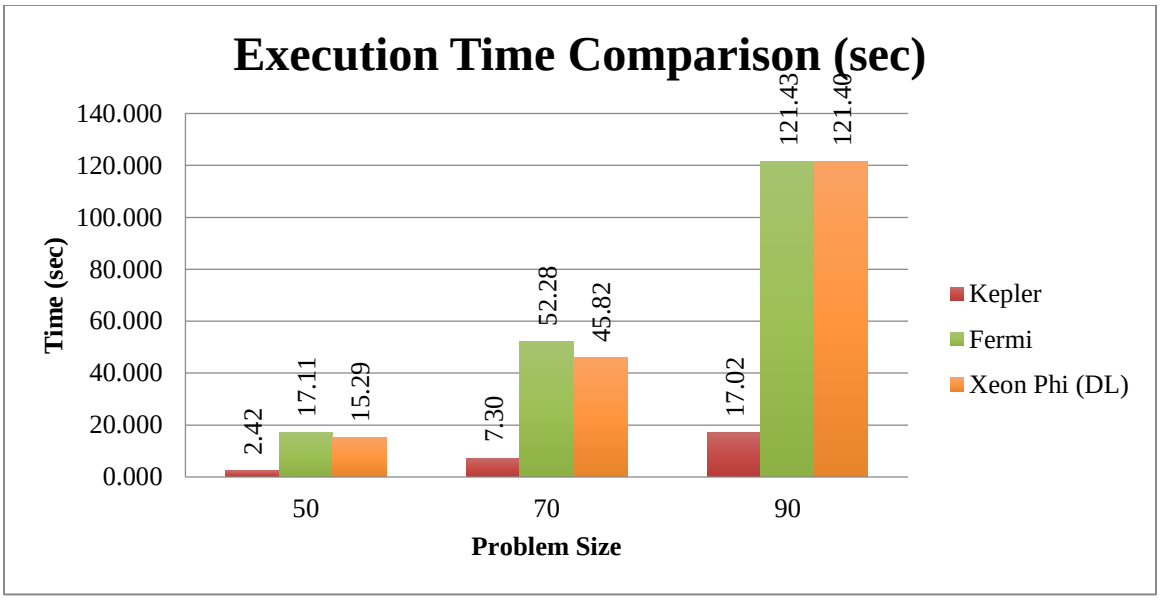


Figure 4.6: LULESH 1.0 execution times across different accelerator/code pairs.

It is common to also measure the speedup attained by a parallel application, over the best sequential code, executed on a specific computer architecture. This indicates how well the parallel code performs in comparison to the sequential code. The codes for the accelerators are parallel codes by definition, and there is only one sequential code available for LULESH 1.0 [14], which was compiled with the Intel compiler and the O3 level of optimization and executed on one core of the Sandy Bridge. Table 4.21 presents the speedups attained. Note the exceptionally larger speedup provided by KGPU/K, which, like FGPU/F and unlike Phi/DL, increases with the problem size. The increase of speedup with problem size indicates that the GPUs can solve larger problem sizes with potentially larger speedups.

Table 4.21: Speedup of architecture/code over best serial version of LULESH (SB=Sandy Bridge, Phi=Xeon Phi, FGPU=Fermi, KGPU=Kepler.

	Problem Size		
Architecture/Code	**50^3**	**70^3**	**90^3**
Phi/DL	13.778	14.735	14.628
FGPU/F	12.287	12.744	14.321
KGPU/K	86.836	91.253	102.159

4.2.1 Distribution and Percentage of Execution Time

To understand if the accelerators perform similarly on all phases of LULESH's execution or if particular phases favor different accelerators, we collected the data shown in Figures 4.7-4.9. Although the accelerators under study are very different in their architectural design, the distribution of LULESH's execution time among its phases of execution has the same characteristics across the accelerator/code pairs. For example, for each problem size it is evident that Calc Volume Force is the most time-consuming phase, with the next being the Lagrange phase. In contrast, the other two phases, Calc & Apply Accel and Time Constraints, consume significantly less time, i.e., together they consume from 5.07% to 14.61% of the total execution time, depending on the accelerator/code pair and problem size.

Comparing the accelerator/code pairs in terms of the three problem sizes, KGPU/K performs best for only the two dominant phases, Calc Volume Force and Lagrange, but note that together these phases represent over 85% of its execution time. And, KGPU/K performs best in terms of total solve time. Of interest is the fact that although KGPU/K, in comparison to FGPU/F and Phi/DL, spends approximately the same time in Time Constraints, it spends about twice as much time executing the Calc & Apply Accel phase – this may represent an opportunity for further code optimization.

The distribution of execution time for both KGPU/K and FGPU/F remains essentially the same across the three problem sizes; when the percentage consumed by each phase is rounded to the nearest whole percentage, it changes by at most 2% from one problem size to the next. This is also true for Phi/DL, however, for this architecture the percentage consumed by each phase changes by at most 5%. Thus, this data does not reveal anything significant.

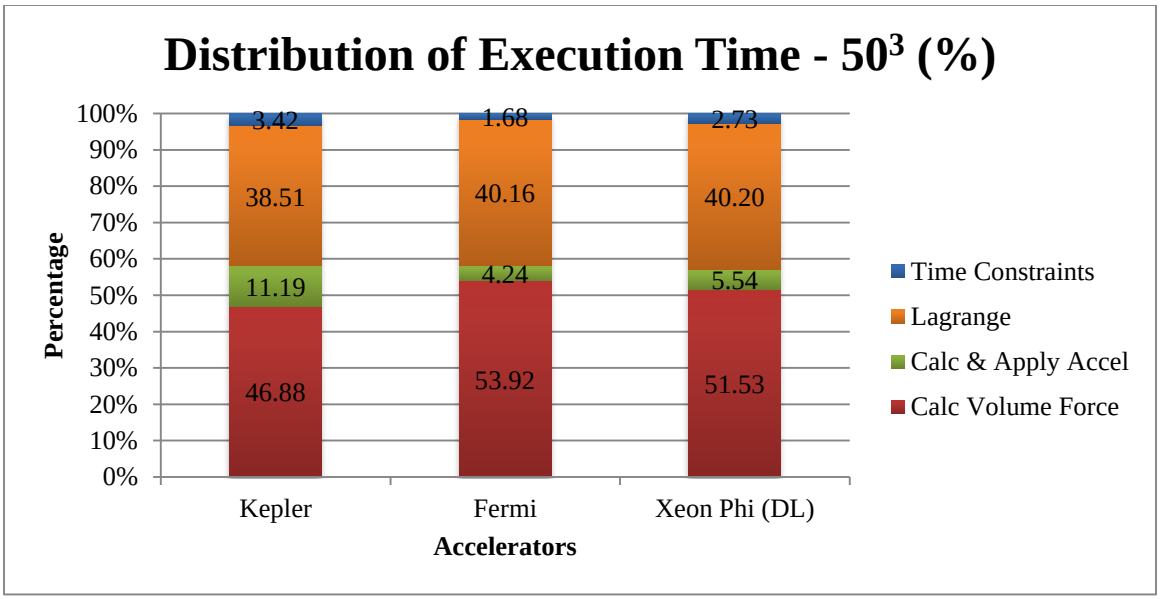


Figure 4.7: Percentage of execution time by phase of LULESH 1.0 – 50^3 problem size.

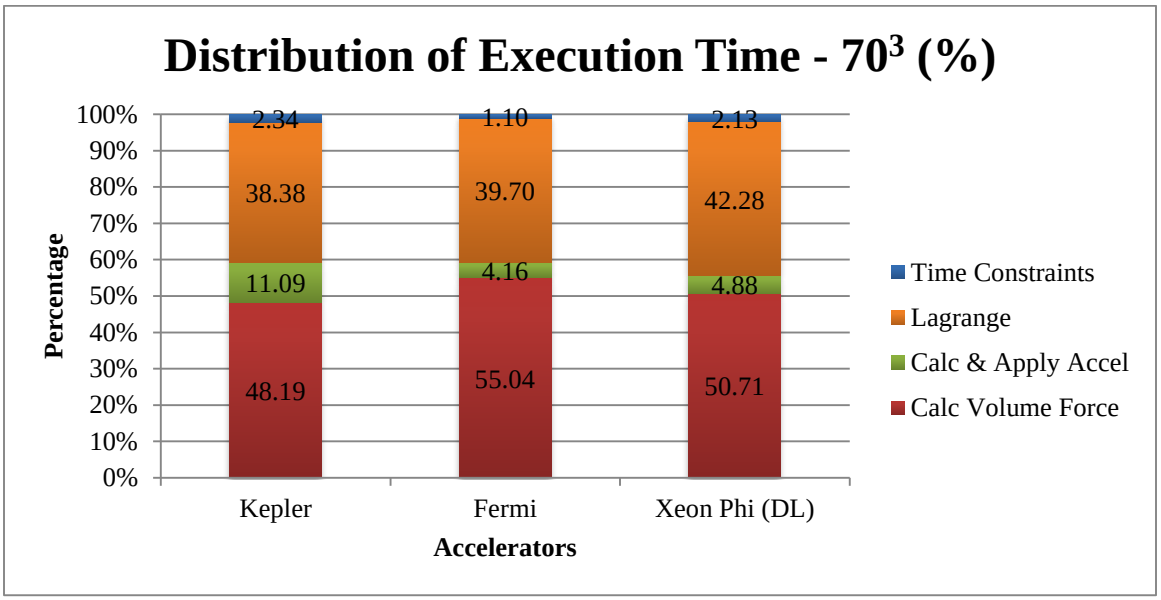


Figure 4.8: Percentage of execution time by phase of LULESH 1.0 – 70^3 problem size.

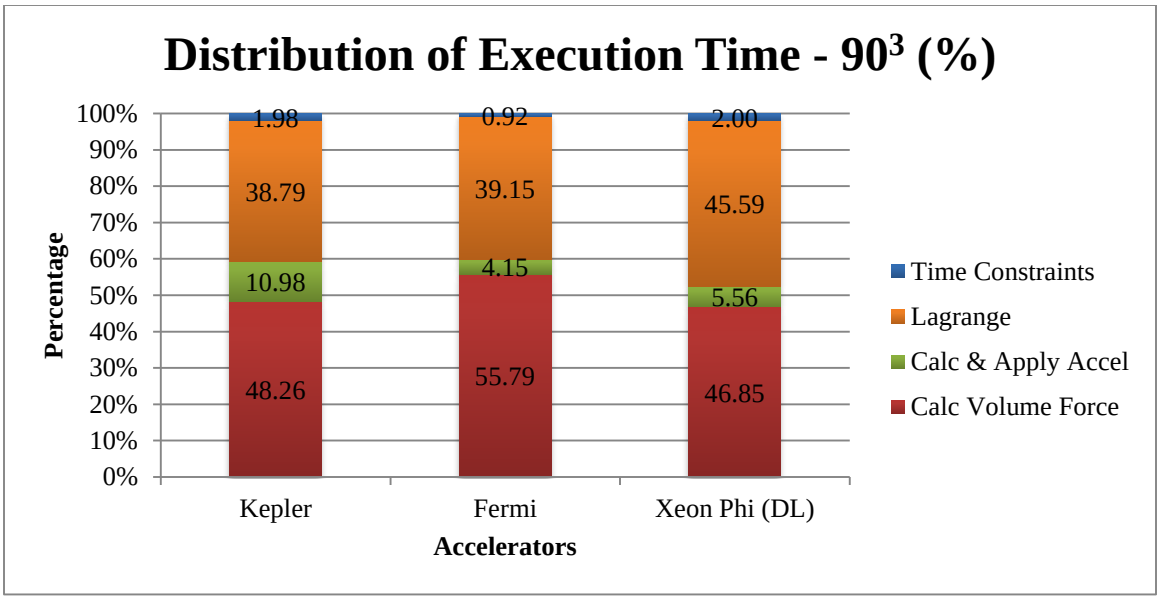


Figure 4.9: Percentage of execution time by phase of LULESH 1.0 – 90^3 problem size.

4.2.2 Power and Energy Consumption

As shown in Figure 4.10, for all the architecture/code pairs, the average power draw increases with the problem size. Although Phi/DL is comparable to FGPU/F in terms of execution time, its average power draw is 21.8%, 14.5%, and 22.7% higher for the 50^3, 70^3, and 90^3 problem sizes, respectively. The Xeon Phi has the highest power draw among all the architecture/code pairs studied.

Although the KGPU/K performs best of all the accelerator/code pairs under study, not only in terms of execution time, but also in terms of power consumption, the rate at which its power consumption increases with problem size is greater than the rates associated with FGPU/K and Phi/DL. For the 50^3 and 70^3 problem sizes, KGPU/K has the smallest power draw, consuming an average of less than 64 W for the 50^3 problem size and 89 W for the 70^3 problem size; while the FGPU/K and Phi/DL consume over 100 W for both problem sizes. However, unlike FGPU/K and Phi/DL, the percentage change in the power consumption of KGPU/K maps to the percentage change in problem size. Specifically, the 70^3 problem size is 40% larger than the 50^3 problem size, and the 90^3 problem size is 28.5% larger than the 70^3 problem size. Similarly, KGPU/K uses 39.06% more power to execute the 70^3 problem size as compared to that used to execute the 50^3

problem size; and 29.21% more power for the 90^3 problem size than for the 70^3 problem size. In contrast, the power consumption of the other accelerator/code pairs grows less than 12% from one problem size to the next.

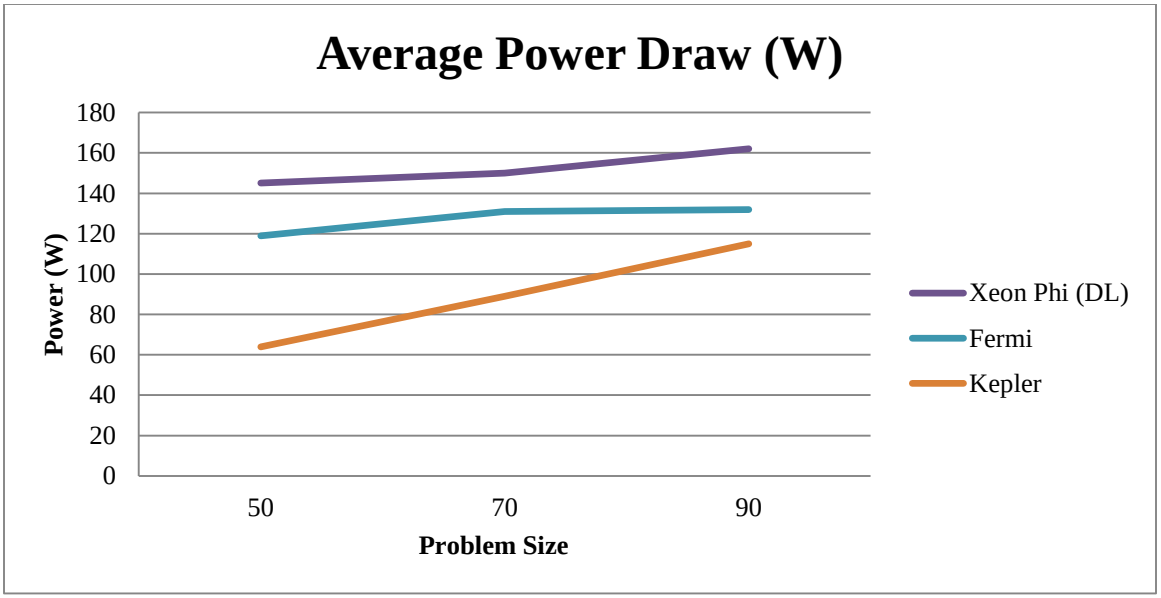


Figure 4.10: Average Power Draw of Architecture/LULESH 1.0 Codes.

For each of the architecture/code pairs studied, the average energy consumption tracks execution-time performance. As shown in Figure 4.11, KGPU/K (which had the best execution-time performance) consumes the least amount of energy for all problem sizes; it consumes 154.88, 649.88, and 1,957.53 Joules for the 50^3, 70^3, and 90^3 problem sizes, respectively. In addition, the employment of the DL code allowed the Xeon Phi to consume levels of energy consumption similar to those consumed by the Fermi GPU. For the 50^3 and 70^3 problem sizes, there is less than 10% difference between the energy consumption of the Xeon Phi and the Fermi. At the 90^3 problem size, however, there is a 20% difference between the energy consumption of the two architectures, with the Xeon Phi's energy consumption being 3,639.63 J higher than the Fermi's.

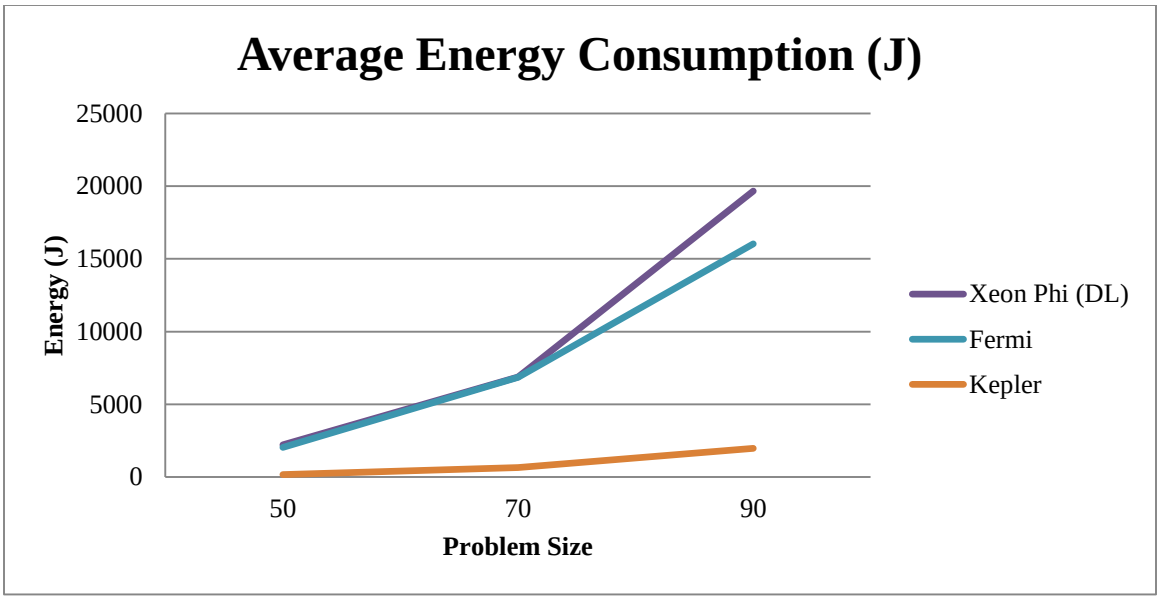


Figure 4.11: Average Energy Consumption of Architecture/LULESH 1.0 Codes.

4.3 Accelerator Performance Analysis

Although looking at runtime data alone is sufficient to determine on which architecture a code will perform best, it is necessary to look at additional performance metrics to understand why an application benefits more from the use of one architecture over another. However, the architectural design of the accelerators observed in this study is quite different and the metrics available for each device are not always comparable. Nonetheless, in this section, the runtime performance of LULESH is characterized through observation of its memory performance, vectorization capability, and its IPC.

4.3.1 Memory Performance

Regardless of the processing unit utilized, if an application exhibits poor memory behavior, its performance will be negatively affected. As a result, for each of the accelerators under study we attempted to determine the number of misses at each level of its memory hierarchy, both for the entire application and for each of its four execution phases. In addition, we attempted to compare the memory performance of the accelerators in an effort to understand its impact on the execution time of LULESH 1.0.

However, comparing the memory performance of these architectures is not a trivial task. This is because the memory hierarchy of the Xeon Phi is quite different from that of the GPUs and because some relevant performance data is not accessible. The Xeon Phi's memory hierarchy has the design of a standard processor, i.e., it consists of two levels of cache, an L1 data translation lookaside buffer (TLB), and an L2 TLB that acts as a true second-level TLB by behaving as a cache for page directory entries. In contrast, both GPUs have a texture memory, a constant memory, and an L1 data cache per SM/SMX, and a unified L2 cache that services all operations; in addition, the Kepler has a read-only data cache per SM. Additionally, the L1 cache of the GPUs can be configured so that part of it can be devoted to shared memory. Finally, NVIDIA GPUs have a level of abstraction in their main memory. Specifically, a distinction is made between global and local memory. Although both types of memory are placed in the same physical location (i.e., they reside in main memory), local memory is visible only to the thread that wrote it, while global memory is visible to all threads within the application.

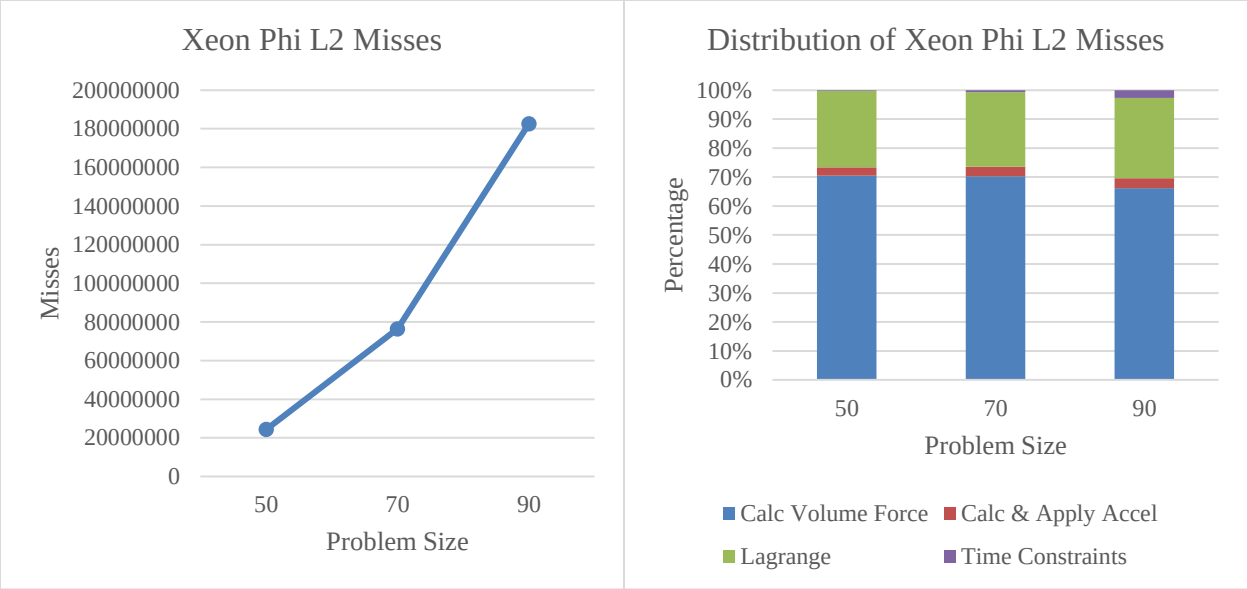
Xeon Phi (DL Code)

First, we examine the data collected regarding Phi/DL's memory performance. Because there are no performance events that allow the counting of L2 TLB misses on the Xeon Phi, we could only collect performance data for the L1 data caches and unified L2 caches (see Figures 4.12 and 4.13), and the L1 data TLB (see Figure 4.14). Figure 4.12 (a) graphs the number of L1 data cache misses generated by Phi/DL for the three problem sizes and Figure 4.12 (b) displays a histogram of the number of L1 data cache misses attributable to each phase of the execution of LULESH 1.0. As shown, (1) the number of L1 data cache misses exhibits a linear growth as the problem size increases, and (2) almost all of the L1 data cache misses are attributable to Lagrange and Calc Volume Force, i.e., essentially, for all the problem sizes, Calc Volume Force generates over 70% of the L1 data cache misses, while Lagrange generates 27%. The remaining phases, Time Constraints and Calc & Apply Accel, generate less than 1% of the misses to the L1 data cache.

As shown in Figure 4.13 (b), Lagrange and Calc Volume Force also generate the vast majority of L2-cache misses; Calc Volume Force generates 66%-70% of the misses to the L2 cache, and Lagrange generates 26%-27%. As was the case for the L1 data cache, the number of misses that resulted from the execution of Time Constraints and Calc & Apply Accel was minimal (ranging from 3%-6%). However, as shown in Figure 4.13 (a), as the problem size increases, the number of L2-cache misses grows much faster than does the number of L1-cache misses.

(a) (b)

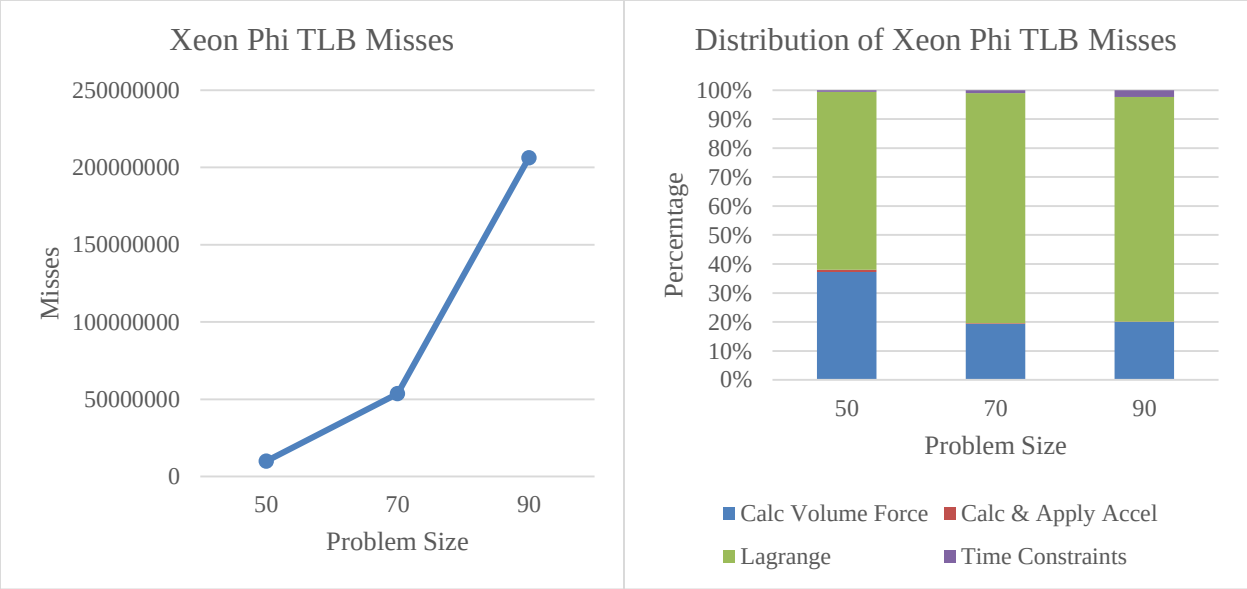
Figure 4.12: Phi/DL L1-Cache Misses: (a) Number of L1-cache misses across problem sizes; (b) Distribution of L1-cache misses.



(a) (b)

Figure 4.13: Phi/DL L2-Cache Misses: (a) Number of L2-cache misses across problem sizes; (b) Distribution of L2-cache misses.

In terms of TLB behavior, as shown in Figure 4.14, the growth of the L1-TLB misses is similar to that of the L2-cache misses, and the L1-TLB misses are mainly generated by Calc Volume Force and Lagrange. However, unlike the behavior of the caches, it is Lagrange, rather than Calc Volume Force, that generates over 60% of L1-TLB misses. Specifically, Lagrange generates 61.42%, 79.38%, and 77.53% for the 50^3, 70^3, and 90^3 problem sizes, respectively, while Calc Volume Force generates 37.24%, 19.33%, and 20.02%. As was the case with the L1 and L2 caches, Time Constraints and Calc & Apply Accel generate a small number of L1-TLB misses, i.e., only 1%-3%.



(a) (b)

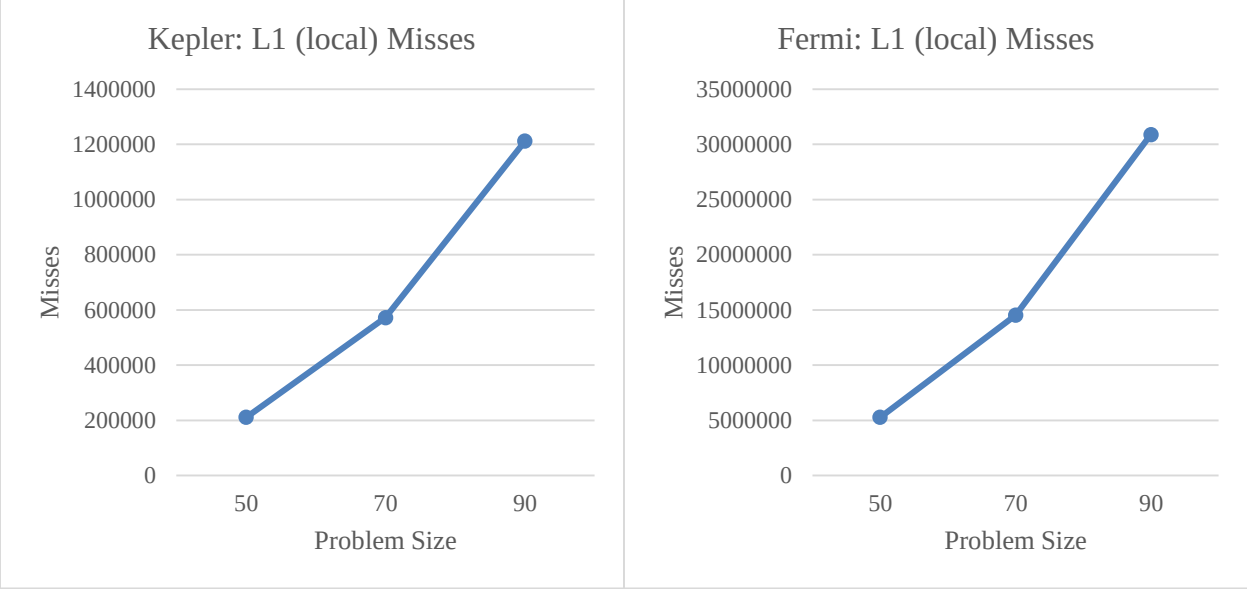
Figure 4.14: Phi/DL L1-TLB Misses: (a) Number of L1-TLB misses across problem sizes; (b) Distribution of L1-TLB misses.

Fermi and Kepler GPUs (CUDA Code)

On the Fermi, the L1 cache stores data that is resident in either local or global memory, while on the Kepler it stores only data that is resident in local memory. Thus, the L1-cache misses generated by FGPU/F were due to accesses to both local and global memory, while those generated by KGPU/K were due to accesses to local memory (note, again, that global memory transactions are not cached at the L1 cache). Also, unlike FGPU/F, KGPU/K made use of the available texture memory. Thus, KGPU/K L2-cache misses were the result of L1-cache misses and accesses to texture memory, while FGPU/F L2-cache misses were the result of only L1-cache misses.

Figure 4.15 (a) and (b) graph the number of L1-cache misses generated by KGPU/K and FGPU/F as a result of local memory transactions, respectively. For both architecture/code pairs, the number of L1-cache misses grows similarly as the problem size increases. For FGPU/F these misses were generated by only Calc Volume Force and Lagrange, while for KGPU/K they were generated by only Lagrange. For FGPU/F 60% of the misses were generated by Calc Volume Force, while 40% were generated by Lagrange. This behavior is very similar to that of Phi/DL, for which over 70%

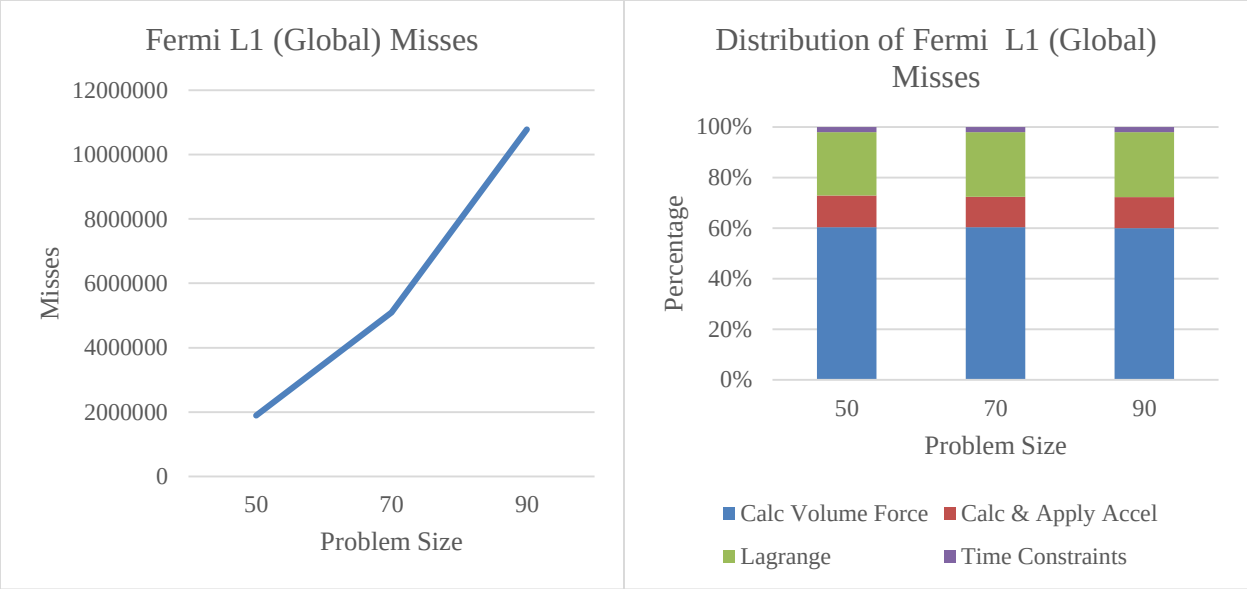
of L1-cache misses were attributable to Calc Volume Force and 27% of the misses were attributed to Lagrange.



(a) (b)

Figure 4.15: GPU L1-Cache (local) Misses: (a) KGPU/K L1-cache (local) misses; (b) FGPU/F L1-cache (local) misses.

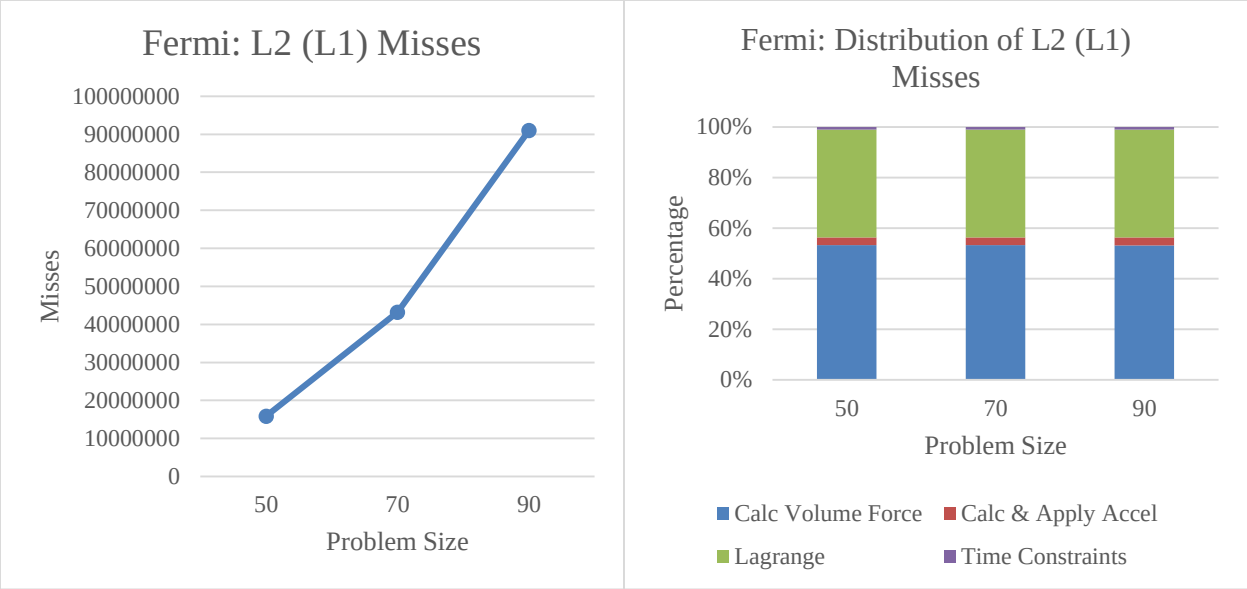
As shown in Figure 4.16, the number of L1-cache misses generated by FGPU/F due to global memory transactions and the number due to local memory transactions grow similarly as the problem size increases. As was the case with local memory transactions, 60% of the L1-cache misses generated by accesses to global memory in FGPU/F were due to Calc Volume Force, however, only 25% of the L1-cache misses (as compared to the 40% of the L1-cache misses generated by local memory transactions) were generated by Lagrange; the remaining 15% were generated by Time Constraints and Calc & Apply Accel. Thus, in terms of the distribution of L1-cache misses among the execution phases of LULESH, for all three problem sizes FGPU/F and Phi/DL behave similarly.



(a) (b)

Figure 4.16: FGPU/F L1-Cache (Global) Misses: (a) FGPU/F L1-cache (Global) misses; (b) FGPU/F Distribution of L1-cache (Global) misses.

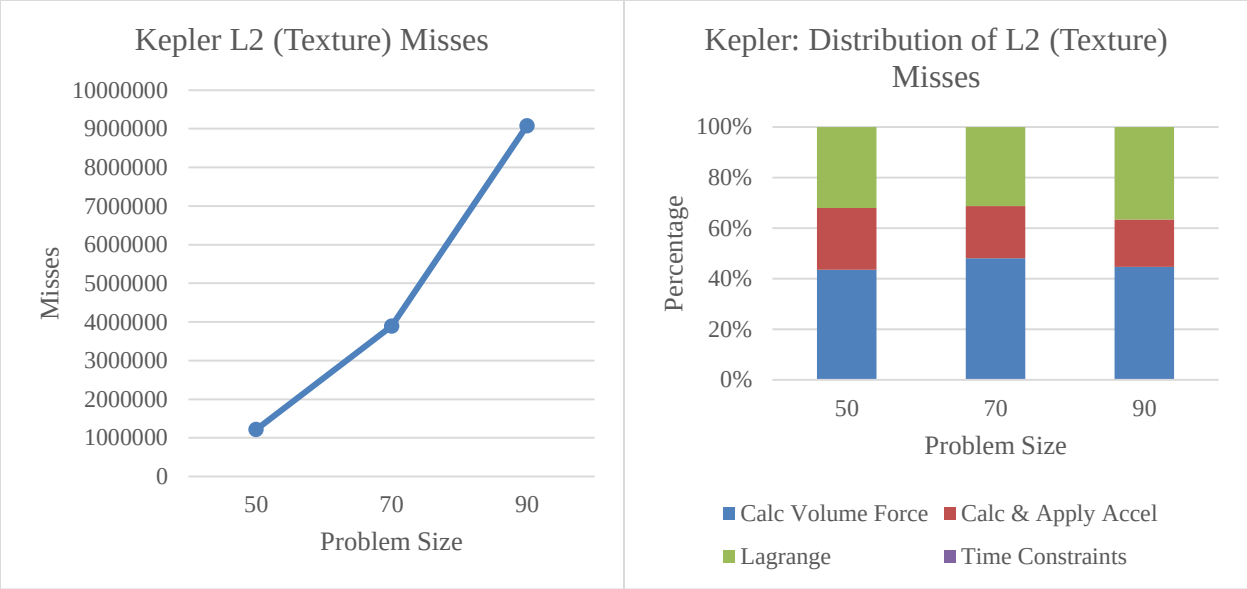
As shown in Figure 4.17b, like Phi/DL, regardless of problem size, each phase of FGPU/F generates L2-cache misses. And, also like Phi/DL, over 90% of FGPU/F L2-cache misses (due to L1-cache misses) were attributable to the two most time-consuming phases of LULESH. However, only 53% (vs. 66%-70% for Phi/DL) of the misses were generated by Calc Volume Force, while 43% (vs. 26%-27% for Phi/DL) were generated by Lagrange. Finally, 4% (3%-6% for Phi/DL) of the L2-cache misses were generated by Calc & Apply Accel and Time Constraints. Also, the growth rate of FGPU/F L2-cache misses is similar to that of Phi/DL, since, as shown in Figures 4.13a and 4.17a.



(a) (b)

Figure 4.17: Fermi L2 (L1) Cache Misses: (a) L2-cache misses across problem sizes; (b) Distribution of L2-cache misses.

Despite the fact that LULESH made use of the Kepler's texture memory, while it did not make use of the Fermi's texture memory, the L2-cache behavior of KGPU/K, which is illustrated in Figure 4.18, was similar to that of FGPU/F. However, the distribution of KGPU/K L2-cache misses across the four phases of execution of LULESH mirrors the distribution of the execution time (shown in Section 4.2.1): (1) 40% of the execution time of KGPU/K was associated with the execution of Calc Volume Force, and this phase generated from 43% to 49% of L2-cache misses; (2) Lagrange generated 31%-37% of the L2-cache misses and consumed about 38%-39% of the execution time; and (3) Calc & Apply Accel generated 20% of the L2-cache misses and consumed 10% of the execution time. In comparison, in the case of both Phi/DL and FGPU/F Calc & Apply Accel consumed less than 6% of the execution time and generated only up to 4% of the misses in all of the observed levels of the memory hierarchy.



(a) (b)

Figure 4.18: Kepler L2-Cache (Texture) Misses: (a) L2-cache misses across problem sizes; (b) Distribution of L2-cache misses.

Effect of Memory Performance on Execution Time

From the memory behavior observed, regardless of the architecture and the problem size, it is clear that most of the misses at all levels of the memory hierarchy are attributable to Calc Volume Force and Lagrange. Furthermore, while the Xeon Phi and Fermi's memory behavior is comparable (as is their execution time), the Kepler stands out. KGPU/K devoted more time to the Calc & Apply Accel than did FGPU/F and Phi/DL, but it also generated more L2-cache misses on KGPU/K than it did on FGPU/F and Phi/DL.

4.3.2 Vectorization Usage

A metric of vectorization usage gauges the ability of the architecture/code pair to simultaneously compute one operation over multiple pairs of operands. For the Phi/DL, the vectorization reports were generated at compile time to determine the number of loops that were effectively vectorized for the DL code. Since loop vectorization is performed at compile time, the improvement in performance is reflected in all three problem sizes. Regrettably, hardware event

counts could not be used for this purpose due to limitations in the number of reliable hardware counters available on the device.

Due to the fact that the design of GPU architectures is different than that of the studied Intel architectures, and considering that the GPU architecture can be loosely classified as an SIMD processor (a class of computers with multiple processing elements that perform the same operation in parallel), the vectorization usage of KGPU/K and FGPU/F was measured by determining the occupancy of KGPU/K and FGPU/F. The occupancy is the ratio of the number of active warps over the maximum number of warps that can be active during the execution of a GPU kernel. In this case, a warp is a group of 32 threads. The occupancy of a GPU can range from zero to one, which can be translated to a percentage of utilization of the available parallel resources on the architecture.

The vectorization reports indicated that for the DL code, 30% of the loops were successfully vectorized on the Xeon Phi. However, 26.667% of those loops were not completely vectorized. On the Fermi, LULESH achieved an occupancy of .378, .369, and .376 for the 50^3, 70^3, and 90^3 problem sizes, respectively. These values indicate that there were some functions in LULESH that made poor utilization of the GPU's available parallelism. On the Kepler architecture, LULESH achieved an occupancy of .428, .433, and .435 for the 50^3, 70^3, and 90^3 problem sizes, respectively. Considering the occupancy to be a percentage and comparing it to the upper bound of vectorization for the DL code executed on the Xeon Phi, the Kepler does best in terms of vectorization usage, followed by the Fermi, and then the Xeon Phi. Nevertheless, considering the occupancy achieved on the Fermi and the percentage of loops that were vectorized on the Xeon Phi, the difference between the vectorization capacity of the Fermi and the Xeon Phi is less than 15%. This maps to the execution time performance of the accelerators, i.e., the Kepler's execution time is superior to that of the Fermi for all problem sizes, and, although the execution time of the Xeon Phi is superior to that of the Fermi's for all problem sizes, the differences between the execution times do not exceed 15% percent. These results indicate that the execution time

performance of LULESH is linked to the accelerator's ability to exploit the inherent parallelism in the LULESH code.

4.3.3 Instructions Per Cycle

The instructions per cycle (IPC) of a program indicates the efficiency with which the device on which the program executes is able to exploit the parallelism that is inherent in the code. Although this metric is useful for determining the parallelism achieved by a specific device, the metric alone cannot be used to compare different architectures. This is due to the nature of how the experiments were conducted. In terms of those involving the Xeon Phi, the limited number of hardware counters available on the Xeon Phi required that IPC measurements be made for a single thread (IPC_{Thread}). In addition, the IPC of the GPU architectures was collected with the use of a profiler, which provided the IPC per streaming multiprocessor of the devices (known as an SM in the Fermi architecture and SMX in the Kepler architecture). Considering that it is debatable how comparable a GPU SM/SMX is to a thread of the Xeon Phi, the collected IPC cannot be used to determine which device was able to better exploit the inherent parallelism of LULESH.

Since a core of the Xeon Phi and the SM/SMX of the Fermi/Kepler architecture are the main computation units of the accelerators, the IPC per SM of the Fermi (IPC_{SM}) and the IPC per SMX of the Kepler (IPC_{SMX}) are compared to the IPC per core of the Xeon Phi (IPC_{Core}). IPC_{Core} is calculated by multiplying the IPC_{Thread} by the size of its pipeline. Each core of the Xeon Phi consists of a dual-issue pipeline, thus:

$$IPC_{Core} = IPC_{Thread} \times 2.$$

Although one could argue that a better approach would be to multiply IPC_{Thread} by the number of threads used per core, it requires making the following assumption: each thread used Phi/DL achieves the same IPC. However, this is not realistic. Furthermore, directly comparing a Xeon Phi thread to an SM/SMX of a GPU is unfair since a thread cannot distribute its work across simpler parallel components (as a Fermi SM, Kepler SMX, and Xeon Phi core does). Consequently, given

the size of the Xeon Phi's pipeline, it is more reasonable to assume that at least half of the threads used on a core of the Xeon Phi were able to reach the measured IPC_{Thread}.

Instructions per Cycle Comparison

Given the caveats mentioned above, as shown in Table 4.22, the Kepler achieved the highest IPC across all problem sizes. And, as mentioned before, the best performance achieved by LULESH was through the use of the Kepler architecture. Furthermore, the Kepler IPC increases with the problem size, as did the speedup achieved by KGPU/K (recall that in Section 4.2, the speedup of KGPU/K also increased with problem size).

With respect to the other accelerators, although the performance of FGPU/F was similar in magnitude to that of Phi/DL across the three problem sizes, the IPC of Phi/DL was consistently higher by about 20%. This behavior is reflected in the execution-time performance for the two smaller problem sizes. Specifically, the performance of Phi/DL for the 50^3,and 70^3 problem sizes was 1.12x and 1.14x higher than that of FGPU/F, while its IPC was 1.18x and 1.16x higher. Unfortunately, the inaccuracy of the IPC_{Core} calculation is somewhat exposed, with it being more prominent at the 90^3 problem size, where the difference between the performance of FGPU/F and Phi/DL is less than 1%, but the IPC of Phi/DL is 1.17x higher than that of FGPU/F.

Nevertheless, this method did expose that the IPC achieved by LULESH was much higher for the Kepler, than for the Fermi and Xeon Phi, and it translated to the execution-time performance of LULESH. Additionally, Phi/DL's IPC was higher than that FGPU/F, allowing it to exhibit better performance than the GPU, but the difference between the execution-time performance and IPC of these two accelerator/code pairs remained at less than 20% across the three problem sizes.

Table 4.22: IPC for each Architecture/Code Pair.

Architecture/ Code Pair	**Problem Size**		
	50^3	**70^3**	**90^3**
KGPU/K	1.161	1.175	1.178
FGPU/F	.306	.309	.307
Phi/DL	.360	.361	.361

Instructions per Cycle Comparison: Phase by Phase Analysis

Tables 4.23-4.25 present the IPC of the different accelerators for each execution phase of LULESH. Previously in Section 4.2.1, we presented the percentage of the execution time of LULESH attributable to each phase. Tables 4.26-4.28 again present these data again. As was the case with overall IPC, KGPU/K delivers the best IPC for all phases of LULESH, with the exception of the Calc & Apply Accel phase, where Phi/DL dominates. With the exception of Calc & Apply Accel and Lagrange, for the other two phases, KGPU/K's IPC increases as the problem size increases; this indicates that the different tuning techniques applied to LULESH have the potential of scaling well on this architecture/code pair.

Unfortunately, it cannot be generalized that LULESH will achieve good performance and scalability on all GPU architectures since FGPU/F's IPC became higher with the problem size for Time Constraints; for the other phases, it remained fairly constant. However, the advantage of KGPU/K in terms of IPC is reflected in the execution times of each phase. Calc Volume Force, Calc & Apply Accel, Lagrange, and Time Constraints executed up to 8.27x, 2.72x, 7.50x, and 3.53x faster on the Kepler than on the Fermi, and up to 7.07x, 3.39x, 7.89x, and 6.79x faster than on the Xeon Phi.

Overall, the FGPU/F and Phi/DL experienced comparable performance and similar IPCs. By breaking down the execution-time performance and IPC of LULESH by phase, it is clear why Phi/DL delivered the best execution time. It achieved higher IPC than did FGPU/F, with the exception of Time Constraints and Lagrange for the 50^3 problem size. In fact, FGPU/F's IPC and execution-time performance were 2x higher than that of Phi/DL for Time Constraints. However, the Fermi's overall performance does not exceed that of the Xeon Phi. This is likely due to the fact that Phi/DL's IPC is 1.26x higher (over all problem sizes) than that of FGPU/F for Calc Volume Force, which is the longest executing phase of LULESH (it consumes from 46.85% to 55.79% of the execution time of LULESH on the accelerators). Nevertheless, the difference between the IPC of FGPU/F and Phi/DL diminishes as the problem size increases. As a result, the difference

between the overall performance of the two architecture/code pairs is less than 1% for the 90^3 problem size.

In conclusion, the accelerators follow a similar pattern in terms of IPC. The best IPCs are achieved in the Time Constraints phase, which consumes the least amount of LULESH's execution time on the accelerators. Additionally, both FGPU/F and Phi/DL achieve their lowest IPCs in the Calc Volume Force phase, which consumes most of the execution time of LULESH. While KGPU/K achieves its lowest IPCs in Calc & Apply Accel, it must be noted that this phase consumes 8% more of the execution time of LULESH on the Kepler than on the Xeon Phi or Fermi. Nevertheless, the KGPU/K experiences its second lowest IPCs in Calc Volume Force. Thus, it appears that the execution time of LULESH is limited by the IPC achieved by each phase of LULESH. Each of the devices observed in this study has a maximum number of instructions that it can execute during each cycle. Regrettably, this peak value can only be achieved if there are no dependencies between instructions that can be executed in parallel, and if the application is not memory bound. Also, during its execution, the code must not experience bottlenecks in terms of its usage of the available resources on the device (e.g., not experience stalls in the pipeline due to memory latency). The IPC of a program, therefore, gives an indication of the number of instructions in the code that were executed simultaneously, which has a direct effect on the execution-time performance of the program.

Table 4.23: Phi/DL IPC per phase.

	Problem Size		
Code Section	**50^3**	**70^3**	**90^3**
Calc Volume Force	0.348	0.348	0.342
Calc & Apply Accel	0.510	0.520	0.522
Lagrange	0.398	0.372	0.358
Time Constraints	0.560	0.614	0.588

Table 4.24: Fermi IPC per phase.

Code Section	Problem Size		
	50^3	**70^3**	**90^3**
Calc Volume Force	0.269	0.272	0.270
Calc & Apply Accel	0.333	0.331	0.330
Lagrange	0.341	0.339	0.339
Time Constraints	1.088	1.243	1.402

Table 4.25: Kepler IPC per phase.

Code Section	Problem Size		
	50^3	**70^3**	**90^3**
Calc Volume Force	0.916	0.924	0.933
Calc & Apply Accel	0.332	0.333	0.334
Lagrange	1.645	1.690	1.681
Time Constraints	1.898	1.933	1.936

Table 4.26: Phi/DL - Distribution of Execution Time per Phase.

Code Section	Problem Size		
	50^3	**70^3**	**90^3**
Calc Volume Force	7.852	23.301	53.781
Calc & Apply Accel	0.845	2.242	6.380
Lagrange	6.127	19.426	52.330
Time Constraints	0.415	0.979	2.296

Table 4.27: FGPU/F - Distribution of Execution Time per Phase.

Code Section	Problem Size		
	50^3	**70^3**	**90^3**
Calc Volume Force	9.179	28.968	67.952
Calc & Apply Accel	0.722	2.188	5.052
Lagrange	6.836	20.895	47.681
Time Constraints	0.286	0.580	1.120
Time Constraints	0.081	0.170	0.338

Table 4.28: KGPU/K - Distribution of Execution Time per Phase.

Code Section	Problem Size		
	50^3	**70^3**	**90^3**
Calc Volume Force	1.110	3.508	8.251
Calc & Apply Accel	0.265	0.807	1.877
Lagrange	0.912	2.794	6.632
Time Constraints	0.081	0.170	0.338

Chapter 5: Conclusions and Future Work

Processor design continues to evolve, and as the needs of applications change, new architectures will be introduced. In HPC the use of accelerators is becoming more prominent to help "accelerate" the performance of applications by taking advantage of their highly parallel design. However, in order to fully understand the benefits of using a particular architecture, different performance metrics must be observed. Unfortunately, different devices will not necessarily expose the same metrics, and the differences in architectural design affect the comparability of the exposed metrics. Although several tools exist to collect performance metrics on supported architectures, there does not exist a standard way to compare the different metrics. Consequently, this study presents an attempt to develop a methodology that compares the performance of the NVIDIA Fermi, Kepler and MIC architectures by observing the execution time performance, power/energy consumption, memory behavior, vectorization capacity, and IPC of the LULESH application executed on each architecture.

5.1 Conclusions

By observing the execution-time performance of each of the three architecture/code pairs, it is evident that, in the case of LULESH: (1) the Kepler, i.e., KGPU/K provides the fastest execution time and (2) the Fermi, i.e., FGPU/F, and the Xeon Phi, i.e., Phi/DL, have comparable execution times. Additionally, the speedup provided by the Kepler and the Fermi, i.e., FGPU/F, increases with the problem size, while that provided by the Xeon Phi, i.e., Phi/DL does not. Despite the differences in architectural design, a breakdown of LULESH's structure highlights that the distribution of the execution times among the different phases of LULESH are similar for all three architecture/code pairs.

Although the power and energy consumption, and the execution times, of the Sandy Bridge and the Fermi GPU are comparable, the Xeon Phi has the largest power draw and energy consumption of all three architectures. And, even though the Kepler has the lowest power and energy consumption, the growth of its power draw is similar to that of the problem sizes used in

LULESH. The growth of the power consumption of the other architectures observed in this study did not experience any correlation with the problem size.

With respect to memory performance, the number of misses in the L1 and L2 caches of the Kepler and Fermi GPUs grow faster than the number of Xeon Phi L1-cache misses. And, the number of misses in the Xeon Phi's L2-cache and TLB grow faster than its L2-cache misses. As was the case with the execution time and power/energy data, the memory behavior of the Xeon Phi and the Fermi are comparable, i.e., both the Calc Volume Force and Lagrange phases of LULESH generate a similar percentage of misses (from 85%-98%)when executed on both architectures. Regrettably, the limited number of metrics available that can be used to quantify the memory behavior of LULESH on the three architecture/code pairs under study is not sufficient to explain the effect of memory performance on the execution time performance of the application. Nevertheless, the IPC and vectorization usage of LULESH executed on these architectures provide more insight.

Given the metrics that we used, the Kepler has the best vectorization usage and highest IPC across all problem sizes. Furthermore, as was the case with the speedup provided by the Kepler, the IPC of this architecture/code pair increases with the problem size. In contrast, although the execution times of the Fermi and Xeon Phi are comparable, the vectorization usage of the Fermi is roughly 10% higher than that of the Xeon Phi, but the Xeon Phi's IPC is consistently higher than the Fermi's by about 20%. Thus, it appears that the Kepler's high IPCs correlate with its execution times. Furthermore, the runtime behavior of the Xeon Phi and Fermi was comparable, and the IPC, vectorization, and memory behavior of LULESH on these architectures is also similar.

Finally, the Fermi and the Xeon Phi, which have comparable performance, have similar clock rates of 1.15 GHz and 1.1 GHz, respectively. The Sandy Bridge has a clock rate of 2.7 GHz (which is expected since the cores on the Sandy Bridge are much more sophisticated than the ones on the accelerators), and the Kepler has a clock rate of 705 MHz. However, the Kepler has much more CUDA cores than the Fermi, and the profiler does not give the IPC of an individual CUDA core. It makes measurements based on the streaming multiprocessors and/or warps.

5.2 Future Work

In order to expand and improve the contribution of this work, additional runtime execution configurations must be explored and other metrics must be considered. The Xeon Phi offers three different methods of executing an application and this study only explored one. To improve the comparison of the Xeon Phi with that of a GPU, its execution time performance in offload and symmetric mode must be measured. In these scenarios, the transfer time between the accelerators and the host processors must be collected, as it can, on occasion, result in a bottleneck due to the latency of the data-transfer operation.

Furthermore, Section 3.1 details that there was no LULESH code that was tuned for the Xeon Phi. Therefore, to improve the fairness of the comparison between the architectures, modifications must be made to LULESH to better use of the resources available in the Xeon Phi. However, we refrained from doing so in this study since one of the more attractive features of the Xeon Phi is the fact that it supports native execution and it executes codes meant for a multi-core Intel processor without modification.

Finally, in an attempt to develop a methodology that could be used to explain the performance of different types of accelerators, it is evident that at this time, because some performance metrics cannot be compared across the accelerators under study, such a methodology is premature. Nonetheless, several metrics were identified that can be used to explain why the execution times differed. The use of these metrics should be verified by using them to compare the performance of additional applications executed on these architectures. Doing this would also help establish what coding strategies benefit the most from one architecture versus another.

References

1. J. Dongarra, “Trends in High-Performance Computing: A Historical Overview and Examination of Future Developments,” *IEEE Circuits Devices Mag*, vol. 22, no. 1, pp. 22–27, Feb. 2006.
2. C. Rosales, “Porting to the Intel Xeon Phi: Opportunities and Challenges”. in *Extreme Scaling Workshop*, Boulder, CO, 2013, pp. 1–7.
3. S. Che, *et al*., “Rodinia: A Benchmark Suite for Heterogeneous Computing,” in *IEEE Int. Symp. Workload Characterization*, Austin, TX, 2009, pp. 44–54.
4. G. Misra, *et al.*, “Evaluation of Rodinia Codes on Intel Xeon Phi,” in *4th Int. Conf. Intelligent Systems Modelling & Simulation*, Bangkok, 2013, pp. 415–419.
5. J. Carabano, *et al.*, "An Exploration of Heterogeneous Systems," in *8th Int. Workshop Reconfigurable and Communication-Centric Systems-On-Chip*, Darmstadt, 2013, pp. 1–7.
6. M. G. B. Johnson, *et al.*, “Performance Tradeoff Spectrum of Integer and Floating Point Applications Kernels on Various GPUs”, in *Proc. 13th Int. Conf. Computer Design*, Auckland, 2013, pp. CDE4086.
7. B. Li, *et al.*, “The Power-Performance Tradeoffs of the Intel Xeon Phi on HPC Applications,” in *2014 IEEE Int. Parallel and Distributed Processing Symp. Workshops*, Phoenix, AZ, 2014, pp. 1448–1456.
8. K. Krommydas, *et al.*, "On the Programmability and Performance of Heterogeneous Platforms," in *2013 Int. Conf. Parallel and Distributed Systems*, Seoul, 2013, pp.224–231.
9. M. Bernaschi, *et al.*, “Multi-GPU Codes for Spin Systems Simulations”, *Computer Physics Communication*, vol. 183, no. 7, pp. 1416–1421, July. 2012.
10. I. Karlin, *et al.*, “Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application,” in *Proc. 2013 IEEE 27th Int. Symp. Parallel & Distributed Processing*, Boston, MA, 2013, pp. 919–932.
11. K. Sharma, *et al.*, “User-Specified and Automatic Data Layout Selection for Portable Performance,” Rice University, Houston, TX, Tech. Rep. TR13-03, April 2013.
12. I. Karlin, *et al.*, “Tuning the LULESH Mini-App for Current and Future Hardware,” *in Proc. 2012 Nuclear Explosive Code Development Conf.*, 2012.
13. I. Karlin, *et al.*, “Memory and Parallelism Tuning Exploration Using the LULESH Proxy Application”, in *2012 SC Companion: High Performance Computing, Networking, Storage, and Analysis*, Salt Lake City, UT, 2012, pp. 1429.
14. I. Karlin, *et al.*, “LULESH Programming Model and Performance Ports Overview”, Lawrence Livermore National Laboratory, Livermore, CA, Tech. Rep. LLNL-TR-608824, Dec. 2012.
15. J. M. Bull, “Measuring Synchronization and Scheduling Overheads in OpenMP”, in *Proc. of the First European Workshop on OpenMP*, Lund, 1999, pp. 99–105.
16. “Performance Application Programming Interface.” Internet: http://icl.cs.utk.edu/papi/, Dec. 3, 2014 [Oct. 30, 2014].

Appendix

A.1 Code Mappings

Although LULESH's algorithms were ported across different programming models, each version of LULESH does not necessarily use the same naming convention and structure for every function in the code. As mentioned in Section 3.1, LULESH was broken up into four phases. The following shows the call structure of the different code versions used in this study, and a mapping of the functions in the codes to the identified phases. The Calc Volume Force phase is denoted by the blue areas, Cal & Apply Accel is denoted by the red areas, Lagrange is denoted by green areas, and Time Constraints is denoted by orange areas.

LULESH 1.0 Optimized and DL-Optimized Mapping

```
A. TimeIncrement
B. LagrangeLeapFrog
    a. LagrangeNodal
        i.  CalcVolumeForceForElems
              1. IntegrateStressForElems
              2. CalcFBHourglassForceForElems
                    a. CollectDomainNodesToElemNodes
       ii.  CalcAccelerationForNodes
      iii.  ApplyAccelerationBoundaryConditionsForNodes
    b. LagrangeElements
        i.  CalcMonotonicQRegionForElems
       ii.  EvalEOSForElems
              1. CalcEnergyForElems
                    a. CalcPressureForElems
                    b. CalcSoundSpeedForElem
                    c. CalcPressureForElems
                    d. CalcSoundSpeedForElem
                    e. CalcPressureForElems
              2. CalcSoundSpeedForElem
      iii.  UpdateVolumesForElems
    c. CalcTimeConstraintsForElems
        i.  CalcCourantConstraintForElems
       ii.  CalcHydroConstraintForElems
```

LULESH 1.0 Fermi Mapping

```
A. CalVolumeForceForElems
   A1. IntegrateStressForElems
        A1.1 ComputeElemStress_Global
             K1. CollectElemPositions_device
                    CalcElemShapeFunctionDerivatives_device <<
                         CalcElemNodeNormals_device ***
                    SumElemStressesToNodeForces_device
        K2. AccumulateElemStressToNodes_ //
   A2. If (hgcoef > Real_t(0.)) CalcFBHourglassforElems
        A2.1 CalcElemFBHourglassForce_global_
             K3. CollectElemPositions_host_device ---
                    CollectElemVelocities_device_ ---
                    CalcElemVolumeDerivative_device_ ^^
        K4. AccumulateElemHGForceToNodes_global_
B. SimulateBoundaryExchange2
   K5. BoundaryToBuffer2
   K6. BufferToBoundary2
C. LagrangeNodal
   K7. CalcAccelerationForNode_global_ $$
   K8. ApplyAccelerationBCForNode_global_ **
   K9. CalculateVelocityPositionForNode_global_ ---
D. LagrangeElements
   K10. UpdateLagrangeElementsPart_1_ForElem
   D1. SimulateBoundaryExchange
        K11. BufferToBoundary
        K12.BoundaryToBuffer
   K13. UpdateLagrangeElementsPart_2_ForElem
   K14. UpdateVolumesForElem
E. CalcTimeConstraintsForElems
   E1. CalCourantConstraintForElems
        K15. CalcCourantLimitForElem
        K16. CalcArrayMin_kernel
   E.2 CalcHydroConstraintForElems
        K17. CalcHydroLimitForElem
        K16. CalcArrayMin_kernel
```

LULESH 1.0 Kepler Mapping

```
A. LagrangeLeapFrog
   A1. LagrangeNodal
        A1.1 CalcForceForNodes
             A1.1.1 CalcVolumeForceForElems
                  K1. CalcElemVolumeDerivative ^^
                        CalcHourglassModes
                        CalcElemShapeFunctionDerivatives <<
                        CalcElemNodeNormals ***

CalcElemFBHourglassForce
                  K2. AddNodeForcesElems //
        K3. CalcAccelerationForNodes $$
        K4. ApplyAccelerationBoundaryConditionsForNodes **
        K5. CalcPositionAndVelocityForNodes ---
   A2. LagrangeElements
        A2.1 CalcKinematicsAndMonotonicQGradient
             K6. CalcElemShapeFunctionDerivatives_device_
                 CalcElemVelocityGradient_device_
                 CalcMonoGradient_device_
        K7. CalcMonotonicQRegionForElems
        A2.2 ApplyMaterialPropertiesAndUpdateVolume
             K8.  ApplyMaterialPropertiesForElems_device
                    CalcEnergyForElems_device
                    CalcPressureForElems_device
                    Calc SoundSpeedForElems_device

UpdateVolumesForElems_device
   A3. CalcTimeConstraints
        K9. CalcTimeConstraintsforElems
        K10. CalcMinDtOneBlock
```

A.2 PAPI Profiling

On both the Xeon Phi and Sandy Bridge, PAPI was employed to collect event counts of metrics of interest. The following code samples show the instrumentation included in LULESH to collect the counters described in Section 3.3.5.

Profiling with Preset Events

```
int EventSet = PAPI_NULL; //Event set that will be measured
long_long values[NUM_EVENTS]; //array containing measurements
int events[] = {PRESET1, PRESET2, …}; //Events

if(PAPI_create_eventset(&EventSet) != PAPI_OK ) {
     fprintf(stderr, "PAPI Event set creation error!\n");
```

```
     exit(1);
}

for( i = 0; i < NUM_EVENTS; i++ ) {
     if( PAPI_add_event( EventSet, events[i] ) != PAPI_OK ) {
          fprintf(stderr, "PAPI add event  error!\n");
          exit(1);
     }
}

/* Start counting events */
if(PAPI_start(EventSet) != PAPI_OK) {
     fprintf(stderr, "PAPI start counters error!\n");
     exit(1);
}
                                   .
                                   .
                                   .
                        <CODE REGION OF INTEREST>
                                   .
                                   .
                                   .
/* Stop counting events and collect results */
if(PAPI_stop(EventSet, values) != PAPI_OK) {
     fprintf(stderr, "PAPI stop counters error!\n");
     exit(1);
}
```

Profiling with Native Events

```
int EventSet = PAPI_NULL; //Event set that will be measured
long_long values[NUM_EVENTS]; //array containing measurements
int native = 0x0; //Will be used to decode native event names
char *native_name[] = {"NativeName1", "NativeName2", ..};

if(PAPI_create_eventset(&EventSet) != PAPI_OK ) {
     fprintf(stderr, "PAPI Event set creation error!\n");
     exit(1);
}
/* Add each event that will be measured in event set */
for( i = 0; i < NUM_EVENTS; i++ ) {
     /* Translate native string name, to integer mask */
     if(   PAPI_event_name_to_code(native_name[i],   &native)   !=
     PAPI_OK ) {
          fprintf(stderr,    "Failed    decoding    name    %s\n",
          native_name[i]);
          exit(1);
     }
```

```
     /* Add event to event set */
     if( PAPI_add_event( EventSet, native ) != PAPI_OK ) {
          fprintf(stderr,  "PAPI  add  %s  event  error!\n",
          native_name[i]);
          exit(1);
     }
}

/* Start counting events */
if(PAPI_start(EventSet) != PAPI_OK) {
     fprintf(stderr, "PAPI start counters error!\n");
     exit(1);
}
                                   .
                                   .
                                   .
                        <CODE REGION OF INTEREST>
                                   .
                                   .
                                   .
/* Stop counting events, and copy results to values array */
if(PAPI_stop(EventSet, values) != PAPI_OK) {
     fprintf(stderr, "PAPI stop counters error!\n");
     exit(1);
}
```

A.3 Xeon Phi Configuration

Although each code version of LULESH implements the same algorithm, the codes for the Fermi, Kepler, and Sandy Bridge architectures were developed specifically for these architectures. In other words, LULESH was not tuned to run optimally on any architecture, but LULESH has been ported specifically to the Fermi, Kepler, and Sandy Bridge architectures. However, there is currently no port for the Xeon Phi. Thus, with no alternative, the Opt and the DL codes are used to analyze the Xeon Phi's performance in this study. Since LULESH was ported to the Fermi, Kepler, and Sandy Bridge, the code versions of each had either an accompanying Makefile or documentation that indicated the proper runtime configuration to achieve the best performance on each of the architectures. Since this was not the case for the Xeon Phi, due to its large number of cores (in comparison to the Sandy Bridge) and the number of hardware threads that each core can

run, experiments had to be conducted to determine the appropriate runtime configuration of LULESH on this architecture.

Compiler Options

Although the Xeon Phi offers different modes of execution, we chose to only run natively on the device. This is due to the fact that this mode of execution avoids the introduction of changes to the code and the need to transport data between the Xeon Phi and the host processor during execution. To indicate that an application is meant to run on a Xeon Phi, one must add the `-mmic` flag at compilation. The advised level of optimization to use when compiling for the Xeon Phi is O3, but the default level of optimization, which is O2 for the Intel compiler, was also used to verify that a higher level of optimization would provide better performance.

Both the Opt and DL codes were compiled using O3 and the default level of optimization (O2). Each binary was executed with the three problem sizes, using all of the available hardware threads on the Xeon Phi (244 threads). As shown in Figure A.1, when the default optimization (02) was used, the Opt code was 5.091%, 5.495%, and 3.685% faster for the 50^3, 70^3, and 90^3 problem sizes, respectively.

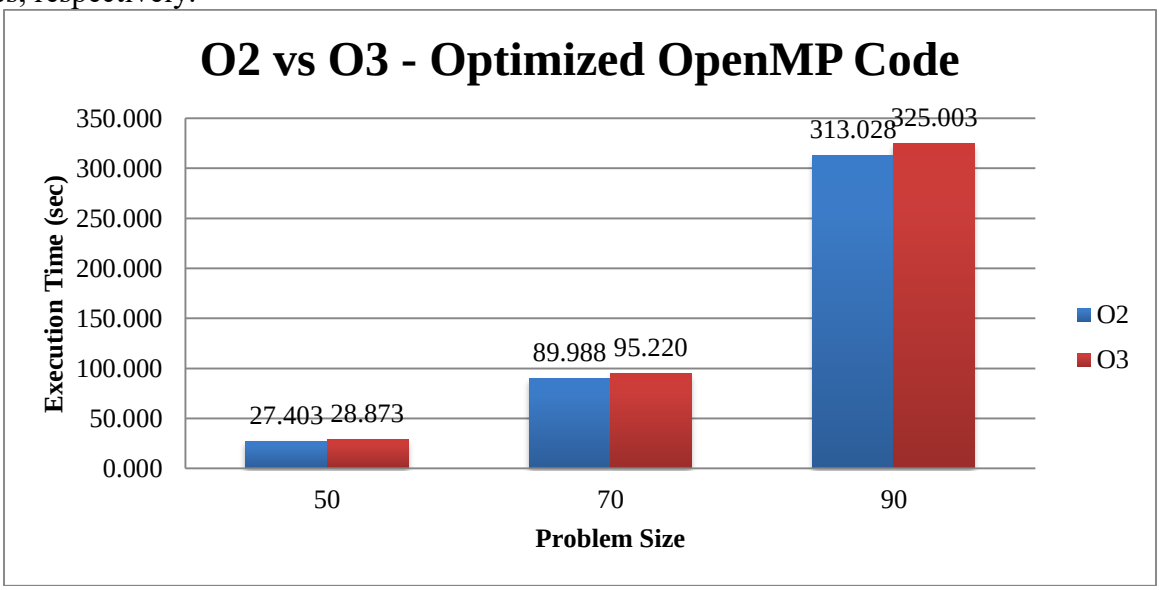


Figure A.1: O2 vs. O3 optimization for the optimized OpenMP code.

As shown in Figure A.2, the differences between the use of the default optimization (O2) and O3 are much more prominent for the DL code. Like the Opt code, the DL code experienced better performance using the default optimization level. However, the performance improvement for the DL code was 6.665%, 9.824%, and 5.878% faster for the 50^3, 70^3, and 90^3 problem sizes, respectively. Thus, in comparing the Xeon Phi with KGPU/K and FGPU/F, we use the DL code compiled with optimization level 2.

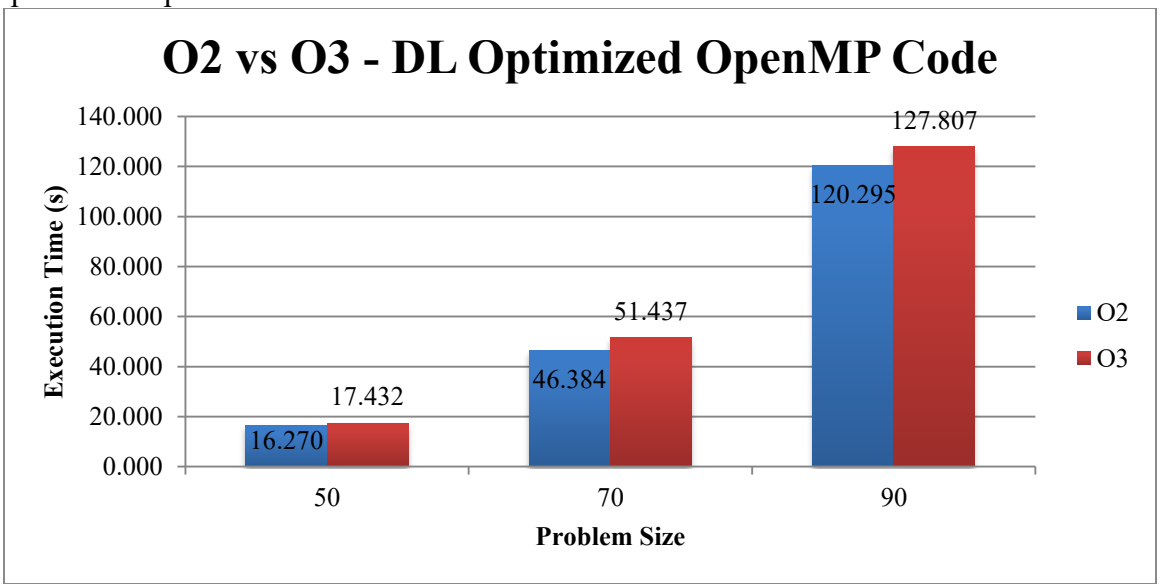


Figure A.2: O2 vs. O3 optimization for the DL optimized OpenMP code.

Affinity Settings

The experiments to determine the appropriate level of optimization to use were performed using the default affinity settings, which employs scatter affinity, and the maximum number of hardware threads available on the Xeon Phi. However, setting the affinity may provide additional performance benefits due to the fact that it is used to control how the threads map to the cores of the architecture. The four available options are scatter, balanced, compact, and none. As shown in Illustration A.1, setting the affinity to scatter results in threads being assigned to cores in a round-robin fashion, while setting the affinity to compact results in each core being assigned the maximum number of hardware threads it can run before attempting to fill another one. Balanced

affinity results in a similar mapping to that of scatter affinity, but threads with adjacent numbers end up placed on the same core. When no affinity is set, scatter affinity is used by default.

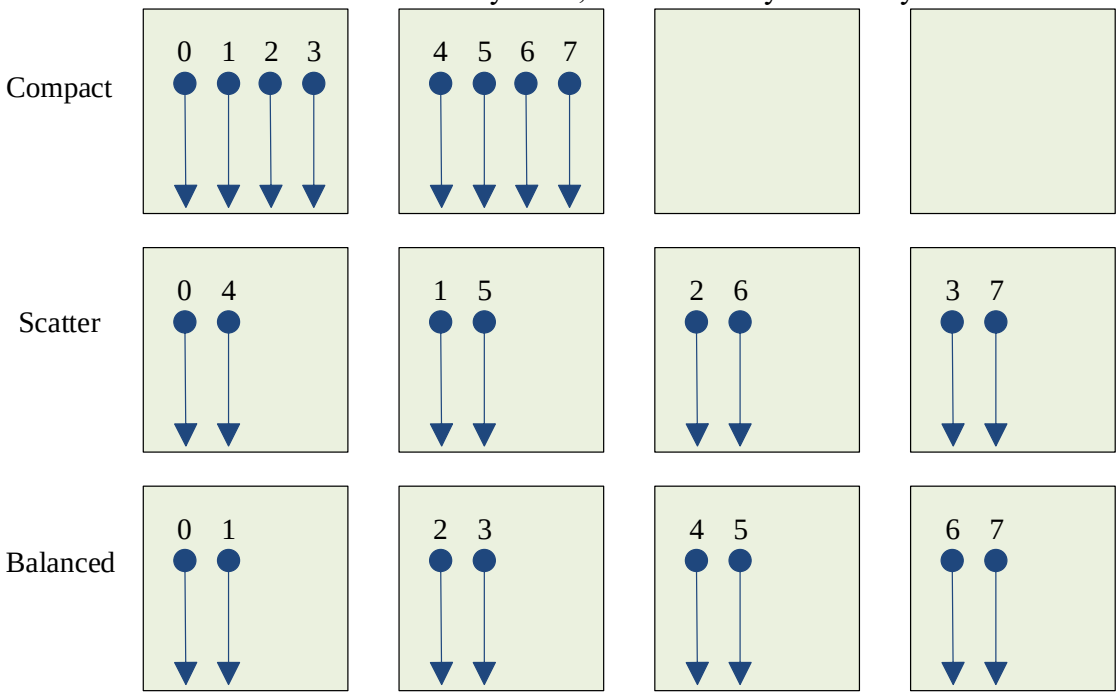


Illustration A.1: Affinity Settings on the Intel Xeon Phi.

Although the Xeon Phi allows the use of up to four hardware threads per core, it is not uncommon for an application's problem size to be insufficient to effectively make use of all the hardware threads. Considering this issue in scalability, both OpenMP versions of LULESH were run with each affinity setting, using 120, 122, 240, and 244 threads. The runs were conducted for each problem size. As shown in Tables A.1, A.2, and A.3, the experiments indicated that the optimized (Opt) code mapped very well to the parallelism available on the Xeon Phi. For all of the affinity settings available, the best performance (in green) was achieved when all of the hardware threads available on the architecture were employed. The best execution time for Phi/Opt was experienced with compact affinity, although the execution times with balanced affinity are very much the same at 240 and 244 threads. However, this affinity setting also had the worst performance (in red) when only 120 threads were used. This behavior is not surprising, since at 120 threads only 30 cores (of the available 61) are being actively utilized when the compact affinity

setting is employed. The remaining affinity settings, scatter and balanced, do not encounter this issue at the 120 thread count since they run at least 2 threads on a majority of the cores (60 out of 61).

Table A.1: Phi/Opt Runtime Under Different Affinity Settings – 50^3.

Affinity Setting	120 threads	122 threads	240 threads	244 threads
None	33.042	33.223	29.135	27.403
Scatter	33.007	34.591	28.453	27.510
Balanced	32.822	34.444	27.744	26.931
Compact	**53.988**	50.866	27.741	**26.917**

Table A.2: Phi/Opt Runtime Under Different Affinity Settings – 70^3.

Affinity Setting	120 threads	122 threads	240 threads	244 threads
None	104.808	102.718	96.314	89.988
Scatter	105.828	107.993	90.491	89.934
Balanced	106.407	105.791	87.763	83.052
Compact	**174.174**	159.542	87.628	**82.853**

Table A.3: Phi/Opt Runtime Under Different Affinity Settings – 90^3.

Affinity Setting	120 threads	122 threads	240 threads	244 threads
None	477.985	427.518	347.114	313.028
Scatter	483.117	442.448	324.407	311.782
Balanced	452.857	416.997	321.571	300.513
Compact	**625.937**	555.338	321.928	**300.298**

As shown in Tables A.4, A.5, and A.6, the DL code exhibits similar behavior to that of the Opt code, except it is much better. Again, with compact affinity, the worst execution times occurred when only 120 threads were employed and the best execution times occurred with all of the hardware threads available on the device in use. The exception to this pattern was at the 90^3 problem size, which achieved better performance with a balanced affinity setting; but, similar to the differences between the achieved performance with balanced and compact affinity for Phi/Opt, the differences for Phi/DL for all problem sizes were less than 1% with 240 and 244 threads. As

supported by the data presented in this section, 244 threads and compact affinity were used in the experiments that comprise this study.

Table A.4: DL Optimized Code Runtime Under Different Affinity Settings – 50^3.

Affinity Setting	**120 threads**	**122 threads**	**240 threads**	**244 threads**
None	19.606	19.695	16.090	16.270
Scatter	18.946	19.321	15.598	15.614
Balanced	19.146	19.093	15.042	15.003
Compact	**27.941**	27.819	14.969	**14.898**

Table A.5: DL Optimized Runtime Under Different Affinity Settings – 70^3.

Affinity Setting	**120 threads**	**122 threads**	**240 threads**	**244 threads**
None	58.090	57.813	46.451	46.384
Scatter	57.571	57.428	46.534	44.929
Balanced	56.344	57.572	44.867	44.550
Compact	**86.353**	85.317	45.273	**44.274**

Table A.6: DL Optimized Runtime Under Different Affinity Settings – 90^3.

Affinity Setting	**120 threads**	**122 threads**	**240 threads**	**244 threads**
None	154.748	152.076	116.963	120.295
Scatter	143.312	144.609	111.818	112.860
Balanced	143.003	141.727	109.583	**107.607**
Compact	**211.144**	207.052	109.680	107.721

Vita

Esthela Gallardo is a graduate research assistant at the University of Texas at El Paso. She has been involved with the HiPerSys Research Group from the Computer Science Department since her undergraduate career. She earned a Bachelor of Science in Computer Science from the University of Texas at El Paso in 2012. During her undergraduate career she participated in various internship opportunities. In 2010 she was selected as a participant in the Army High Performance Computing Research Center (AHPCRC) Summer Institute for Undergraduates in Computational Science and Engineering. During this research experience, Esthela visualized iterations of an optimization algorithm that attempted to solve the Hamiltonian cycle problem. Her participation in the AHPCRC Summer Institute led to her recruitment for an internship at the Army Research Lab (ARL) in 2011. At ARL she utilized the NASA World Wind Java API to develop a user interface for GPU-accelerated code responsible for detecting regional ballistic threats. In the last year of her undergraduate career she interned at Lawrence Livermore National Laboratory (LLNL), where she identified and documented performance analysis tools for GPU codes and ported optimizations of the OpenMP version of the LULESH proxy application to a CUDA version and evaluated the changes in performance. Finally, as a graduate student, she interned at the Texas Advanced Computing Center during the summer of 2014 where she aided in the development of a tool that makes use of the MPI Tools Interface to identify common performance bottlenecks in large-scale applications and provide suggestions on how to improve performance. The work associated with her internship at LLNL motivated the comparative study presented in this thesis.