Raytheon

Analyzing Computer Architecture of Intel Processors for Time Critical Applications

WPI- Raytheon

A Major Qualifying Project

Submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements of the

Degree of Bachelor of Science

in Electrical and Computer Engineering

by

Amanda Gatz

Date: December 19th, 2016

Approved:

Alexander Wyglinski, Faculty Advisor, WPI

Date: December 19th, 2016

Abstract

For time critical applications, it is essential that a digital processor is able to sustain the demand of the application's timing requirements. To analyze if a family of high performance Intel processors are equipped for different time critical applications, this project created a tool to calculate the performance of these processors in giga-floating point operations per second (GFLOP/s). With the proposed tool, this project compared the performance benefits of a traditional multi-core architecture, Intel Xeon E5-4669 v4, with a new many-core architecture, Intel Xeon Phi 7210.

Acknowledgements

Without the continuous support I received this project would not have been as successful as it was able to be. A special thanks for my advisor Professor Wyglinski for his continuous advice and support me throughout all stages of this project. I am very grateful for Sandy Clifford, who guided me throughout this project and gave me insight on Intel computer architecture. I would like to thank Brian Martiniello, who helped review and critique my signal processing comparison tool. I would like to thank Shane English, who taught me how to work with the Intel processors and Intel compilers. Lastly, I would like to thank the entire Receiver, Exciter, and Signal Processing section for their continuous support with my project and professional growth.

Executive Summary

As our world continues to develop and rely on technologies, we become dependent on knowing how and how fast our devices will work. For instance, when a paramedic is using a defibrillator in a critical situation she expects the digital signal processor (DSP) within the defibrillator to have no delays after she pushes the button and that the DSP will not miss any compressions. In any real life application, especially life critical situations, it is essential that our technologies are reliable and efficient. It is important that the computer architecture of the processors integrated within these technologies are able to keep up with the demand of the application. To be able to determine if the DSP is appropriate for the application, there are a range of software tools that can be used to measure their performance. While benchmarking tools are useful, the only way to ensure accuracy when measuring the performance of DSPs is to test the software application on the specific computer [4]. There are three reasons for potential variation between in the DSP being used in day-to-day applications and the benchmark tests done by companies: (1) there are considerable effects when the DSP is employed in the entire system, (2) different operating systems and compilers will impact the results, and (3) the benchmark tests are written for one particular application that may not be similar to every day-to-day application [4].

As a results of these differences, this project is tasked with testing the performance of the Intel processors for time critical applications. Intel has created a product line of Xeon processors for mission-critical workloads [5]. Within the Xeon processor series, there are several family generations. In one of its releases in recent years, Intel provided the Intel Xeon family series that consists of eight generations of development that included the Xeon E5-4669 v4 [6]. Within the past year, Intel has created the Intel Xeon Phi x200 family, also known as the Knights Landing, which improved upon the computer architecture of the earlier Xeon Phi generation, also known as the Knights Corner. Since this generation was released within the past year, there are few real world test results of the Knights Landing's performance. The goal of this project is to create a performance measuring tool to compare Intel Xeon processors to the newer Intel Xeon Phi processors. At the end of the project the signal processing comparison tool should be used on the Xeon E5-4669 v4, an older Intel generation processor and the Xeon Phi 7210, a newer Intel generation processor. The following are three major criteria for the signal processing performance tool:

1. Calculate multiple scenarios of vector lengths and number of threads based off of user inputs.
2. Compare the performance of the processors in GFLOP/s.
3. Relate the results from the comparison of the Xeon Phi 7210 and the Xeon E5-4669 v4 to the Xeon Phi 7290.

The computer architecture of the Intel processors will impact the performance of the machine. The different aspects of the Xeon Phi 7210, Xeon E5-4669 v4, and Xeon Phi 7290 can be found in Table i. However, not just one aspect of the computer architecture can determine how it will impact the overall performance of the machine. This is because even though with more cores and threads per core the machine will more than likely achieve a faster GFLOP/s rate it may not achieve a faster GFLOP/s per core rate. Additionally, the clock speed is misleading to the amount of work the processor can accomplish because it is not only the rate of the ticks on the machine, but how much work can be done within each tick [32, p. 264]. For example, "If machine A has a clock rate twice as fast as machine B, but each instruction on machine A takes twice as many clock

cycles as machine B to complete, then there is no discernible speed difference" [32, p. 264]. This means that the clock speed does not entirely explain how fast a processor will compute instruction of the application. Finally, the memory speed, type, and amount of storage can impact the performance of the machine by determining how fast the instructions can be handled in the memory.

Table 1: Overview of all of the Intel processors discussed within this report. Specifically, this table shows the major differences between in the Xeon Phi 7210 and Xeon Phi 7290 are more cores/threads, faster clock speed, and faster memory bandwidth. From the results, it may be possible that the Xeon Phi 7290 not only has a faster GFLOP/s rate, but a faster GFLOP/s per core rate than the Xeon Phi 7210.

Categories	**Intel Xeon Processor E5-4669**	**Intel Xeon Phi Processor 7210**	**Intel Xeon Phi Processor 7290**
Number of Cores	22	64	72
Number of Threads	44	256	288
Processor Base Frequency	2.20 GHz	1.3 GHz	1.50 GHz
Cache	55 MB	32 MB L2	36 MB L2
Max Memory Bandwidth	68 GB/s	102 GB/s	115.2 GB/s
Max Memory Size	1.54 TB	384 GB	384 GB

To be able to observe the differences in performance, the signal processing comparison tool was implemented to have three steps of computation:

1. The user interface asked for four vector lengths, three numbers of threads, number of iterations, and a vector operation number that corresponded with a vector function.
2. The tool would compute the vector function for the 12 scenarios each for the given number of iterations to produce 12 data points that are stored in a text file.
3. The text file will be imported to Microsoft Excel to produce graphs with three processing curves per processor.

The proposed signal processing tool aimed to have at least five vector functions implemented to be able to test the performance of the machine in a variety of applications. However, the learning curve of the POSIX threads was significantly more challenging than accounted for in the proposed timeline. The unexpected complexity of POSIX threads impacted the final results of the Xeon Phi 7210 performance only allowing the time to modify the tool to achieve 50% the expected GFLOP/s per core. Due to this, the vector add function was the only vector function that was implemented in the signal processing comparison tool.

While several of the proposed tasks were not completed, the results were still sufficient to prove that the Xeon Phi 7210 achieved a better GFLOP/s rate than the Xeon E5-4669 v4. The graph in Figure i the Xeon Phi 7210 and the Xeon E5-4669 v4 were compared with their respective optimized advance vector instruction sets, which are shown at the bottom of the graph. With their

respective optimized advance vector instruction sets the Xeon Phi 7210 and the Xeon E5-4669 v4 obtained similar GFLOP/s per core.

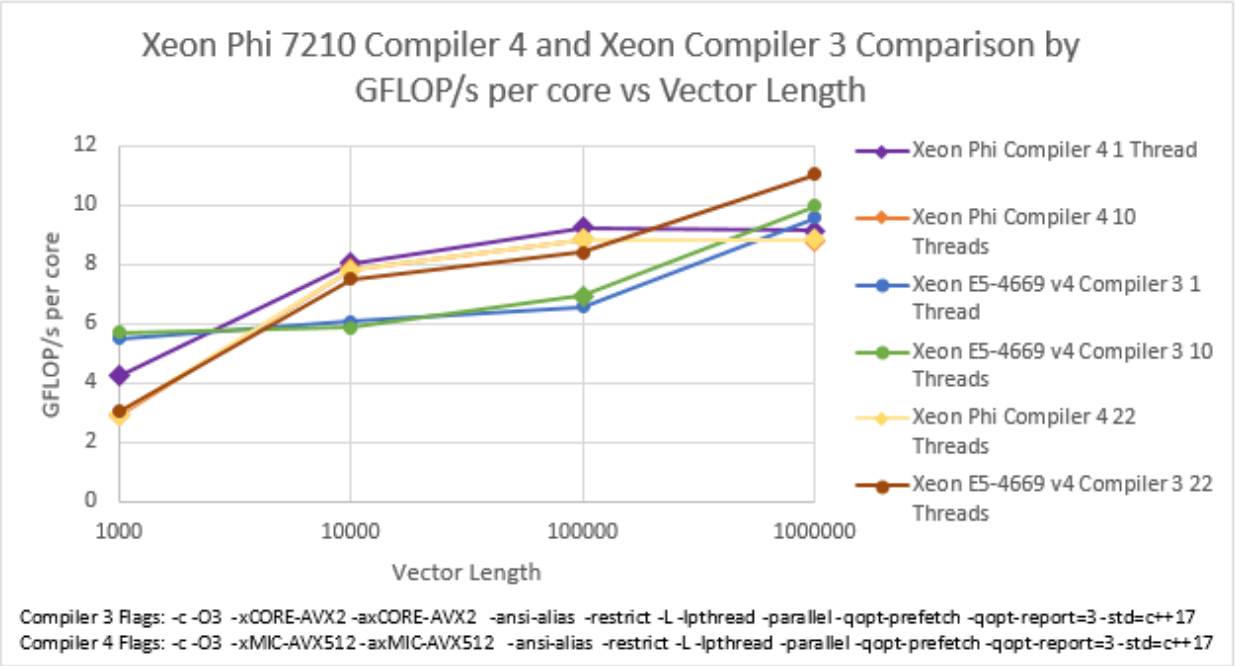


Figure 1: The graph compares Xeon Phi 7210 with the compiler option 4 shown below the graph and the Xeon E5-4669 v4 with the compiler option 3 shown below the graph. As can be seen the Xeon E5-4669 v4 and the Xeon Phi 7210 have comparable results.

From the results in the graph in Figure i, the best achieved GFLOP/s per core for the Xeon E5-4669 v4 was 11.04 GFLOP/s per core with 60 threads with 1,000,000 vector length and the best achieved GFLOP/s per core for the Xeon Phi 7210 was 8.99 GFLOP/s per core with 60 threads with 1,000,000 vector length. With this best achieving GFLOP/s per core for each machine the achieved GFLOP/s were:

The Xeon Phi 7210 achieved GFLOP/s:

8.99 (GFLOP/s per core) * 60 (cores) =539.4 GFLOP/s

The Xeon E5-4669 v4 achieved GFLOP/s:

11.04 (GFLOP/s per core) * 22 (cores) = 242.88 GFLOP/s

Given several incomplete tasks and the Xeon Phi 7210 only achieved 50% its expected GFLOP/s per core, there are many areas for continued work with the tool. Some areas for continued tests are: the structure of the program, memory settings, and data collection. After these recommended the tests are complete, there should be further investigations to ensure that there is an in-depth understanding of the Knights Landing generation architecture. An in-depth understanding would be beneficial for any future applications employing the Knights Landing generation processors.

Table of Contents

List of Figures

List of Tables

1. Introduction

As our world continues to develop and rely on technologies, we become dependent on knowing how and how fast our devices will work. For instance, when a paramedic is using a defibrillator in a critical situation she expects the digital signal processor (DSP) within the defibrillator to have no delays after she pushes the button and that the DSP will not miss any compressions. In any real life application, especially life critical situations, it is essential that our technologies are reliable and efficient. In addition to medical technologies, some other real world applications that DSPs handle are: radar/electronic warfare, single board computers, and smart cameras [1]. In all applications, DSPs must capture digitized real world information—such as sound, light, temperature, pressure, a button press, *etc*.—and process the information to be displayed, analyzed, or converted to another type of signal [2]. It is important that the computer architecture of the processors integrated within technologies are selected to be able to keep up with the demand of the application.

To be able to determine if the DSP is appropriate for the application, there are public tools that can be used to measure their performance. Generally these tools measure the performance in millions of instructions per second (MIPS), millions of operations per second (MOPS), and multiply-accumulations per second (MACS) [3]. One tool that benchmarks general-purpose processors and systems is Standard Performance Evaluation Corporation (SPEC) benchmarks [3, p. 2]. The SPEC benchmarking tools are mainly for high level applications that test software portability to calculate the MIPS, MACS, and MOPS of different applications [3]. However, measuring just MIPS, MACS, and MOPS does not consider memory usage and can be misleading since the amount of work varies for different types of instructions, operations, and multiply-accumulations [3, p. 2]. In contrast the benchmark tool, algorithm kernel benchmarking tool, created by Berkeley Design Technology *Inc*. calculates the execution time, memory usage, and energy consumption for fast Fourier transforms, vector add, filters, and other functions [3, p. 3]. To prove the accuracy of the algorithm kernel benchmark performance, the algorithm kernel benchmark performance results was compared to the MIPS performance results on Texas Instruments' TMS320C6203, VLIW-based processor. The comparison showed that the algorithm kernel benchmark was able to calculate more accurate results based on the expectations of the processor architecture [3].

Though the algorithm kernel benchmark has been proven to be more accurate than other benchmarking tools, no single numerical measurement can entirely describe the performance of DSPs [4]. The only way to be completely accurate when measuring the performance of DSPs is to test the software application on the specific computer [4]. There are three reasons for variance in the DSP being used in day-to-day applications compared to the benchmark tests done by companies which are: (1) there are considerable effects when the DSP is in the entire system, (2) different operating systems and compilers will impact the results, and (3) the benchmark tests are written for one particular application that may not be similar to every day-to-day application [4].

This project proposed the creation of a software tool that creates the performance of the Intel processors for time critical applications. Intel has created the product line of Xeon processors for mission-critical workloads [5]. Within the Xeon processor series, there are several family generations. In one of its releases in recent years, Intel provided the Intel Xeon family series that consists of eight generations of development that included the Xeon E5-4669 v4 [6]. Within the past year, Intel has created the Intel Xeon Phi x200 family, also known as the Knights Landing, which improved upon the computer architecture of the earlier Xeon Phi generation, also known as the Knights Corner. Since this generation was released within the past year, there are minimal real

world test results of the Knights Landing's performance. The goal of this project is to create a performance measuring tool to compare the Intel Xeon processors. At the end of the project the signal processing comparison tool should be used on the Xeon E5-4669 v4 and the Xeon Phi 7210. The following are three major criteria for the signal processing performance tool:

- The performance tool will be able to create a data set of points for different vector lengths and number of threads to create processing curves. The tool will be able to run one algorithm or vector function. For accuracy of each scenario, the tool will run for a user given number of iterations. For example if the users uses one-thousand iterations, the latency of each scenario will be averaged to account for the hit of the first iteration, which is expected to be much longer than the proceeding iterations. The first iteration will be much longer than the remaining iterations as the first iteration will have to go from out-of-cache to in-cache for the remaining loops.
- Instead of comparing the performance of the Intel processors in MIPS, MACS, and MOPS the performance measuring tool will compare the values in GFLOP/s. The data points returned from the processing tool will be in nanoseconds that will then be converted into GFLOP/s in an Excel workbook. The calculations will consider overhead and will calculate the GFLOP/s with the measured frequency from the processor. While Intel does give a theoretical GFLOP/s for the processors, the processors can rarely meet this theoretical GFLOP/s in real life application.
- Finally, the end goal of the project is to use the signal processing comparison tool on the Xeon Phi 7290. However, the Xeon Phi 7290 processor was not available for this project. Within the current scope of the project, the comparison tool will be tested on the Xeon Phi 7210 and the Xeon E5-4669 v4. However, the results and conclusions from the tests on these two processors will be applicable to when the signal processing comparison tool is used on Xeon Phi 7290. Additionally, the tool will be built to be adaptable for future processors such as the Xeon Phi 7290 and other Intel processors.

This report explains background information on signal processing, the Knights Landing generation architecture, and how the Xeon Phi 7210 differs from the Xeon E5-4669 v4 in the Intel Xeon Computer Architecture Overview chapter. The chapter on the Proposed Approach describes the proposed signal processing comparison tool and a plan used to complete the project successfully. Afterwards, the Methods and Implementation chapter discusses the trial and error of the project and the decision process on what was implemented. From the final implementation of the signal processing comparison tool the Results and Discussion chapter shows what graphs were made and discuss the outcomes of each vector function test. The Future Recommendation chapter outlines areas for further improvement and potential investigations. Finally, the conclusion chapter summarizes main points and how the results from the Xeon Phi 7210 and Xeon E5-4669 v4 comparison relate to the Xeon Phi 7290.

2. Intel Xeon Computer Architecture Overview

This chapter will create context for the design of the signal processing comparison tool. The following sections are divided into the general background information on processor architecture, Knights Landing 7290 architecture, the difference between the Xeon Phi 7210 and Xeon E5-4669 v4 and what are the expected results from the changes in the Xeon Phi 7210 compared to Xeon E5-4669 v4 processors.

2.1 Processor Architecture

Computer processing architecture has evolved over time based on the demand of the user and their expectations. With an increase in demand for computers to handle multiple tasks simultaneously, computer architecture needs to handle seemingly continuous and uninterrupted delivery of content [7, p. 2]. This section describes the difference in processor instruction sets, processor memories, and computer architectures, as well as how each of these parts of the architecture impact the execution of various applications. Additionally, there be will some discussion on the design decisions for the Knights Landing computer architecture.

2.1.1 Processor Instruction Set Architecture

An instruction set architecture (ISA) is related to the computer architecture in terms of addressing modes, instructions, native data types, registers, memory architecture, interrupt and exception handling, and external I/O [8]. The ISA is a component of the firmware system [9]. Where the firmware is the non-volatile memory structure that is stored even when the power is turned off; the firmware is usually written in assembly language [9]. The relationship between the ISA and the firmware required on a machine is that if any changes are made to the ISA, the firmware must be rewritten to reflect those changes [9]. This means that the machine language and how the machine functions are dependent on the ISA. In terms of software architecture, the ISA is the second level of architecture after the firmware, also known as the program binary, which is shown in Figure 2. The figure illustrates a software architecture overview from the lower level to the host interface. After the ISA in Figure 2 the computer architecture and memory are including in the performance core, which are explained in greater detail in Sections 2.1.2 and 2.1.3.

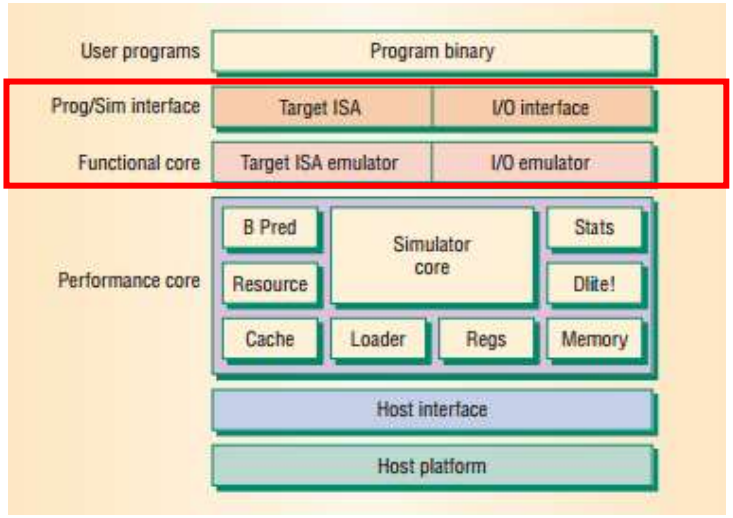


Figure 2: Overview of the software architecture [3, fig, 2]. The highlighted section shows the area of the software architecture that the ISA pertains to.

The ISA is split into the functional core and the program/simulator interface. In the functional core, the target ISA emulator interprets each instruction and directs each hardware model activity through call back interfaces [10, p. 61]. Then, in the program/simulation interface the target ISA writes a comprehensive mechanism to describe how to modify the registers and [10, p. 61]. The ISA is executed based on the instruction sets that are unique to each processor which can impact the way the processors performs.

2.1.2 Memory in Processors

Memory storage perpetually evolves in order to increase the capacity and performance of computers. As the memory storage adapts there are many design considerations that affect the memory access. The main design considerations depend on bandwidth and cost. The Knights Landing uses two types of memory: Multi-channel DRAM (MCDRAM) and the other is Double Data Rate (DDR). The following describes the different types of memory as well as the rationale behind the Knights Landing's usage of MCDRAM and DDR.

Random Access Memory (RAM) has two different types of memory: Static RAM (SRAM) and Dynamic RAM (DRAM). SRAM is fast, yet to maintain the SRAM stable state, it needs constant power, which is expensive [2]. The DRAM architecture requires much less power than SRAM, but consists of a capacitor [2]. When memory is read from DRAM the capacitor discharges, which is a disadvantage of DRAM because the DRAM must be constantly refreshed in order to compensate for the leakage of the capacitor [2]. Additionally, the discharge and charge of the capacitor is not instantaneous, which limits the speed of the DRAM [2]. On the contrary, the SRAM architecture does not need a refresh state to read or write memory [2]. However, even though DRAM has a limited speed, DRAM is much more cost effective than SRAM, which is why it is used in many applications.

Since DRAM is often used for many applications, DRAM is specified even further into separate memory types. One DRAM type is synchronous DRAM (SDRAM) which is a memory controller with a clock to transfer data. SDRAM has two successors: single data rate (SDR) and double data rate (DDR). SDR is expensive because the energy consumption rises with the frequency [2]. With increasing frequency the processors will have to increase the voltage to maintain stability, which adds to the cost [2]. In contrast, DDR is able to transfer twice the amount of data in the same cycle because it transports data on the rising and falling edge known as the "double-pumped" bus [2]. With increasing the frequency of the DDR, there is a concern that this will also increase the cost, which was a disadvantage for SDR. To overcome the increased frequency, DDR was designed with an I/O buffer Dual In Line Memory (DIMM) which should be capable of running at higher speeds [11]. This design solution allows DDR to be able to handle the same frequency and transmit double what DRAM is able to with half the power consumption.

In the Knights Landing architecture MCDRAM and DDR are used to handle the memory capacity and bandwidth that the design team required; both technologies are DRAM successors. To access DRAM, the memory address is encoded as a binary number using a smaller set of address lines that is then passed to the DRAM chip through a demultiplexer [11]. DRAM is organized by rows and columns, where the input address lines select the address lines of the whole row of cells then the input address lines find the column in those row that it is searching for [11]. This is conducted numerous times in parallel depending on the width of the data bus. To make this process scalable for other applications, there should be two external addresses for the row and column value of the address [11]. Timing constants are critical for the performance of the DRAM

in order for it to be able to handle the parallelism as well as the charging/discharging of the capacitor. For instance, there needs to be time between the outputs of one cycle being fed into the input of another cycle with respect to the formation of pipeline DRAM. When waiting for the next data set, it is ideal if the transfer time is less than the precharge time [11]. If the transfer time is longer than the precharge time, a delay is needed until the transfer time is complete [11].

The precharge and transfer times significantly affects the pipelining of the memory of the entire chip. Within the chip, on each individual tile there is local L2 cache. Cache memory is to store program instructions that are frequently re-referenced by the software during operation [12]. During the first cycle of the chip the data will be out-of-cache, which will have a longer latency than the succeeding cycles which will have the frequently reference operations stored in cache. There are three levels of cache memory: L1, L2, and L3. These cache memory types increase in speed in ascending order and increase in size in descending order [12]. L2 cache memory type is used in the Knights Landing tiles, which is faster than the L3 cache memory type and has more memory storage than L1 cache memory type.

2.1.3 Computer Architecture

As with other components of computer processing, the computer architecture itself continuously evolves over the years. One of the first computer architectures is the Von Neumann architecture, which consists of a program of sequence instructions that are stored sequentially in the computer's memory which are executed one after another in a linear, single-thread fashion [7, p. 1]. However in the 1960s computer architectures moved towards a concurrent program execution architecture. In this architecture, multiple users could simultaneously access a single mainframe computer to submit tasks [7, p. 1]. Nevertheless, even with the ability to handle multiple users, allocating CPU time was only handled by the CPU and the rest of the switching tasks was handled by the programmer [7, p. 1].

Before explaining the evolution of the computer architecture, it is important to understand the concept of threads. "A thread can be defined as a basic unit of CPU utilization. It contains a program counter that points to the current instruction in the stream. It contains CPU state information for the current thread. It also contains other resources such as a stack" [7, p. 8]. Threads are resources for the architecture state for the general purpose CPU registers and interrupt controller registers, caches, buses, execution units, and branch prediction logic [7, p. 8].

This process of only being able to handle one thread at a time is referred to as concurrency. Concurrency is an efficient use of system resources that maximizes the performance of computing systems by removing unnecessary dependencies on different components [7, p. 4]. This is an improved utilization of resources since all threads may be active, where one thread can be processed and the remaining threads can become ready until there are appropriate resources. Concurrency is easier to maintain because if there are multiple users are trying to access the application the architecture can handle each user on its own thread [7, p. 4]. Concurrency varies from parallelism since when threads are in parallel, they are running simultaneously on different hardware resources or processing elements [7, p. 5]. This means that in order to have parallel computing, the hardware must have multiple threads and processing elements. On the other hand, concurrency means that the threads are interleaved onto a single hardware resource meaning only one thread is processed at a time even if other threads are active [7, p. 5].

Whether a computer will be concurrent or in parallel depends on the computer architecture. Two dimensions in computer architecture that we need to consider are: (1) number of instruction

streams, and (2) number of data streams. Instruction streams may be able process at a single point in time while data streams are processed at a single point in time [7, p. 5]. These two dimensions of computer architecture can be categorized as either single instruction, single data (SISD), multiple instruction, single data (MISD), single instruction, multiple data (SIMD), or multiple instruction, multiple data (MIMD). Modern computing generally employees SIMD or MIMD to exploit data-level and task level parallelism [7, p. 6]. For general applications, the SIMD platform is useful for one instruction to handle multiple data streams in one clock cycle [7, p. 6]. On the other hand, the MIMD "machine is capable of executing multiple instruction streams, while working on a separate and independent data stream" [7, p. 6]. The Knights Landing architecture uses MIMD to have parallelism in its architecture.

"Instruction-level parallelism (ILP), also known as dynamic, or out-of-order execution, gives the CPU the ability to reorder instructions in an optimal way in order to eliminate pipeline stalls" [7, p. 7]. The goal is to have effective, multiple, independent instructions that execute more instructions in one clock cycle. Instead of having an in-order program execution, the processor reorders the instructions in order to allow independent instructions to execute in parallel [7, p. 7]. The two approaches to thread level parallelism are time-slicing and address thread-level parallelism. Time-slicing allows developers to hide the latencies present in the I/O by interleaving the execution of multiple threads [7, p. 8]. However, with time-slicing only one instruction stream can run on a processor at a single point in time [7, p. 8]. With the other approach, address thread-level parallelism increases the number of physical processors in the computer [7, p. 8]. Multiple threads or processes run simultaneously with more hardware on a multiprocessor system.

Simultaneous multi-threading (SMT), otherwise known as hyper-threading, makes a single processor appear to be multiple logical processor from a software perspective [7, p. 8]. Hyper-threading enables a single processor core to be used for two or more concurrent executions [11]. Multiple threads share the same execution resources, which enables the microarchitecture to determine how and when to interleave the execution of two or more threads [7, p. 8]. The executing engine allocates instructions to the one thread not being used. The executing engine interleaves the threads when the processor resources are available [7, p. 10]. If there are only two execution paths on one core and one thread is taking longer to read, the engine only processes the thread that is ready on one execution path until the other is available.

While parallelism seems to be the most sophisticated computer architecture, it is not the best solution for all processing applications. Depending on the application, different computer architectures will be needed. For example, a single core architecture is much less expensive since all the threads share the same L1 cache [13, p. 3]. In Figure 3, the single core architecture shows how the CPU state, interrupt logic, execution units, and cache are all on the same hardware. Conversely, multiprocessors are more expensive than their single core counterparts and multiprocessors can lose performance due to the difficulty of finding instructions that can execute together [13, p. 4]. However, single cores are unable to perform parallelism, while multiprocessors are able to. Performance on single-cores is limited since the system can interleave the instruction streams but cannot execute them simultaneously. While multi-core processors do not have to wait for resources, they can distribute different instructions on different processors [7, p. 12]. If the application requires parallelism, multiprocessors or multi-cores will be needed even though it is more expensive. In Figure 3, both the multi-core and multiprocessor architectures are shown. The main difference between the two architectures is that the CPUs are in parallel on the same hardware with the multi-core, while in multiprocessor architecture the entire processors are on separate

hardware in parallel. Additionally, in Figure 3, there is a multi-core with shared cache architecture, which allows more data sharing, remove false sharing, and reduces front-side bus traffic than having separate cache for the multiple cores [14]. False sharing is when the cores try to read or write memory from another core that is using that memory address [7, p. 13]. These benefits with shared cache is because the cores are controlled by the same instruction set, but are executed with different execution units.

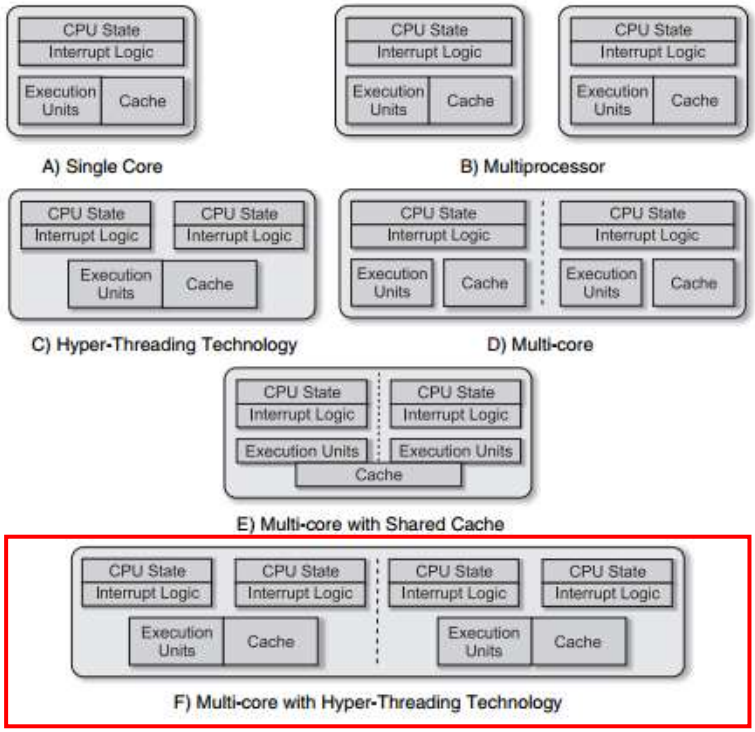


Figure 3: Overview of different computer architectures [6]. The highlighted is the computer architecture that the Knights Landing has.

Multiprocessors can produce several issues with respect to bottlenecking, which is why the Knights Landing was redesigned to be an independent processor instead of a coprocessor like the Knights Corner. To still have the parallelism possessed by multiprocessors, the Knights Landing architecture is a multi-core setup with hyper-threading. The main difference between hyper-threading with and without the multi-core is that with multiple cores the performance increases because without multiple cores the threads have to wait for a first come first served basis [15]. Several considerations for multi-threading with multi-core and single core are with respect to memory caching and thread priorities. With multicores each memory processor may have its own cache, which could become out of sync with other processors. If processors are out of sync with each other false sharing can occur. Additionally, with multiple cores the developer must optimize the code to recognize the higher priority threads must run before lower priority [7, p. 13]. The code could be unstable if the memory cache and thread priorities are not considered. An overview of the difference between hyper-threading architecture and hyper-threading with multi-core architecture can be seen in Figure 3. Overall, hyper-threading can potentially increase the throughput by up to 30%, which can increase the number of executable instructions by 1.3 times the number of executable instructions [7, p. 10].

While the Knights Landing was created as an independent processor, if it is using legacy applications that require a coprocessor it can switch modes to act as a coprocessor, note that the different modes will be explained further in the following sections.

2.2 Knights Landing

The Knights Landing was designed to improve upon the Knights Corner processor. Throughout the design process, the Knights Landing was improved upon architecture, memory, vector instructions, and other features. The original goal of the Knights Landing was to remove the overhead of the off-loading computation from the host processor to a co-processor card connected across the PCIe, Peripheral Component Interconnect Express, bus [16, p. 21]. To remove the transfer overhead, the Knight Landing was designed as a standalone processor, which can be connected to the network directly without needing additional data hops across the PCIe bus [16, p. 21]. By eliminating the need for the processor to send data through a software driver to the PCIe bus then to the coprocessor, the Knights Landing will no longer possess bottleneck issues from latency and bandwidth [16, p. 21]. An additional benefit to removing the need for a coprocessor the Knights Landing will enable the data to remain on its own main memory. As the goal for making the Knights Landing standalone evolved, it needed to be able to run single threaded, non-parallel, parts and still be power efficient when executing in parallel [16, p. 21]. Through the creation of the standalone processor the design team had decided to make the instruction set binary compatible to be able to use legacy code and adaptable for future use [16, p. 22]. To have a large compute capacity and standalone status, the memory architecture of the Knights Landing needed to be improved from the Knights Corner. To meet the user memory requirements of 2GB-4GB and bandwidth requirements, the Knights Landing used DDR to meet capacity requirements and MCDRAM memory to provide 450GB/s bandwidth [16, p. 22]. Finally, the Knights Landing uses one-die fabric to interconnect the tiles, which is a 2D mesh that allows for lower latency connections and can easily move in excess because of 450 GB/s of bandwidth delivered from the memory across the chip [16, p. 22].

In the following sections the improvements on ISA and memory are applicable to the both of the current released Knights Landing generations, the Xeon Phi 7210 and the Xeon Phi 7290. The Knights Landing architecture section is specific to the Xeon Phi 7290. The main difference between the Xeon Phi 7210 and Xeon Phi 7290 architecture is that the Xeon Phi 7210 has eight more cores than the Xeon Phi 7210.

2.2.1 Knights Landing Instruction Set Architecture

The Knights Landing architecture uses the Intel AVX-512, which has 9 subsets of instructions [17, p. 3]. The Intel AVX-512 is designed for the Knights Landing as well as future Intel processors. There are subsets that are shared between Intel Xeon Phi and Intel Xeon processors as well subsets that are unique to both, which is shown in the Venn diagram presented in Figure 4.

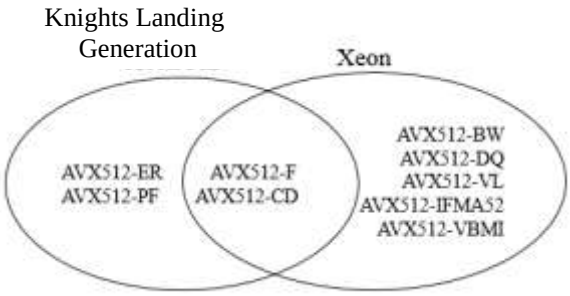


Figure 4: Replicated comparison of Instruction sets in Xeon and Xeon Phi [3, fig. 2]. The ISA of the Knights Landing helps improve the binary compatibility of the Knights Landing.

AVX512-F and AVX512-CD are two subsets used to maintain binary compatibility between the Knights Landings and future processors [17, p. 3]. "[AVX512-F] contains vectorized arithmetic operations, comparisons, type conversions, data movement, data permutation, bitwise logical operations on vectors and masks, and miscellaneous math functions like min/max" [17, p. 3]. The AVX512-F is similar to the AVX2 instruction set in the Knights Corner. However, the AVX512-F has wider registers and more double precision and integer support [17, p. 3]. The AVX512-CD is known as the "conflict detection" instruction set [17, p. 3]. Conflict detection is an instruction set that determines when two transactions cannot both safely commit [18]. The conflict detection searches for inconsistent data and read-write and write-write conflicts [18] . For instance, in the guide there are two locations for one variable that the compiler will want to automatically vectorized. However, this may cause a vector dependence due to conflicting memory access. If conflict detection instructions are added, this will make it possible for the compiler to generate vectorized code [17, p. 4]. Other subsets are the AXV512-ER and AVX512-PF, which will not maintain binary compatibility for future processors but can be used to vectorized, which previously was unvectorizable code [17, p. 3]. The AXV512-ER is the exponential and reciprocal instruction set, which includes base two exponential functions, reciprocals, and inverse square root in single and double precision with rounding masking options [17, p. 4]. This subset differs from the Knights Corner architecture IMCI instruction set, since it only supported single precision for these features [17, p. 4]. Finally, AVX512-PF is known as the Prefetch instruction set, which contains instructions for gathering and scattering instructions [17, p. 4]. These instructions do include software prefetch support, but Knights Landing processors possess a stronger emphasis on hardware prefetching [17, p. 4]. The instructions are similar to hints to the processors as to how to prefetch [17, p. 4]. Prefetch is fetching blocks of data before the memory address is called [19]. These instruction sets should optimize the Knights Landing to help it perform faster and utilize its resources efficiently.

2.2.2 Knights Landing Memory Architecture

The memory of the Knights Landing architecture needs to meet three requirements: (1) standalone processing, (2) memory capacity, and (3) the adaptability to different operating systems and applications. The memory types DDR and MCDRAM were chosen to meet the memory and bandwidth capacity [16, p. 22]. For adaptability, the memory types utilize cluster modes and memory modes to act almost as a different machine for different operating systems and applications [20, p. 25]. The clusters modes are groupings of the active tiles and the memory modes are what percentage of each memory type is with respect to either memory or cache. There are

three different memory modes and five different cluster modes, which can be set in different combinations that best fit the operating system and application. The default cluster mode is quadrant and memory mode is cache, which were based on the assumption that most applications that will be running on the Knights Landing architecture will be cache friendly [20, p. 25].

While there are five cluster modes, all modes are fully cache coherent. "This means that every core has the same view of what data is in every location" [20, p. 26]. The only variance between different cluster modes is whether the view of the memory is Uniform Memory Access (UMA) or Non-Uniform Memory Access (NUMA) [20, p. 26]. In UMA configurations all processors have equal access to the main memory controller at the same speed [21]. In contrast, in NUMA configurations all processors have access to all memory structures, but not all memory access is equal [21]. The access is unequal because each CPU has its own memory controller and the cost of accessing a specific location in the main memory is different for each CPU [21]. While the access is unequal, NUMA configurations allow for the programmer to manage his own memory to maximize the portability by reducing the lock contention and maximizing memory allocation [22]. The five cluster modes are divided into two groups of two modes that are NUMA and three modes that are UMA, which then differ slightly within those groups. The SNC-4 and SNC-2 cluster modes use NUMA, which divides the memory into four quadrants with lower latency for memory access between the same quadrant and higher latency for latency accessing different quadrants [20, p. 27]. SNC-4 is mainly helpful for applications that are running on four (or multiples of four ranks) of Knights Landing [20, p. 27]. SNC-2 is a variation of SNC-4, which runs in two ranks. However, it is more beneficial to use SNC-4 since an additional latency penalty is incurred when running in two ranks [20, p. 27]. In contrast, the quadrant mode, hemisphere mode, and all-to-all mode use UMA memory, which means the latency from any core to any memory location of the same memory type will essentially be the same throughout [20, p. 27]. Due to this, there is little variance of latency throughout the mesh, but this limits the user to be unable to determine the latency of a specific memory region [20, p. 27]. If an application requires a UMA, the cluster mode can then be determined by population of the DDR DIMMs connected to the Knights Landing and the number of active tiles. If the DDR DIMMs connected to the Knights Landing are not evenly populated, then the all-to-all mode will be used [20, p. 26]. If the DDR DIMMs connected to the Knights Landing are evenly populated, the quadrant or hemisphere cluster can be used. The hemisphere mode is a variation of the quadrant mode. Similar to SNC-2, hemisphere runs in halves, which causes latency and reduces the bandwidth [20, p. 27]. Otherwise, the quadrant mode is used when the tiles are a multiple of four.

The Knights Landing architecture has 300GB of storage, which is distributed as MCDRAM on eight devices with 2GB capacity on each processors and DDR, external memory, one for every three channels with max capacity at 284 GB [16, p. 20]. DDR and MCDRAM can both handle single data access at the same bandwidth. However, the MCDRAM can handle higher bandwidth allowing for more simultaneous data access than the DDR [20, p. 27]. Consequently, the DDR is solely used for memory storage and MCDRAM changes depending on mode to be cache or memory storage. The three memory modes are flat mode, cache mode, and hybrid mode. In flat mode, the DDR and MCDRAM is accessible to all applications and operating systems as separate NUMA nodes [20, p. 27]. This means both DDR and MCDRAM are used as memory storage for the application and there is no cache. If there is a coprocessor in the application, this is the only mode that is available for Knights Landing since many coprocessors do not have DDR support and only have MCDRAM support [20, p. 28]. Cache mode is when 100% of MCDRAM is used as

cache and the DDR is used as memory. Finally, hybrid mode is when 25% to 50% of MCDRAM is used as cache and the rest of MCDRAM is used as memory along with the DDR memory [20, p. 28].

2.2.3 Knights Landing Architecture

The Knights Landing has a multi-core with hyper-threading architecture. The Knights Landing supports up to four threads per tile [23, p. 63]. The design consists of 38 tiles although usually only 36 tiles are active and the remaining two are present to improve future manufacturing chip enhancements [16, p. 17]. On each tile, there are two 2 unit vector processing units (VPUs) and two cores share the same 1M L2 cache. With 36 active tiles, there are a total of 72 cores and 144 VPUs on the chip [16, p. 17]. In Figure 2, there is a high level overview of the Knights Landing architecture within the processing die. In the block that is zooming into a single tile, we notice that it has a similar structure to the hyper-threading architecture in Figure 3. The larger block in Figure 5 is an overview of the chip package that shows the 36 tiles that are in each package.

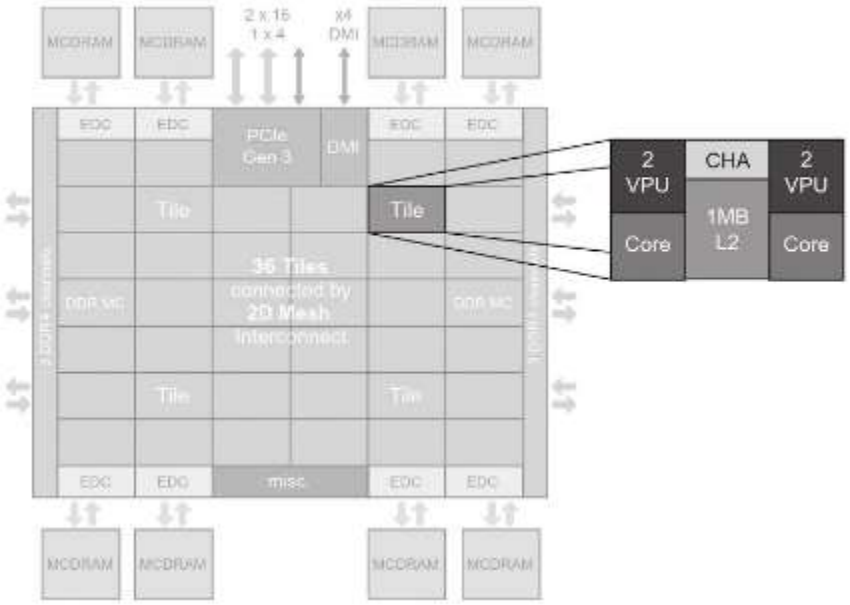


Figure 5: Overview of the Xeon Phi 7290 Chip Architecture [2, fig 4.1]. The image shows the 36 tiles on the Xeon Phi 7290. The zoomed in block of a single tile on the right side is what a tile on any current generation of the Knights Landing processors would entail.

The VPU is responsible for the floating-point computation that can handle legacy and new vector functions [23, p. 63]. Since there are two VPUs in every tile, each tile can handle 64 single precision floating-point and 32 dual precision floating-point [23, p. 63]. For each tile, L2 memory is used such that each tile can read it on their respective tile [23, p. 65]. In conjunction with the VPUs and core handling the vector functions, hyper-threading allows for the Knights Landing to be able to dynamically process all memory. The core supports four threads of execution [23, p. 70]. The Knights Landing can turn off threading, which differs from the Knights Corner since if one thread is accessed in the Knights Corner then all of the resources will then be available [23, p. 70]. The ideal amount is two threads per core active to allow for out-of-order processing, which can hide instruction and cache access latencies for many applications [23, p. 70]. During each cycle in Knights Landing, there are thread selections that decide which instructions move forward based

on factors such as: thread fairness, instruction availability, thread-specific stalls, and other constraints [23, p. 70]. Based on the factors, the partitioned structure readjusts resources for threads as needed, which share resources on a first come first served basis on each tile [23, p. 70]. Dynamic cache partitioning estimates the amount of cache allocations different processes needs then gives the estimated amount to each process [24, p. 8]. The goal of dynamic cache partitioning is to limit the miss rate of cache [24, p. 9].

Within each tile, the core is designed with two-wide, out-of-order core derived from Intel Atom processor code, Silvermont [16, p. 18]. The Knights Landing has similar microarchitecture to the Silvermont such as: four hyper-threads per core, deeper out-of-order buffers, higher L1 and L2 cache bandwidths, adding AVX-512 vector instructions, improved reliability features, large TLBs, and larger L1 cache [16, p. 18]. Each tile has its own local L2 cache, but to communicate data stored outside of the tile it must use the on-die interconnect [16, p. 19] This architecture can be seen in Figure 4, where the L2 cache is local on the tile. However, the on-die interconnection is not visible in the figure. The on-die interconnect is a cache-coherent, two-dimensional mesh interconnect [16, p. 19]. This allows the Knights Landing to be more scalable for future use than the Knights Corner, which is one-dimensional. Additionally, the different clusters allow the tiles to be separated into virtual regions, lower latency, and increase bandwidth [23, p. 71]. To keep the tiles L2 cache coherent of each other, the on-die interconnect follows the MESIF cache-coherent protocol, which is Modified, Exclusive, Shared, Invalid, and Forward state [16, p. 19] Each tile is connected to the mesh and holds a portion of the MESIF structure in its CHA, caching/home agent [16, p. 19]. Since the tile has the inputs and outputs on the same edge, the architecture avoids long connections across the chip [16, p. 19] Additionally, to increase efficiency the mesh uses "YX routing," which has the data travel vertically to the target row then horizontally to the target [16, p. 19]. The aim for a routing algorithm is to minimize the number of network hops and delays through the router [25]**.** The YX routing achieves this by linking with nodes in the Y direction until it reaches the correct Y address, then links with nodes in the X direction until it reaches the X address.

By having four VPUS, two cores, and shared L2 cache per tile, gives the Knights Landing the advantage of hyper-threading four threads per core and eight threads per tile to compute more work simultaneously than other Intel processor generations.

2.3 Expected Impact from Differences in Intel Processors

The changes made in Knights Landing are predicted to increase the performance of the processor. In a study completed by Intel comparing the Knights Landing projected performance with 1 socket and 200W CPU TDP verse the 2 socket Intel Xeon processor E5-2697v3 demonstrates that in almost all categories—except standard performance evaluation corporation integer rate and standard performance evaluation corporation floating point rate—the Knights Landing was able to have a better performance value than the Xeon E5-2697 v3 [26].

The signal processing comparison tool will compare the differences between the Xeon Phi 7210 processor and the Xeon E5-4669 v4 processer. In Table 2, some factors between the Xeon E5-4669 v4 and Knights Landing are compared. The reason the Knights Landing has almost six times the amount of threads and only triple the amount of cores compared to that of the Xeon E5-4669 v4, is due to the Knights Landing's more advanced hyper-threading. This hyper-threading allows the Knights Landing to have four threads per core as opposed to the Xeon E5-4669 v4, which only has two threads per core. The Knights Landing has a lower processor frequency base

compared to the Xeon E5-4669 v4. However, the Knights Landing memory bandwidth is greater than the Xeon E5-4669 v4. Finally, the Knights Landing has much less memory and cache than the Xeon E5-4669 v4.

Table 2: This table compares fundamentals between the Intel Xeon E5-4669 v4 [27] and Xeon Phi 7210 [28]. One noteworthy variance between the two processors is that the Knights Landing has hyper-threading which allows it to have almost six times the number of threads as the Xeon. The other standout difference is that the Xeon has about 4000 times more max memory size than the Knights Landing.

Categories	Intel Xeon Processor E5-4669 v4	Intel Xeon Phi Procesor 7210
Number of Cores	22	64
Number of Threads	44	256
Processor Base Frequency	2.20 GHz	1.3 GHz
Cache	55 MB L3	32 MB L2
Max Memory Size	1.54 TB	384 GB
Max Memory Bandwidth	68 GB/s	102 GB/s

Each of these differences within Xeon Phi 7210 and Xeon E5-4669 v4 contribute to the performance of the machines. The categories can be divide into three sub-groups: architecture, speed of the processor, and memory.

Since the architecture of Xeon Phi 7210 has almost triple the amount of cores with ability to hyper-thread double the amount of threads per core as Xeon E5-4669 v4, the Xeon Phi 7210 should be able to compute more GFLOP/s than the Xeon E5-4669 v4. Computing more GFLOP/s allows the processor to compute more math functions than a processor with less GFLOP/s. The reason why more threads allows for a higher rate of GFLOP/s is because it requires fewer resources to be managed since the threads share same address space on each core [29]. However, while the threads do share the same address space the threads will not always share memory [29]. Yet this memory sharing is overcome by the Xeon Phi 7210 being able to hyper-thread four threads per core. With the ability to hyper-thread four threads per core the Xeon Phi 7210 instead of two threads like the Xeon E5-4669 v4, the Xeon Phi 7210 can theoretically do double the amount of tasks per core; this concept is similar to the idea of moving from a two lane highway to a four lane highway, which means four cars can pass at the same rate instead of two [30]. In addition, with hyper-threading the processor can dynamically adapt the workload and automatically disables inactive core, which allows the processor to increase the frequency on the busy core allowing threads to be processed faster [30].

While the hyper-threading on the cores can impact the frequency for individual tasks, the overall clock speed, in frequency, determines how many instructions the processor can handle within a second [31]. However, the clock speed is misleading to the amount of work the processor can accomplish because it is not only the rate of the ticks on the machine, but how much work can be done within each tick [32, p. 264]. For example, "if machine A has a clock rate twice as fast as machine B, but each instruction on machine A takes twice as many clock cycles as machine B to complete, then there is no discernible speed difference" [32, p. 264]. This means that the clock speed does not entirely explain how fast a processor will compute instruction of the application. Due to this a more accurate benchmark of machine speed is instruction rate measured in MIPS [32, p. 264]. This measured how fast the machine can handle machine language instructions in a second [3]. However, on machine instructions can require multiple clock ticks, which is a process that parallel processors can compute multiple times within one single tick [32, p. 264]. Even from using MIPS, the entire time to execute a function cannot be determine. Due to this, users often determine the time a processor will take to execute a subset of instructions most important to their applications that is generally gauged by GFLOP/s [32, p. 264]. Overall, this means that even though the Xeon E5-4669 v4 has a faster clock speed until testing is complete it cannot be determined if the Xeon E5-4669 v4 or Xeon Phi 7210 will have a faster GFLOP/s rate.

The memory of the processor is another characteristic of the machine to understand the achieved GFLOP/s rate. The L2 cache is memory which enhances the cache on each core by storing data and instructions that the processor needs access to quickly [33]. While in Table 1 the Xeon E5-4669 v4 does have a larger amount of cache memory than the Xeon Phi 7210, the Xeon Phi 7210 has L2 cache which is faster than L3 cache in the in Xeon E5-4669 v4, which is explained in Section 2.1.2. This means the Xeon Phi 7210 will be able to handle instructions faster. In addition the Xeon Phi 7210 has almost double the bandwidth of the Xeon E5-4669 v4 that will contribute to the speed up of the Xeon Phi 7210. However, the RAM memory of the Xeon E5-4669 v4 has four times the storage than then Xeon Phi 7210 which helps the performance of the Xeon E5-4669 v4 by giving the machine more room to store data while it cannot process the instructions as quickly. While the changes on the Xeon Phi 7210 are meant to achieve a higher rate of GFLOP/s it cannot be determined based off of the specifications whether or not the Xeon Phi 7210 will perform better in day-to-day application compared to the Xeon E5-4669 v4.

2.4 Chapter Summary

This chapter reviewed computer architecture to give the reader context to the functionality of the Intel processors. It then describe the modifications in the Knights Landing generation. Finally, it reviewed the differences between the Xeon Phi 7210 and Xeon E5-4669 v4 expected impacts on the latency from those differences. This information helps give background for the reader to understand the thought process behind design decisions that are described in later chapters. Understanding the changes in the Knights Landing 7290 will be useful when review the results of the comparison between the Xeon Phi 7210 and Xeon E5-4669 v4. The conclusions drawn from those results can predict the future results of the signal processing comparison tool used on the Knights Landing 7290.

3. Proposed Approach

This chapter will discuss the proposed design of the signal processing comparison tool and the schedule to meet the deadlines to complete the project successfully. Since the project will be completed individually and within a seven week term diligence is needed to complete the project. The goal of the signal processing comparison tool is to be able to test the Xeon Phi 7210 performance level in contrast to the performance level of the Xeon E5-4669 v4. This will be done by testing the performance of various vector functions for different scenarios that vary the number of threads used, the length of the vector, and number of iterations. Once these tests are completed the conclusions drawn from the results will aid in making predictions about the performance of the Xeon Phi 7290.

The signal processing comparison tool will be created in the Linux Intel C compiler. Each vector length for each thread will be a data point. The data points from each program call will be written to a text file for the each processor. Ideally, as the data points are written to the text file the program will generate a graph of the processing curves. If the graphs are created from the C program the major complication will be getting the data points from both processors onto on graph.

3.1 Functionality of the Signal Processing Comparison Tool

The goal of the signal processing comparison tool is to be able see how the changes in the computer architecture impact the processors' performances. In Section 2.3 the differences in the characteristics between the Xeon Phi 7210 and Xeon E5-4669 v4 were discussed. The characteristics are split into categories: architecture, speed of the processor, and memory. Out of these characteristics, the signal processing comparison tool will only control the number of threads used and what cores the threads work on. However, the memory modes can be changed to see the impact of different memory types. This will not be done by the signal processing tool, but is a setting on the Xeon Phi 7210 that can be changed. Finally, the clock speed will not be manipulated because the machines should reach their maximum clock speed [31].

To be able to observe the differences in computer architecture, the program creates a graph of processing curves for a specific vector function on the Xeon Phi 7210 and Xeon E5-4669 v4 to compare the performance level. The signal processing curve will have three main features: (1) dynamic user inputs for adaptability for future use, (2) at least five vector functions, and (3) a graph with three processing curves from the Xeon Phi 7210 and Xeon E5-4669 v4. Once the tool is called in the Linux command window, the user will give three different numbers of threads and four different vector lengths; this creates processing curves for the vector function for a variety of threads and vector lengths that will be compared on one graph. The x-axis will be the vector length and the y-axis will be time. Each of the processing curves will be based on the number of threads and which processor it was computed on. For instance, if the user wanted to compare the following:

- Vector Lengths: 1000, 10000, 100000, and 1000000
- Threads: 1, 10, and 44
- Iterations: 1000
- Operation: hand-coded complex vector add operation

The graph would have six processing curves: three curves for the Knights Landing with 1, 10, and 44 threads and three curves for the Xeon E5-4669 v4 with 1, 10, and 44 threads. Each

processing curve would have points at 1024, 10,000, and 1,000,000 vector lengths. Each vector length point would be an average of 1000 iterations.

3.2 Objectives

The functionality of the signal processing comparison tool will be implemented with the ability to compute at least five vector functions, allow the user to give variety of input values, and produce a graph with all processing curves on it. These three objectives are the goals of each section of the logic block diagram in Figure 6. In Figure 6, a logic block diagram shows how the program should handle the user inputs only if the hand-coded complex vector add operation, Intel Math Kernel Library (MKL) in single thread mode, and MKL in multi-threaded mode are implemented. The logic block diagram shows how the program will determine which vector function will be run then based off how many iterations the vector function will repeat for the given vector length and number of threads. After each specific thread number and vector length pair are computed a point for the thread processing curve will be created. Once each scenario is complete, a graph will be created from all of the processing points.

The first section of the block diagram handles the user inputs. When the user calls the program in the Linux command window, the program should prompt the user for four vector lengths, three numbers of threads, the number of iterations, and which operation the user wants computed. By having the user inputs, the program will be adaptable for future technologies that have more threads. The adaptability of the program is important for future applications and scenarios because whenever a new processor is acquired, performance tests will need to be conducted on the machine since it is only way to obtain an accurate performance assessment on the specific computer [4]. While Intel and other companies may provide benchmarks, there are considerable effects when the processor is implemented in the entire system. Intel may have used a different operating systems and compilers, and the benchmark tests are written for one particular application that may not be similar to a target application [4]. Even further, by having the operation number as a user input it would be easy for another company to modify the program and add or remove additional vector functions to test.

The second section of the block diagram handles the vector functions. By having more vector functions, the user is able to see how the processors work for in different applications. This is important because often users want to determine the time a processor will take to execute a subset of instructions most important to their applications that is generally gauged by GFLOP/s [32, p. 264]. Based on what operation number the user gave, this section will determine which function should be called based on the operation number the user. The module will handle the delegation of the operation call then also storing the average latency to print to the text file. The operations are different vector functions that will be processed by the Xeon Phi 7210 and Xeon E5-4669 v4. The goal is to create processing curves for each of the following operations:

1. Hand-coded complex vector add operation
2. Intel Math Kernel Library (MKL) in single thread mode
3. MKL in multi-threaded mode
4. Complex scalar-vector multiply operation
5. Complex vector-matrix multiply operation

In addition to these operations, the stretch goal of the project would be to include the complex FFT, complex IFFT, and complex vector-multiply hand-coded implementations. In 3.2 the timeline of the project aims to complete these stretch goals as well, but this will be dependent on the first six vector functions to be complete.

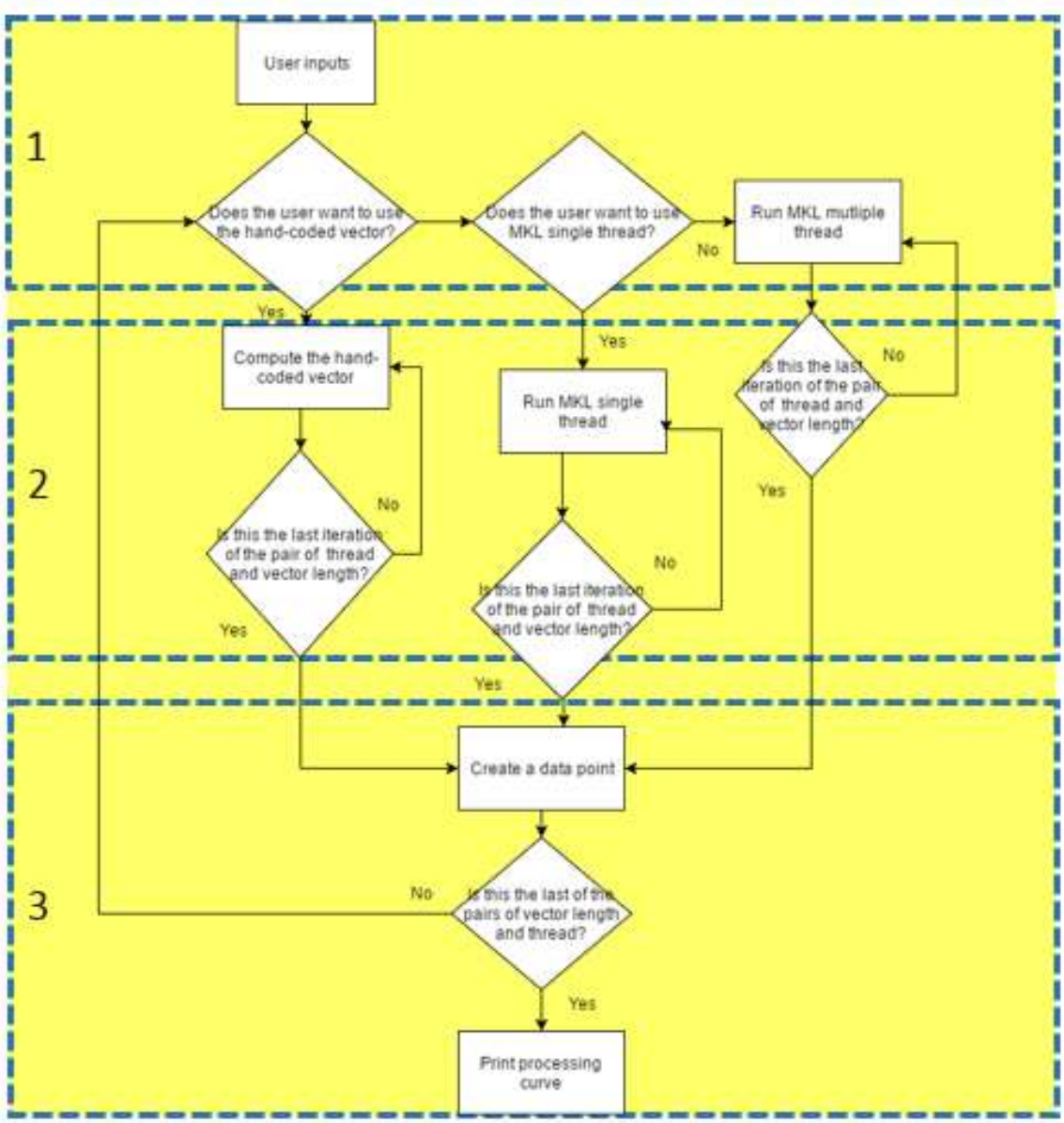


Figure 6: A logic block diagram of the signal processing comparison tool divided into sections. The first section is the user inputs. The second section is the computation of the vector functions. The third section is the processing curve output.

The third section handles the latency data points and producing the graph. The data points will be the average latency to compute the vector operation for all of the threads. When there are multiple threads the average latency will be taken for each thread then averaged together to put in text file. There is a potential that creating the two different processors on one graph in the Linux

Intel C compiler will prove too complicated, which will then force graph to be created in a different program. If the processing curve is created in a different program then the data points will be written to a text file and the program will have to be called on each processor separately.

Afterwards the data points will be analyzed to see if Xeon Phi 7210 or Xeon E5-4669 v4 is able to perform better. The better performance will be based on how fast either processor can compute the vector functions

3.2 Timeline and Management

To be able to complete the project within the fast pace of a term, a proposed schedule was created to ensure the project is completed successfully, which can be shown in the Gantt chart in Figure 7. The Gantt chart only includes the tasks to complete the software tool since the research and report writing will be done continuously throughout the project. The first two weeks are scheduled to create the main and vector add program. Ideally, on November 17th during the weekly advisor meeting the first set of processing curves will be reviewed to check that the graphs are meeting the project expectations. After the first review, any revisions will be made and the previous tests will be rerun. Afterwards, the other operations will be added to the project to allow the user to test the performance of different applications. Subsequently, the second review meeting will happen before Thanksgiving break and once the recess is over the revisions can be implemented and run on a complex scalar-vector multiply operation and complex vector-matrix multiply operation. If the Gantt chart schedule is followed and timing allows, the project will have the opportunity to work on the complex FFT, complex IFFT, and complex vector-multiply hand-coded, which are categorized as stretch goals.

MONTH	November																								December							
DATE	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	1	2	3	4	5	6	7	8
Create main to handle program selection																																
Create vector add program																																
First set of processing curves																																
First review software tool																																
Add different implementations of complex vector add operation																																
Second review software tool																																
Create complex scalar-vector multiply operation and complex vector-matrix multiply operation																																
Complex FFT and complex IFFT																																
Complex vector-multiply hand-coded																																

Figure 7: Proposed Gantt chart for creating the signal processing comparison tool to complete the project successfully within the seven week term.

3.3 Chapter Summary

The proposed approach is established to give the project a set starting point. As noted in the beginning of this chapter, there may have to be changes to the comparison tool as issues arise and different design approaches may prove more appropriate for the project. In addition, more than likely that setbacks or tasks completing sooner than expected that will affect the proposed timeline.

4. Methods and Implementation

This chapter will describe the implementation of the comparison tool and the design decisions throughout the process. The signal processing comparison tool was implemented in the Linux operating system and used the C++ Intel compiler. Throughout the implementation the C Intel compiler was used, but to be able to use different Intel applications the C++ Intel compiler was needed. In the proposed approach, it was suggested that the comparison tool could create one graph with both the Xeon Phi 7210 and Xeon E5-4669 v4 processing curves on it. Through further analysis of the operating system, it was determined that having a program switch between the Xeon Phi 7210 and Xeon E5-4669 v4 in one call would require to create a script that would ssh between the two processors in parallel. While this is possible, in the scope of project timeline it was more feasible to run the comparison tool on the Xeon Phi 7210 and then run it on the Xeon E5-4669 v4 to create two sets of data. Both sets of data would then be pulled into Microsoft Excel to produce the graphs.

In this chapter the following sections will correlate with the logic block diagram in Figure 6. Each of the sections outlined with dotted lines will be described as stage of implementation. The three stages of implementation can be described as the user interface, the computation of data points, and the processing curves. Section 4.1 discusses the implementation of the user interface. Sections 4.2, 4.4, and 4.5 discusses the implementation of the computation of data points. Section 4.3 discusses the implementation of the processing curves. Finally, in Section 4.6 setbacks of the project are discussed.

4.1 User Interface

The user interface encompasses the first third of the logic block diagram in Figure 5. This third of the logic block diagram includes the user giving inputs to the comparison tool and then the higher level of program execution.

First, the user interface was designed to ask the user for data to create three processing curves from one program call. Once the program is called in the Linux command window the user is prompted to give three numbers of threads, four vector lengths, the number of iterations, and which operation to compute. To tell the program which operation to compute, each vector function is defined by an operation number within the comparison tool. For instance, the hand-coded vector add was set to the operation number 1. In Table 3, a list of the proposed operations and operation numbers is provided.

Table 3: Proposed operations in the signal processing comparison tool and their operation numbers. The operation numbers are for the user to tell the program what operation to find the latency for.

Operation	Operation Number
Hand-coded complex vector add operation	1
Hand-coded complex vector multiply operation	2
Hand-coded complex vector multiply and add operation	3
Intel Math Kernel Library (MKL) in single thread mode	4
MKL in multi-threaded mode	5
Complex scalar-vector multiply operation	6
Complex vector-matrix multiply operation	7

The operation numbers in Table 3 allow for the program to determine what the operation it should compute the latency for. If the project allowed for the stretch goals to be achieved the operation numbers would be added after the complex vector-matrix multiply operation. To reduce human error, if the user types an operation number that is not provided in Table 3, the program will send a message stating that the number does not have an associated operation and then it will list all of the operations and their correlating operation numbers.

After the user gives the program the expected parameters the program will follow the block diagram in Figure 9. However, before implementing the block diagram in Figure 9, the block diagram in Figure 8 was attempted. In Figure 8, the goal of the block diagram was to have the main program handle the user inputs, which would then pass on the user inputs to the source file, average. The file, average, would determine which vector operation should be run then it would pass the information to the proper operation based on the operation number. The vector functions would handle the thread creation, how many times the vector operations should be run for each scenario, and the thread termination, which is shown in Figure 8.

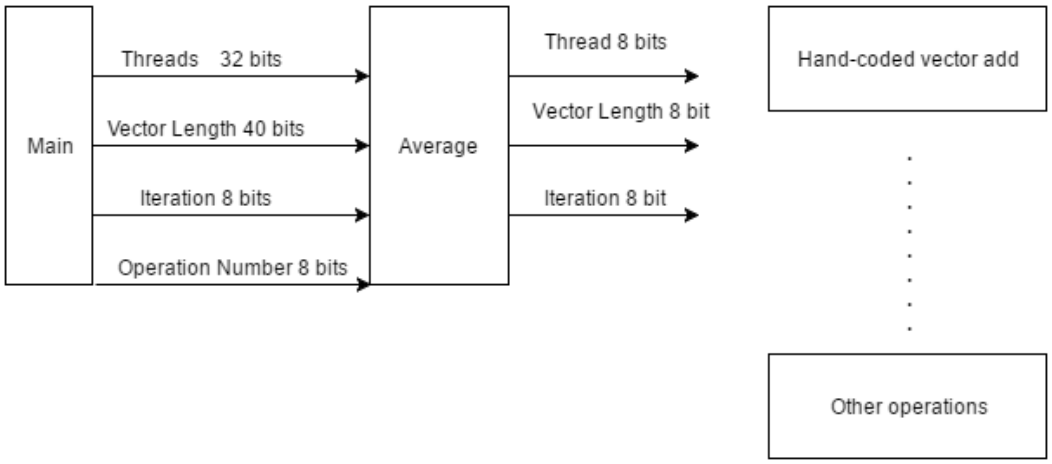


Figure 8: Intended block diagram that had the hand-coded vector add source file handle all of iterations of the vector function. This block diagram failed due to where the iterations were being handled and an incompatible storage type for the two dimensional array.

However, as the design in Figure 8 was implemented, segment fault errors would occur and would cause memory dumps. These segment fault errors were caused by two different aspects of the design. First, the two dimensional array for storing the data points was incompatible because the original storage was causing undefined storage. Second, the iterations needed to be called within the vector add function not when the threads were created. In Figure 8 the iterations were completed each time the hand-coded vector add function was called, which could have caused too many threads to be created.

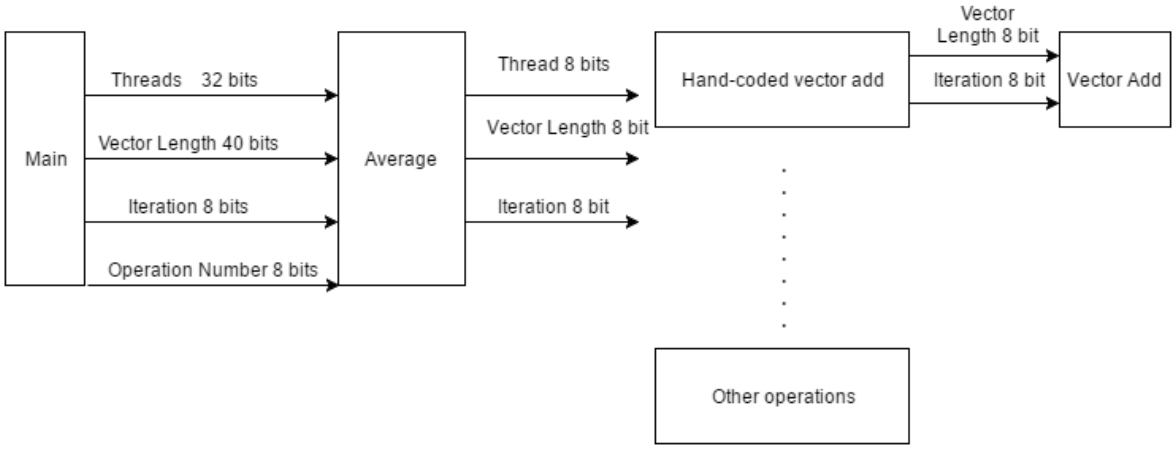


Figure 9: Block diagram implemented, which removed the segment fault errors by moving the iterations to the vector add function and changing the two dimensional array.

Once the framework shown in Figure 9 was implemented the segment fault errors were resolved by removing the two dimensional array to make a one dimensional array that was incremented by a global variable and allowing for the comparison tool to move the iteration loop for each scenario of thread number and vector length to the vector computation function. The aim of the block diagram in Figure 9 is to still have the main program handle the user inputs and pass on the user inputs to the source file, average. The file average would still determine which vector operation should be expected, yet now it passes the information to the proper operation file. The operation file depending on the operation to be computed will only create and terminate the threads. Due to this the operation file, as shown in Figure 8 as the hand-code vector add, passes the number of iterations and vector length to the vector add function that computes the vector add for the number of iterations on the threads created.

4.2 Creation of POSIX Threads

The vector functions are computed in two steps: creating the threads and running the vector function on these threads. For the hand-coded functions both of these steps are manually created, which differs from some of the libraries that handle both creation and execution on the threads. Through the implementation, the C Intel compiler was used for the proof of concept tests that compared the location the time is capture and compared using integers or floating points for the vector functions. However, after determining the need to make changes to improve the vectorization and optimization of the program some libraries required the program to change to the Intel C++ compiler.

The Intel C/C++ compilers allow for threads to be created in a variety of frameworks which are: Intel Threading Building Blocks (TBB), Intel Cilk Plus, OpenMP, C++11 Threads, and POSIX Threads, also known as Pthreads [34]. For this project, Pthreads were chosen because Pthreads can have performance gains since there is much less operating system overhead [34]. To create threads with the Pthread framework the following function is used:

```
int pthread_create(pthread_t *restrict thread,
        const pthread_attr_t *restrict attr,
void *(*start_routine)(void*), void *restrict arg);
```
[35]

This function's arguments are the array of thread IDs, attributes of the threads, the argument the thread should compute, and if there is a return to the function [35]. The thread array holds all of the IDs of the threads created; in the case of the hand-code vector add and implementing only one thread per core this would be a maximum of 64 threads on the Xeon Phi 7210 and a maximum of 22 threads on the Xeon E5-4669 v4. With the attributes are specified by attr and NULL can be used to create the threads with default attributes [35]. The default attributes of a thread are unbound, non-detached, with default stack and stack size, and with the parent's priority [36]. The threads non-detach and unbound, the threads do not have to hold onto information or resources when it has completed a task [37]. The remaining arguments of `pthread_create()` are the function that the threads will compute and arguments to pass to the function. In conjunction to the `pthread_create()` the `pthread_join()` is used to wait for a thread to terminate [36]. "The `pthread_join()` function blocks the calling thread until the specified thread terminates" [36]. The `pthread_join()` is as following:

```
int pthread_join(
      pthread_t thread,
   void **value_ptr);
```
[37]

The arguments of the function are the thread array, the same used in the `pthread_create()`, and the value returned by the thread. When there are multiple threads `pthread_join()` has all threads wait for the target thread to terminate [36]. The return value of `pthread_join()` is only used if the attributes are set to default, meaning the threads are detached because the `pthread_join()` synchronizes the threads, which is not needed when the threads are already detached, also known as bound [36]. In the original implementation the comparison tool used the default attributes when creating threads. However, unless necessary, non-detached threads should be avoided as they cause threads to wait, which is not optimal for the latency [36]. Also, using the attribute object the threads become more portable and it simples the state specification [38]. As a result of this, to make the threads portable and detached the scope of the attribute object was set to `PTHREAD_SCOPE_SYSTEM` which makes the threads bound [39].

Once the threads have been created the vector function will run. While the pthreads were creating and joining correctly, after the first design review of the tool, there were some design changes that were recommended to modify. To record how these modifications impacted the tool, graphs of latency verse vector length were created for each thread to record the performance of the tool with and without the modification. In Section 4.1 the rationale behind the design decision was explained as to why the iteration loop of the comparison tool was moved from the function that created the threads to the function that computed the vector function. Originally, the time was recorded in the function that created the threads. In the first design review, it was advised to move the time capture of the latency to exclude the create and terminate of the threads. When the thread creation and termination were included in the latency, the latency was recorded in the average source file, as shown in Figure 9. However, including the creation and termination of the threads may increase the latency of the vector functions. To test the impact of where the latency time was captured, the latency of each vector length was recorded and saved into an array to later print in the text file. To show the impact of where the time was recorded, two graphs to compare the performance of the Xeon Phi 7210 (in Figure 10) and Xeon E5-4669 v4 (Figure 11) that takes the latency including and excluding the creation and termination of the threads. In both graphs if the

time was captured in average.c the create and termination of threads were included in the latency and if the time was captured in the hand_code_add.c the create and termination of threads were excluded in the latency.

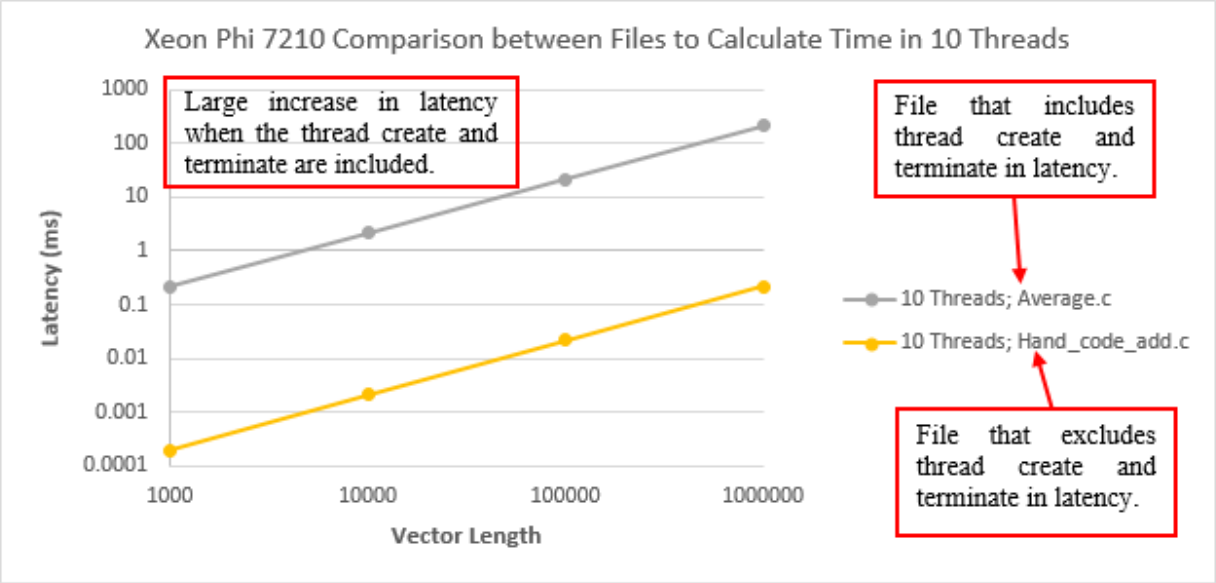


Figure 10: Xeon Phi 7210 1000 iterations for 10 threads comparing including and excluding create and termination of the threads. As expected excluding the create and termination of threads, in the file hand_code_add.c achieved a lower latency. This is shown as the entire processing curve in the hand_code_add.c file obtained a lower latency than the latency in the average.c

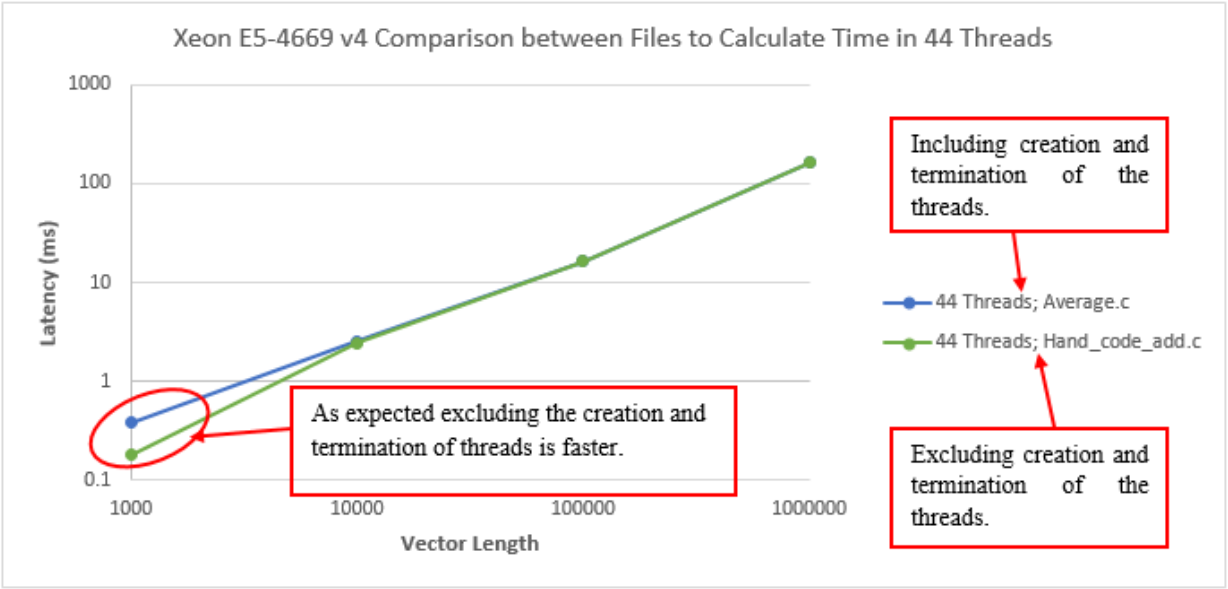


Figure 11: Xeon E5-4669 v4 1000 iterations for 44 threads comparing including and excluding create and termination of the threads. As expected excluding the create and termination of threads, in the file hand_code_add.c achieved a lower latency as highlighted in the circle above.

Overall, the graphs showed that in both the Xeon Phi 7210 and the Xeon E5-4669 v4 excluding the create and termination of the threads improved the latency and in some cases a fairly significant amount of 0.8ms. This was expected since creating and terminating the threads takes time. Because of this finding, the time capture was moved to exclude the create and terminate of threads because the goal of the project is only to test the performance of the machine when it is computing the vector function. To be able to see the comparison of the latency of including or excluding the create and termination of all of the threads on both the Xeon Phi 7210 and the Xeon E5-4669 v4 reference Appendix A.

Another recommended modification was to test the difference between using a floating point value or integer value in the vector functions. To determine if the type of vector function impacted the performance, the vector add was computed with floating point and with integers on the Xeon Phi 7210, shown in Figure 12, and the Xeon E5-4669 v4 as shown in Figure 13. In the graphs the scenarios the computations excluded the creation and termination of the threads in the timing calculations. From the graph in Figure 12 it is shown that Xeon Phi 7210 improved the latency when floating point types were used for the vector add function. The latency on the Xeon Phi 7210 improved significantly across 60 threads, which is shown in Appendix B. This improved latency is expected as the ISA of the Xeon Phi 7210 is meant to optimize floating point numbers. On the contrary, the Xeon E5-4669 v4 did not change in latency in almost all scenarios when using the floating point vector add. This was also anticipated as the ISA of the Xeon E5-4669 v4 was not meant to optimize floating point types. To see the other graphs of the remaining scenarios on both the Xeon Phi 7210 and Xeon E5-4669 v4 reference Appendix B.

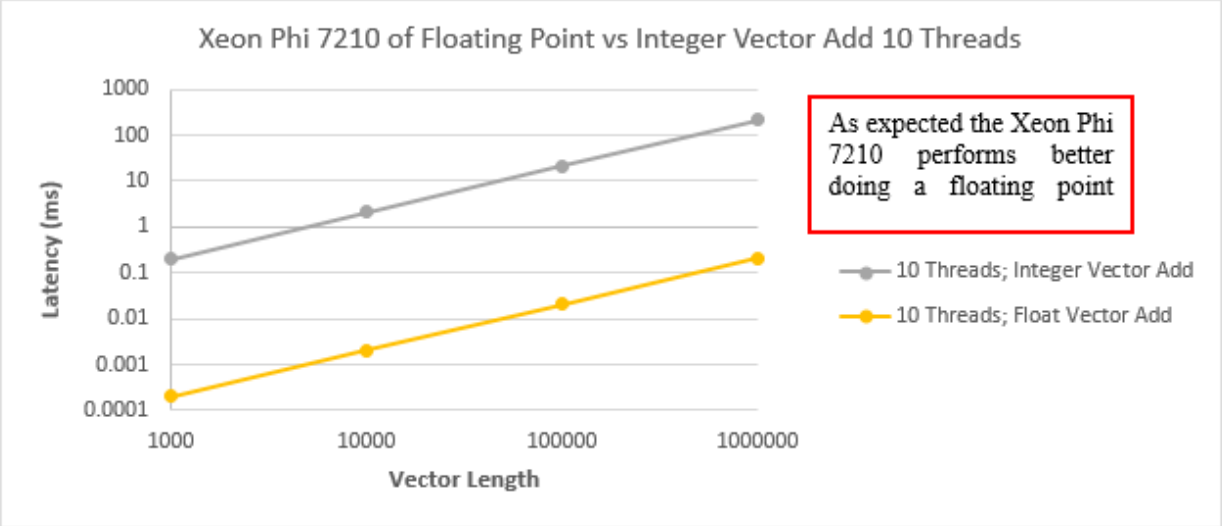


Figure 12: Xeon Phi 7210 1000 iterations for 10 threads comparing the integer vector add and float vector add. The float vector add achieved a much lower latency, which is expected as the ISA of the Xeon Phi 7210 is meant to handle floating point numbers.

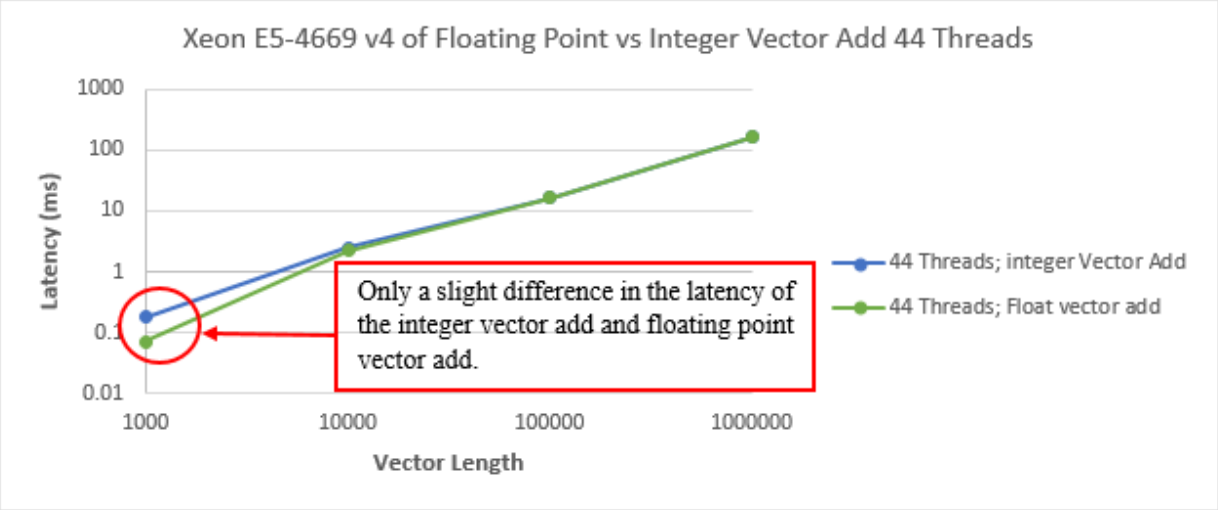


Figure 13: Xeon E5-4669 v4 1000 iterations for 44 threads compares floating point vector add and integer vector add. This graph differs from the Knights Landing showing that the float vector add did only achieved a slightly better latency than the integer vector.

After completing the two comparisons on the Pthread settings, the presentation of the results on the graphs was reviewed. Overall it did not seem that the graphs were comparing data in comparable terms because with a longer vector length it is expected that there is a longer latency. However, it did not seem to be a good representation of the performance of the processor.

4.3 Processing Curves

As the first set of data was collected, it became apparent that showing the processing curves as latency verse vector length was not a good representation of the performance. The latency was not a good representation of the performance because it is expected that with a longer vector length the time will be longer since there is more work to do. However this does not necessarily equate to the performance of how fast the functions are computing. From this, it was decided to change the processing curves from latency (*ms*) to GFLOP/s to be able to determine how fast the processors were executing the function. As described in Section 2.3, the GFLOP/s is a better representation of the performance of the machine because it accounts for not only the clock speed, but how many instructions are handled per tick. The updated graphs compare the theoretical GLFOPs to the GFLOP/s achieved for each processor. Additionally, there is a graph that compares the Xeon E5-4669 v4 to the Xeon Phi 7210 for each vector function.

The theoretical GFLOP/s for the Xeon Phi 7210 is:

GFLOP/sec =16 (SP SIMD Lane) x 2 (FMA) x 1.1 (GHZ) x 60 (# cores) = 2112 for single precision arithmetic [40]

The theoretical GFLOP/s for the Xeon E5-4669 v4 is:

GFLOP/sec =8 (SP SIMD Lane) x 2.2 (GHZ) x 22 (# cores) = 774 for single precision arithmetic

While these are the theoretical GFLOP/s, this is does not necessarily equate to the vector add function computed in the processing tool. Because of the work being done on the tool, there

will be a reduction factor for the number of cores and floating point multiply add (FMA). The FMA factor of two is divided out from the predicted GFLOP/s since the vector add does not include the multiply, therefore it will not need the factor of two in its calculation. Also the number of cores will have to be reduced since the vector function calculated for the average of each thread on one core. That means even though the vector add is expected over more than one thread, the signal processing comparison tool takes the average of the times to report the average latency for one thread. Additionally, since the Xeon Phi 7210 and the Xeon E5-4669 v4 have a different number of cores it will be best to compare the number of GFLOP/s per core to have more comparative terms for the two machines. Using that reduction factor the expected GFLOP/s per core will be:

The expected GFLOP/s per core for the Xeon Phi 7210 is:

GFLOP/sec per core =16 (SP SIMD Lane) x 1 (FMA) x 1.1 (GHZ) x 1 (# cores) = 17.6 for single precision arithmetic

The expected GFLOP/s per core for the Xeon E5-4669 v4 is:

GFLOP/sec per core =8 (SP SIMD Lane) x 2.2 (GHZ) x 1 (# cores) = 17.6 for single precision arithmetic

From these calculations, we expect both the Xeon Phi 7210 and Xeon E5-4669 v4 to compute the same GFLOP/s per core. To calculate the actual GFLOP/s per core from the data, latency achieved in seconds will be multiplied by 10^{-9} to convert to giga-floating point operations per second. The function is described as:

The achieved GFLOP/s per core for the Xeon E5-4669 v4 and Xeon Phi 7210 is:

GFLOP/sec =10^{-9} x latency achieved (s)

With this, the results from the signal processing tool can be compared in two ways: (1) the achieved GFLOPs/ per core on the Xeon Phi 7210 verse the Xeon E5-4669 v4, and (2) each processors achieved GFLOP/s per core verse its expected GFLOP/s per core.

4.4 Memory Optimization

After the data was calculated to better represent the performance of the processors, it was realized that the Xeon Phi 7120 was not optimizing as it should have been. From the vectorization reports it was found that the hand_code_add had a vector speed up of 4.86 and was only reaching 30% of its potential. To try to vectorized better, the memory was optimized by aligning the memory of the vector functions and changing the memory mode of the Xeon Phi 7210 to cache mode.

To make efforts to reach the potential speed up, the allocation of memory was changed to align the vectors. Data alignment creates data objects in memory for specific byte boundaries that increases efficiency of data loads and stores on the processor [41]. Aligning memory optimizes the program by aligning the base-pointer to the space allocated for the array and by making the starting indices benefit from alignment properties for each vectorized loop [41]. This required the header file tbb_allocator to be included to have the ability to allocate memory alignment. However, the tbb_allocator file is written in C++, which forced the signal processing tool to be moved from the Intel C compiler to the Intel C++ compiler. Fortunately, this did not cause many setbacks, but it did require manipulating some of the code to work with the new compiler. During the initial

changes to the new compiler, the program was only working for the scenarios with one thread. While the vector align was only working for one thread, the Xeon Phi 7210 was able to achieve 18.4 GFLOP/s per core for one thread with a vector length of 1,000,000.

Even though this achieved GFLOP/s per core was more than expected, it was not accurate because after the issue with changing compilers was resolved the achieved GFLOP/s for 10 and 60 threads on the Xeon Phi 7210 was not even 1 GFLOP/s per core. After analyzing these results, it was determined that the synchronizing of the threads was not correct nor was the memory mode of the Xeon Phi 7210. In the next section, it will be discussed how to improve the synchronization of threads, but in conjunction with changing the memory mode. By configuring the MCDRAM memory mode to cache, it should extend the cache level which could potential speed up the performance of the machine [42]. To change the memory mode of the Xeon Phi 7210, the BIOS setting must be configured to MCDRAM cache mode. Once this setting is configured the NUMA architecture should have the MCDRAM configured on one NUMA node [42]. However, once changing MCDRAM mode, there was only a slight speed up in the performance.

4.5 Synchronizing the Threads

Once the compiler was changed, the data was still achieving higher latencies than expected. It was then realized that the multiple threads were not synchronizing with each other. To synchronize Pthreads there are multiple methods to implement the synchronization which include: barrier waits, semaphores, mutex, and conditional variables [43]. For the implementation of the signal processing tool, barrier waits were used to have the threads synchronize. The barrier wait has all of threads wait for the participating threads to be created until the barrier releases the threads to compute the function [44]. This means that all threads must wait until the slowest thread, or last created thread, is at the barrier before continuing [43]. The implementation of the barrier wait was proven successful by printing out before and after the barrier wait call which showed that all of the threads waited before the barrier before computing the vector add function.

While synchronization of the threads was successful, it was found that it caused race issues between the threads. The race condition was when the threads computed the average time and stored the values to the average time array. The race issue caused multiple threads to write their average times to one location in the array which caused the latency written to the text file much longer since it was not the average latency of one thread, but multiple threads. To remove this issue all of the threads were given a unique ID. The unique ID would tell the threads which location in the average time array the thread could write to. The threads were given unique IDs when they were created which was passed in the struct argument in the `pthread_create` function. In addition to giving the threads unique IDs, the thread affinity was set to unique CPUs. Thread affinity optimizes the cache performance and overall performance of the threads by binding the CPU resources to one specific thread [45].

4.6 Vectorization Problems

After synchronizing the threads correctly, it was found that the Xeon Phi 7210 was only able to achieve 20% of the expected GFLOP/s per core. From this, it was predicted that the program was not optimizing correctly. To determine if the program was optimizing correctly, the tool was

tested to see whether or not the program was vectorizing. This was done by using the `-no-vec` in the compiler to show if the program had different results when it was not vectorizing verses when the tool was vectorizing. Once completing this test it was found that there was no difference in the GFLOP/s per core achieved.

At this point it was unknown why the program was not vectorizing correctly or achieving close to the expected GFLOP/s per core. Through research one potential issue that was found involving a bottlenecking problem within the vector function that was trying to compute too many iterations for the speed of the memory bandwidth causing a stall [46]. To get rid of this bottlenecking issue, the vectors should be broken into blocks length that the cache can handle, which is 256 floating point blocks [46]. This means if the user tells the program to compute a vector length of 1000 then the program will break the 1000 vector length into three blocks to compute the length in sections to compute the vector add. However, with this change and using the MIC-AVX512 compiler link, the Xeon Phi 7210 was only able to achieve 50% the expected GFLOP/s per core, which was the best GFLOP/s achieved from on the Xeon Phi 7210. The graph showing the comparison of the Xeon Phi 7210 achieved GFLOP/s per core compared to the expected GFLOP/s per core can be seen in Figure 14.

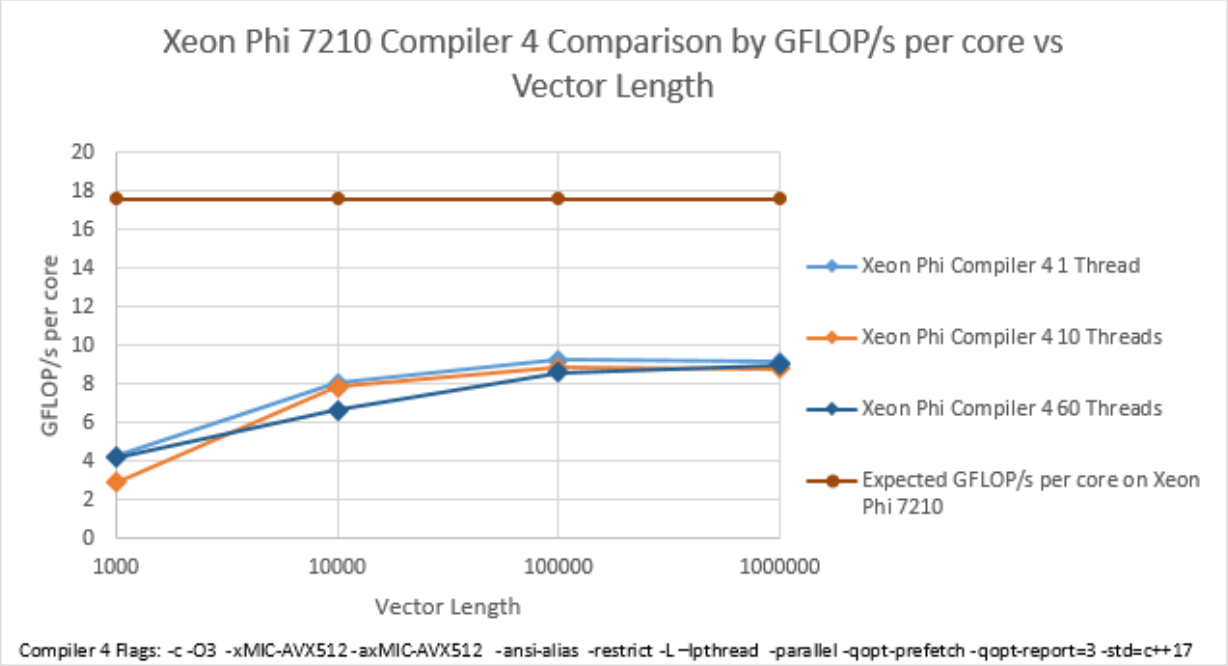


Figure 14: Graph comparing the expected verse actual GFLOP/s per core on the Xeon Phi 7210. The graph shows that the achieved GFLOp/s per core is 50% the expected for the Xeon Phi 7210. The compiler links of the data are on the bottom of the graph.

4.7 Chapter Summary

Throughout the implementation process there many changes to the tool from design reviews, discoveries from the results, and further understanding of how Pthreads functionality. The learning curve with Pthread creation and synchronization was much more than anticipated. By creating graphs when modifications were made to the Pthread implementation, it was also possible

to review how the data from the performance tool was displayed. After changing the data from latency in milliseconds to GFLOP/s per core, it was easier to see the performance of Xeon Phi 7210 comparatively for each vector length. Being able to compare the performance of the Xeon Phi 7210 for each vector length showed issues with the synchronization of the threads and vectorization of the program. These issues caused several setbacks, which limited the GFLOP/s per core performance to only be able to achieve 50% of the expected GFLOP/s per core rate. The finalized code for the tool can be found in Appendix C. To be able to run the current program reference Appendix D.

5. Results and Discussion

As mentioned in Section 4.7 the best achieved GFLOP/s per core on the Xeon Phi 7210 was 50% of the expected GFLOP/s per core. Due to the setbacks with creating the threads and synchronizing the threads, the project ran out of time to resolve the vectorization problems. Since the goal of the project is to achieve close to the expected GFLOP/s per core on each processor, there were several tasks that will be left incomplete from the project goal. During the tool reviews it was determined that it would be more productive if the tool was more accurate than had multiple vector function operations. As it can be seen in Figure 15, the tasks that were left incomplete were the hand-coded complex vector multiply operation, hand-coded complex vector multiply and add operation, MKL in single thread mode, MKL in multi-threaded mode, complex scalar-vector multiply operation, and complex vector-matrix multiply operation. These remaining functions along with the stretch goals will be put in the future recommendations in Chapter 6.

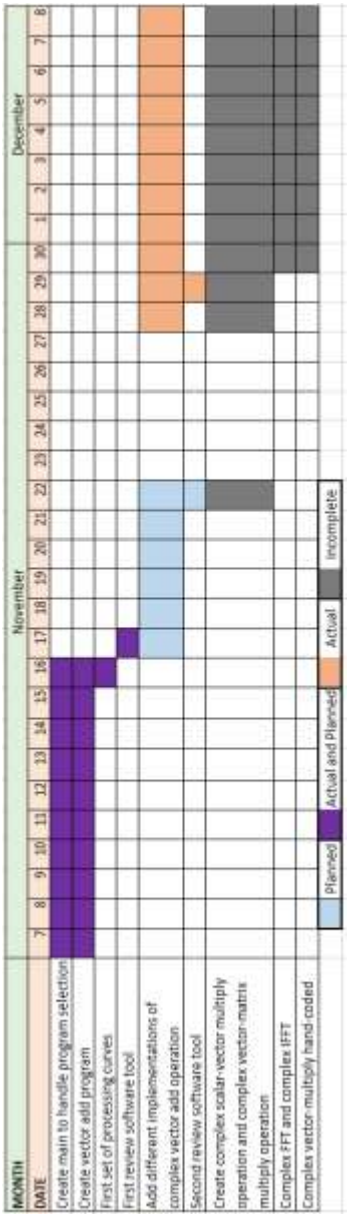


Figure 15: Actual timeline of the project. The only function that was implemented was the vector add function. The remaining functions were left outstanding. There is a key on the side of the Gantt chart which shows when events were completed on schedule or after schedule.

4.1 Vector Add Results

While only 50% of the expected GFLOP/s per core of Xeon Phi 7210 was achieved. The results of the vector add were still able to be analyzed. To be able to compile the program on the Xeon E5-4669 v4, most recent advance vector instruction set that worked on it was the CORE-AVX2 compiler link [47]. The CORE-AVX2 is meant to optimize the Intel Xeon Processor E5 v4 Family [47]. However, this does not optimize the Xeon Phi 7210. MIC-AVX512, the compiler used in Section 4.6, is meant to compile the Intel Xeon Phi processor x200 product family [47]. To be able to compare the Xeon Phi 7210 and the Xeon E5-4669 v4 with the same compiler, CORE-AVX2 compiler link was used to produce data sets for both compilers which is shown in Figure 16. The entire makefile with the CORE-AVX2 compiler link is in Appendix E.

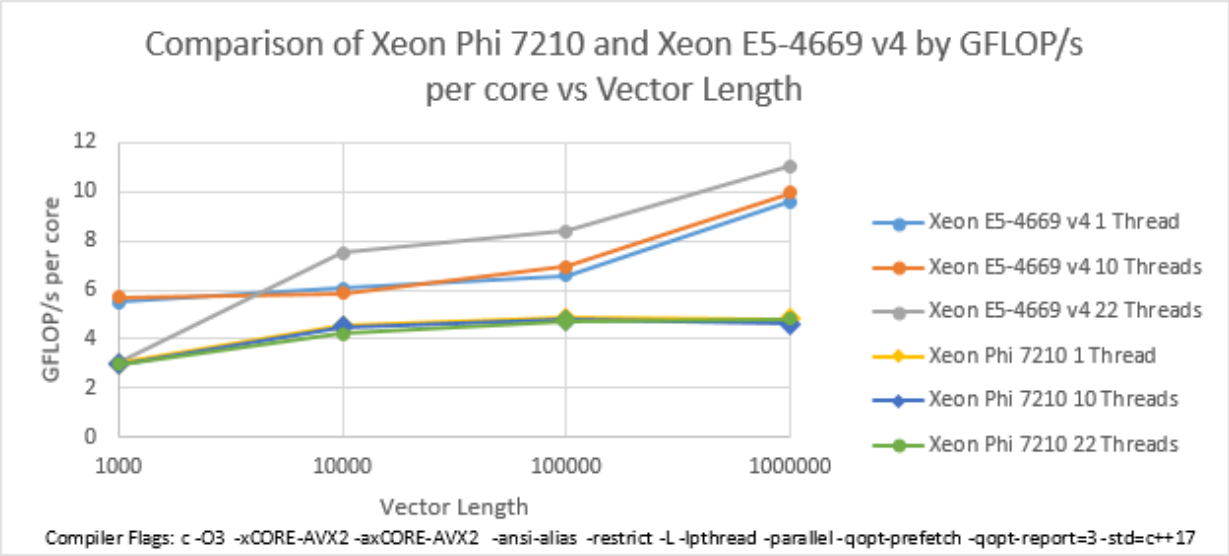


Figure 16: The graph compares the Xeon Phi 7210 and the Xeon E5-4669 v4 for the compiler links of the data are on the bottom of the graph. As expected the Xeon E5-4669 v4 performs better than the Xeon Phi 7210 because the advance vector instruction CORE-AVX2 is meant to optimize the Xeon E5 v4 family.

As expected the Xeon E5-4669 v4 performed better than the Xeon Phi 7210 because the advance vector instruction CORE-AVX2 is meant to optimize the Xeon E5 v4 family. To show how much the compiler links impact the performance of the machine, the compiler MIC-AVX512 link and the CORE-AVX2 link were both run on the Xeon Phi 7210 and compared to each other on the graph in Figure 17. All of the compiler options for the graph in Figure 17 can be found in the entire makefile with the CORE-AVX2 compiler link in Appendix E and the entire makefile with the MIC-AVX512 compiler link in Appendix F. The graph in Figure 17 demonstrates how the MIC-AVX512 link performs double the GFLOP/s per core rate than the CORE-AVX2 link. The CORE-AVX2 link only achieved 25% the expected GFLOP/s per core for the Xeon Phi 7210. This shows how important it is to use the compiler links that optimize that processor.

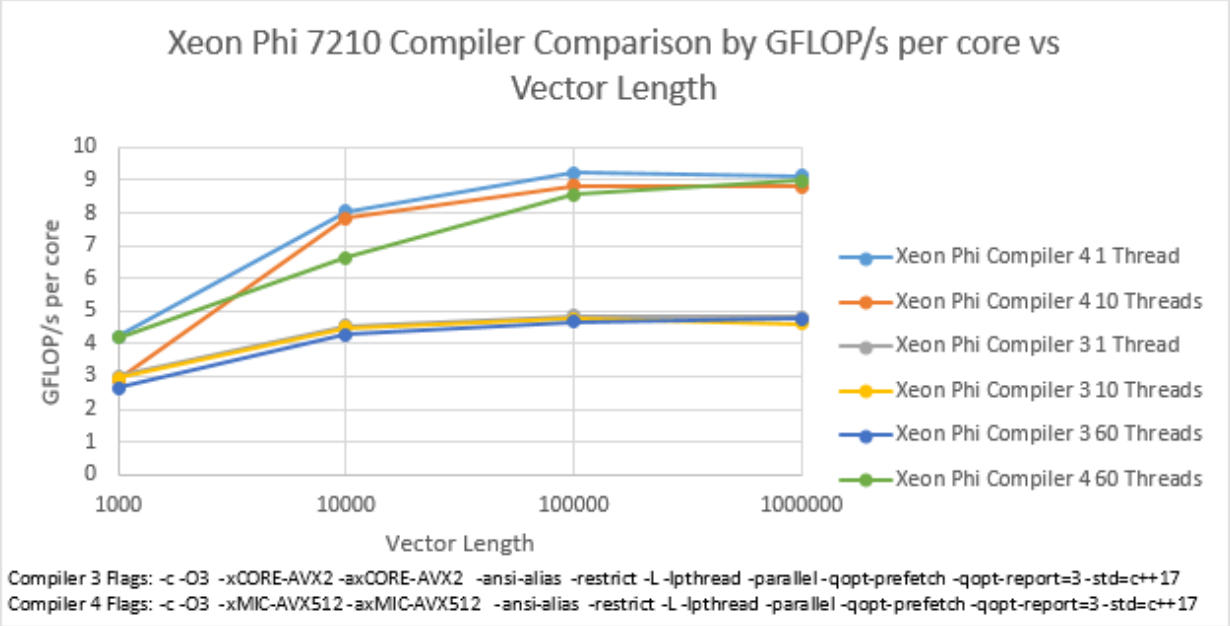


Figure 17: The graph compares the compiler MIC-AVX512 link and the CORE-AVX2 link on Xeon Phi 7210. The compiler options for the compiler MIC-AVX512 link and the CORE-AVX2 link are on the bottom of the graph.

Since it was proved as expected that the MIC-AVX512 would optimize better than the CORE-AVX2 on the Xeon Phi 7210, the Xeon Phi 7210 and the Xeon E5-4669 v4 were compared with their respective optimized advance vector instruction sets. The graph in Figure 18 has the Xeon Phi 7210 with the MIC-AVX512 compiler and the Xeon E5-4669 v4 with the CORE-AVX2 compiler. In the graph it can be seen that the Xeon Phi 7210 with the MIC-AVX512 compiler and the Xeon E5-4669 v4 with the CORE-AVX2 compiler have comparable results. All of the compiler options for the graph in Figure 18 can be found in the entire makefile with the CORE-AVX2 compiler link in Appendix E and the entire makefile with the MIC-AVX512 compiler link in Appendix F. Neither processor is close to the expected GFLOP/s per core as described in Section 4.3.

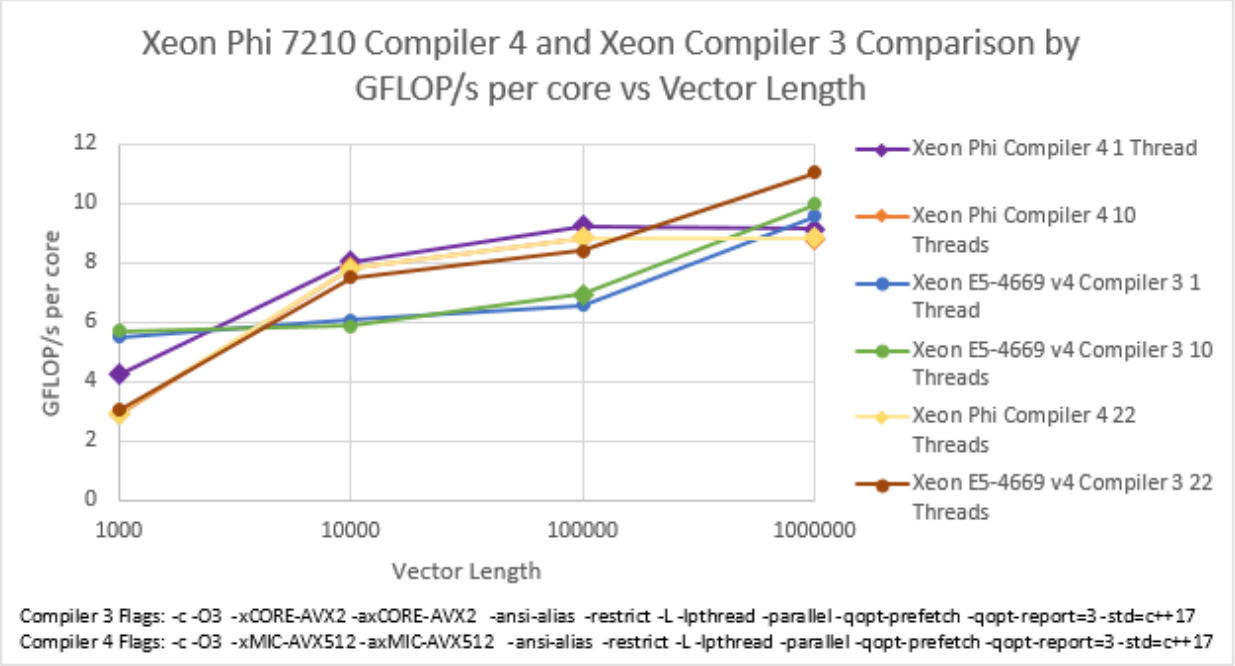


Figure 18: The graph compares Xeon Phi 7210 with the compiler option 4 shown below the graph and the Xeon E5-4669 v4 with the compiler option 3 shown below the graph. As can be seen the Xeon E5-4669 v4 and the Xeon Phi 7210 have comparable results.

From the results in the graph in Figure 18, the best achieved GFLOP/s per core for the Xeon E5-4669 v4 was 11.04 GFLOP/s per core with 60 threads with 1,000,000 vector length and the best achieved GFLOP/s per core for the Xeon Phi 7210 was 8.99 GFLOP/s per core with 60 threads with 1,000,000 vector length. While the Xeon Phi 7210 did not perform as well per core as the Xeon E5-4669 v4, the Xeon Phi 7210 performed a better overall GFLOP/s than the Xeon E5-4669 v4. Based on the information in Section 4.3, to calculate the GFLOP/s of the machine the GFLOP/s per core was multiplied by the number of cores on each processor. From this the Xeon Phi 7210 achieved GFLOP/s:

8.99 (GFLOP/s per core) * 60 (cores) =539.4 GFLOP/s

The Xeon E5-4669 v4 achieved GFLOP/s:

11.04 (GFLOP/s per core) * 22 (cores) = 242.88 GFLOP/s

While it has been recognized that the Xeon Phi 7210 should have achieved a better GFLOP/s per core, this computation still proves that the Xeon Phi 7210 achieves a better GFLOP/s than the Xeon E5-4669 v4 because it has more cores.

4.2. Chapter Summary

Although several of the proposed tasks of the project that impacted the implementation of the signal processing tool were not met, the project was still able to show a performance difference between the Xeon Phi 7210 and the Xeon E5-4669 v4. A major part of the project was to analyze the computer architecture in different applications and how they impact the performance. While

the performance tool was only able to test one application that was not entirely accurate, it was able to prove how the computer architecture does impact the performance of the processor. It was shown that even with a slower GFLOP/s per core on the Xeon Phi 7210, it had a higher GFLOP/s rate than the Xeon E5-4669 v4 because of the number of threads the Xeon Phi 7210 is able to compute on. Additionally, the calculations manipulated in Section 4.3 proved the background research in Section 2.3 that stated even though a processor has a higher clock speed it does not mean it will perform faster. While the Xeon E5-4669 v4 had almost double the clock speed of the Xeon Phi 7210, both had the expected GFLOP/s per core because the Xeon Phi 7210 had a larger register size and was able to do double the work during one clock cycle as the Xeon E5-4669 v4. Although the GFLOP/s per core rate was much lower than expected, the tool was still able to prove similar to the expected results in Section 2.3.

6. Future Recommendations

Since there were several tasks left incomplete and the Xeon Phi 7210 only achieved 50% its expected GFLOP/s per core, there are many areas for continued work with the tool. First, in the back research it was stated that the ideal amount is two threads per core active to allow for out-of-order processing, which can hide instruction and cache access latencies for many applications [23, p. 70]. Currently, the threads are each allocated to a unique core. It may be worth testing hyper-threading two threads per core to see if there is an impact on the performance. After this, some areas for continued tests are: the structure of the program, memory settings, and data collection.

To test the structure of the program, different methods of synchronizing the threads and setting the CPU affinity can be completed. As mentioned in Section 4.2, there are multiple ways to synchronize threads which are: mutex, semaphores, and conditional variables. Without enough time for the project, the multiple methods were not tested for performance. In many readings about synchronizing threads the mutex is recommended for users. The mutex may have advantages because it is only owned by one particular thread [43]. Also, all thread locks can create overhead since they are serialization points for critical sections [48]. The mutex has the ability to reduce some idling overhead by using a function called `pthread_mutex_trylock` [48]. A comparable function was not found with the pthread barrier wait call. This overhead idling may cause stalls with the barrier wait, which were not determined in the first attempt to synchronize the threads. As a results of this potential overhead idle, it is recommended to try another method to synchronize threads to prove whether or not this potential overhead is impacting the performance of the machine. Additionally, there could still be some setbacks to not having a priority of the threads once the barrier wait is unlocked. As it is set now any thread can start computing the vector add function once the barrier is unlocked. Due to this setting a priority to the threads may help increase the performance of the threads.

After tests on the tool structure are complete, there should be further investigation in the memory settings on the Xeon Phi 7210. In Section 4.4, it was mentioned that there was not a notable difference when the MCDRAM mode was set to cache. This was unexpected because the cache mode should have increased the usage of the memory bandwidth. The increased cache should have helped the vectorization problems mentioned in Section 4.7. By changing the cache mode, the Xeon Phi 7210 should have had increased bandwidth speed and able to handle more than 256 floating point values in cache. This unexpected problem may be caused by the compiler missing a flag or linker. To be able to test whether or not the MCDRAM set to cache mode worked, it would be worthwhile to check the assembly code to see if any Xeon Phi x200 specific instructions were utilized. Also there was no drop off from the cache with longer vector lengths. Since this was never investigated, future work should expand the plot to see the impact of the cache with more vector lengths.

Additionally, in Section 2.2.2, the NUMA memory access was described as a cluster feature of the Knights Landing Family. When the modes of the memory are changed on the Knights Landing Family it changes how many NUMA nodes are written too. In the implementation of the tool, none of the memory access settings were changed. Researching how to optimize the cluster mode settings may be worth investigating. Finally, in the Intel Xeon Phi Coprocessor High

Performance Programming manual many of the examples statically defined the memory for the vectors in the vector function [49]. While the tool does have user inputs for the vector lengths. It may be necessary to statically define the memory of the vector lengths to the largest possible vector length to obtain a higher GFLOP/s per core rate. While the code did include a pragma with the minimum iterations to optimize the code for the minimum number of iterations, this is another test to see if this will help increase the Xeon Phi 7210 performance.

Once the investigations with the memory settings are researched, modifications to how the time is captured in the tool should be examined. In Section 4.2, it was discussed how it was better to exclude the create and termination of threads when capturing the latency. While this improved the time capture of the tool, there are still areas to test for improvements in capturing the time. First, in the Intel Xeon Phi Coprocessor High Performance Programming manual many of the examples captured the time outside of the iteration loop [49]. This method excluded individual iterations of the vector add function and took the average of the time capture by dividing the final capture time by the number of iterations. The reasoning behind capturing the time outside of the iteration loop was not explained the in the Intel Xeon Phi Coprocessor High Performance Programming manual. However, taking the time of each iteration may slow down the potential speed up of the vector add functions. In conjunction with this, to make the standard deviation more accurate, it may be worthwhile to have the standard deviation of the threads calculated in the main thread and from the sum of all of the averages. While testing out the differences in capturing the time, it should also be tested to see if the location impacts the GFLOP/s per core. This would require the time on each thread to be captured and printed separately. If it is found that the location does impact the GFLOP/s, it should be researched to find if there is a way to set the CPU affinity to CPUs that are closer together. Testing these variations of the data collection may give insight as to where the tool can be improved.

Even after the tests on the structure of the program, memory settings, and data collection are complete, there should be further investigations to ensure that there is an in-depth understanding of the Knights Landing generation architecture. An in-depth understanding is critical for any future applications with the Knights Landing generation processors. After the accuracy of the vector add function is complete the hand-coded complex vector multiply operation, hand-coded complex vector multiply and add operation, MKL in single thread mode, MKL in multi-threaded mode, complex scalar-vector multiply operation, and complex vector-matrix multiply operation should be implemented.

7. Conclusion

Even though all of the project tasks were not completed, the project was able to show the impact of computer architecture. As discussed in Section 5.2, the tool showed that even with a slower GFLOP/s per core on the Xeon Phi 7210, it had a higher GFLOP/s rate than the Xeon E5-4669 v4 because of the number of threads the Xeon Phi 7210 is able to compute on. Additionally, the project showed that with a faster clock rate does not necessarily equate to a faster perform. This was proven, since the Xeon E5-4669 v4 had almost double the clock speed of the Xeon Phi 7210, both had the expected GFLOP/s per core because the Xeon Phi 7210 had a larger register size and was able to do double the work during one clock cycle as the Xeon E5-4669 v4. From the work completed, it was proven that with more threads a faster GFLOP/s rate can be achieved. Due to this, it is expected that the Xeon Phi 7290 will perform a better GFLOP/s rate than the Xeon Phi 7210, based off of the processor specifications shown in Table 4. Since it was proven that the clock speed and the amount of work per tick impact the performance, there is an additionally improvements for GFLOP/s rate since the Xeon Phi 7290 should be able to do the same amount of work per cycle that the Xeon Pho 7210 can for a slightly faster clock speed, which is shown in Table 4. From the results and specifications shown in Table 4, it may be possible that the Xeon Phi 7290 not only has a faster GFLOP/s rate, but a faster GFLOP/s per core rate than the Xeon Phi 7210.

Table 4: Overview of all of the Intel processors discussed within this report. Specifically, this table shows the major differences between in the Xeon Phi 7210 and Xeon Phi 7290 are more cores/threads, faster clock speed, and faster memory bandwidth. From the results, it may be possible that the Xeon Phi 7290 not only has a faster GFLOP/s rate, but a faster GFLOP/s per core rate than the Xeon Phi 7210.

Categories	**Intel Xeon Processor E5-4669**	**Intel Xeon Phi Processor 7210**	**Intel Xeon Phi Processor 7290**
Number of Cores	22	64	72
Number of Threads	44	256	288
Processor Base Frequency	2.20 GHz	1.3 GHz	1.50 GHz
Cache	55 MB	32 MB L2	36 MB L2
Max Memory Bandwidth	68 GB/s	102 GB/s	115.2 GB/s
Max Memory Size	1.54 TB	384 GB	384 GB

Overall, the MQP was more than just an undergraduate degree requirement for me. I was able to learn about Pthreads and a new level of computer architecture that I have never researched before. This project has prepared me for my future work at Raytheon and future studies on computer architecture.

8. References

[1] Texas Instruments, "Applications for Digital Signal Processors," Texas Instruments, 2016. [Online]. Available: http://www.ti.com/lsds/ti/processors/dsp/applications.page. [Accessed 20 November 2016].

[2] Analog Devices, "Analog Devices," Analog Devices, 2016. [Online]. Available: http://www.analog.com/en/design-center/landing-pages/001/beginners-guide-to-dsp.html. [Accessed 20 November 2016].

[3] Berkeley Design Technology, Inc, "Evaluating DSP Processor Performance," Berkeley Design Technology, Inc, Berkeley, CA, 2002.

[4] Intel , "Processor Benchmark Limitations," Intel, 2016. [Online]. Available: http://www.intel.com/content/www/us/en/benchmarks/resources-benchmark-limitations.html. [Accessed 11 December 2016].

[5] Intel Corporation, "Intel Xeon Processor E7 Family," Intel Corporation , 2016. [Online]. Available: http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-e7-family.html. [Accessed 20 November 2016].

[6] Intel Corporation, "Intel Xeon Phi Product Family," Intel Corporation, 2016. [Online]. Available: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html. [Accessed 20 November 2016].

[7] S. Akhter and J. Roberts, "Multi-Core Programming: Increasing Performance: Increasing Performance through Software multi-threading," *Intel Corporation,* 2006.

[8] S. Mishra, N. Singh Kimar and V. Roussea, System on Chip Interfaces for Low Power Design, Waltham, MA: Morgan Kaufman, 2015.

[9] S. Datta, "Method of communication between firmware written for different isntruction set architectures". US Patent 6081890 A, 27 June 2000.

[10] T. Austin, E. Larson and D. Ernst, "SimplerScalar: An Infrastructure for Computer System Modeling," *IEEE,* vol. 35, no. 2, pp. 59-67, 2002.

[11] U. Drepper, "What every programmer should know about memory part 1," *Lwn.net,* 21 September 2007.

[12] M. Rouse, "Cache Memory," TechTarget, 2016. [Online]. Available: http://searchstorage.techtarget.com/definition/cache-memory. [Accessed 2 November 2016].

[13] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowksi, T. Juan and P. Hanrahan, "Larrabee: A

Many-Core x86 Architecture for Visual Computing," *ACM Transactions on Graphics,* vol. 27, no. 3, pp. 18:1-18:15, 2008.

[14] T. Tian and C.-P. Shih, "Software Techniques for Shared-Cache Multi-Core Systems," *Intel Developer Zone,* 8 March 2012.

[15] D. Hatter. (2016). "What is the Difference Between Hyper Threading and Multi-Core Technology?," *Houston Chronicles* [Online]. Available: http://smallbusiness.chron.com/difference-between-hyper-threading-multicore-technology-58549.html

[16] J. Jeffers, J. Reinders and A. Sodani, "Knights Landing Overview," in *Intel Xeon Phi Processor High Performance Programming*, Cambridge, Elsevier Inc., 2016, pp. 15-25.

[17] B. Zhang, "Guide to Automatic Vectorization with Intel AVX-512 Instructions in Knights Landing Processor," *Colfax International,* 2016.

[18] M. F. Spear, V. J. Marathe, W. N. I. Scherer and M. L. Scott, "Conflict Detection and Validation Strategies for Software Transactional Memory," University of Rochester, Rochester, 2006.

[19] M. Ferdman. (2013) *Computer Architecture Prefetching* [Online] Available: https://compas.cs.stonybrook.edu/course/cse502-s13/lectures/cse502-L3-memory-prefetching.pdf

[20] J. Jeffers, J. Reinders and A. Sodani, "Programming MCDRAM and Cluster Modes," in *Intel Xeon Phi Processor High Preformance Programmming Knights Landing Editin*, Cambride, Elsevier Inc., 2016, pp. 25-63.

[21] M. Bauer, Oracle 8i Parallel Server, Redwood Shores, CA: Oracle Corporation, 1999.

[22] N. Manchanda and K. Anand, "Non-Uniform Memory Access (NUMA)," New York University, New York, New York.

[23] J. Jeffers, J. Reinders and A. Sodani, "Knights Landing Architecture," in *Intel Xeon Phi Processor High Performance Programming*, Cambridge, Elsevier Inc, 2016, pp. 63-85.

[24] G. Suh, L. Rudolph and S. Devadas, "Dynamic Partitioning of Shared Cache Memory," *The Journal of Supercomputing,* vol. 28, no. 1, pp. 7-26, 2004.

[25] M. K. F. Mehmet Ali, "Neural Networks for Shortest Path Computation and Routing in Computer Networks," *IEEE Transactions on Neural Networks,* vol. 4, no. 6, pp. 941-954, 1993.

[26] A. Sodani, "Knights Landing (KNL): 2nd Generation Intel Xeon Phi processor," in *2015 IEEE Hot Chips 27 Symposium*, Cupertino, CA, 2015.

[27] Intel Corporation, "Intel Xeon Processor E5-4669 v4," Intel Corporation, 2016. [Online]. Available: http://ark.intel.com/products/93805/Intel-Xeon-Processor-E5-4669-v4-55M-Cache-2_20-GHz. [Accessed 20 November 2016].

[28] Intel Corporation, "Intel Xeon Phi Processor 7210," Intel Corporation, 2016. [Online]. Available: http://ark.intel.com/products/94033/Intel-Xeon-Phi-Processor-7210-16GB-1_30-GHz-64-core. [Accessed 20 November 2016].

[29] "IBM Knowledge Center," IBM, [Online]. Available: http://www.ibm.com/support/knowledgecenter/ssw_aix_71/com.ibm.aix.genprogc/benefits_threads.htm. [Accessed 3 22 2016].

[30] Intel Corporation, "Get Faster Performance for Many Demanding Business Applications," Intel Corporation, 2016. [Online]. Available: http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html?wapkw=hyper+threading. [Accessed 20 November 2016].

[31] S. Furber, "GCSE Bitesize Computer Science," BBC , 2014. [Online]. Available: http://www.bbc.co.uk/education/guides/zmb9mp3/revision/9. [Accessed 13 December 2016].

[32] M. G. Schneider, J. Gersting and B. Brickman, Invitation to Computer Science, Boston, MA: Cengage Learning, 2015.

[33] U. B. M. LLC-UBM, "Know a Processor's Cache, Bus Speed -- Clock Speed Isn't Everything," *Computer Retail Week,* vol. 219, no. 8, p. 17, 1998.

[34] R. Florian, "Choosing the right threading framework," Intel, 31 May 2013. [Online]. Available: https://software.intel.com/en-us/articles/choosing-the-right-threading-framework. [Accessed 14 December 2016].

[35] The IEEE and The Open Group, *The Open Group Base Specifications,* The IEEE and The Open Group, 2004.

[36] The Oracle, "Basic Threads Programming," in *Multithreaded Programming Guide,* San Antonio, Sun Microsystems, Inc., 2001.

[37] Loic, "Loic OnStage," Loic OnStage, 17 October 2009. [Online]. Available: http://www.domaigne.com/blog/computing/joinable-and-detached-threads/. [Accessed 15 November 2016].

[38] The Orcale, "Thread Attributes," in *Multithreaded Programming Guide,* San Antonio, Sun Microsystems, Inc., 2001.

[39] T. Oracle, "Thread Create Attributes," in *Multithreaded Programming Guide,* San Antonio, Sun Microsystems, Inc., 2001.

[40] R. Rahman, "Intel Xeon Phi Core Micro-Architecture," Intel Corporation, 31 May 2013. [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture. [Accessed 20 November 2016].

[41] R. Krishnaiyer, "Data Alignment to Assit Vectorization," Intel , 7 September 2013. [Online]. Available: https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization. [Accessed 4 December 2017].

[42] L. Q. Nguyen, "Tutorial on Intel Xeon Phi Processor Optimization," Intel, 20 June 2016. [Online]. Available: https://software.intel.com/en-us/articles/tutorial-on-intel-xeon-phi-processor-optimization. [Accessed 17 December 2016].

[43] University of Mary Washington, "Barriers & Conditions," University of Mary Washington, [Online]. Available: http://cs.umw.edu/~finlayson/class/fall14/cpsc425/notes/07-conditions.html. [Accessed 17 December 2016].

[44] The Open Group , "The Open Group Base Speficiations Issue 7," The IEEE and The Open Group , 2008. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/. [Accessed 17 December 2016].

[45] A. Katranov, "Applying Intel Thread Building Blocks observers for thread affinity on Intel Xeon Phi coprocessors," Intel, 31 October 2013. [Online]. Available: https://software.intel.com/en-us/blogs/2013/10/31/applying-intel-threading-building-blocks-observers-for-thread-affinity-on-intel?wapkw=pthread+cpu+affinity. [Accessed 17 December 2016].

[46] P. Cordes, "Why vectorizing the loop does not have performance improvement," Stack Overflow, 5 July 2015. [Online]. Available: http://stackoverflow.com/questions/18159455/why-vectorizing-the-loop-does-not-have-performance-improvement. [Accessed 18 December 2016].

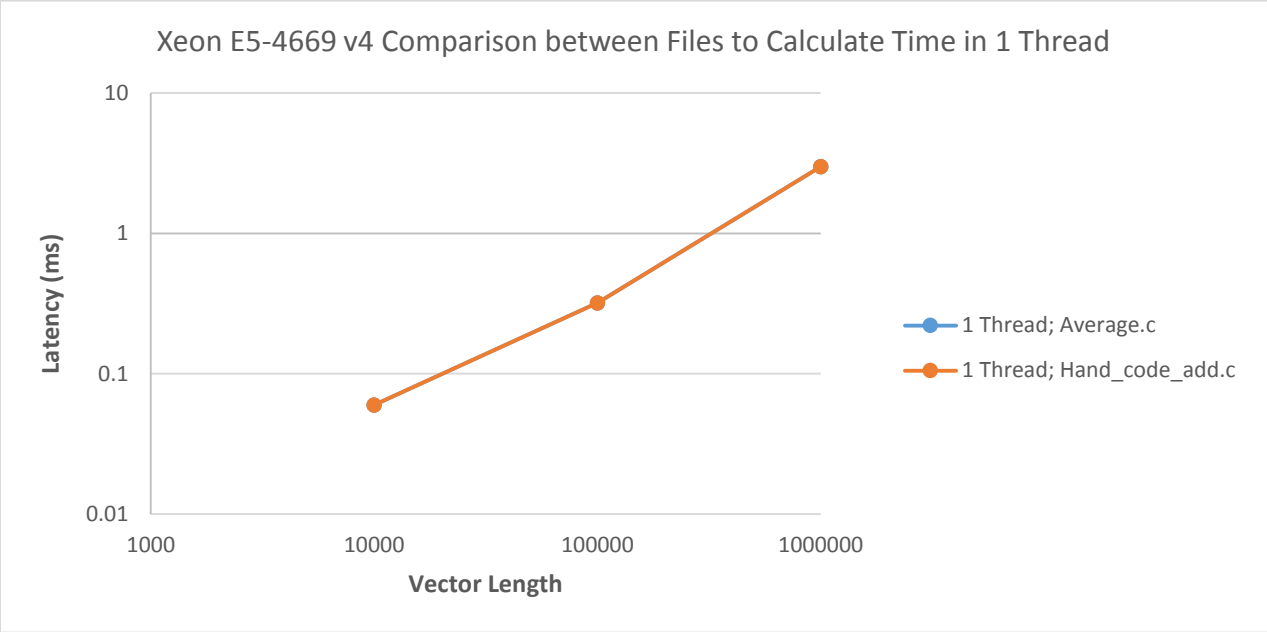
[47] M. Corden, "Intel® Compiler Options for Intel® SSE and Intel® AVX generation (SSE2, SSE3, SSSE3, ATOM_SSSE3, SSE4.1, SSE4.2, ATOM_SSE4.2, AVX, AVX2, AVX-512) and processor-specific optimizations," Intel, 24 Janaury 2010. [Online]. Available: https://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations?wapkw=core-avx. [Accessed 18 December 2016].

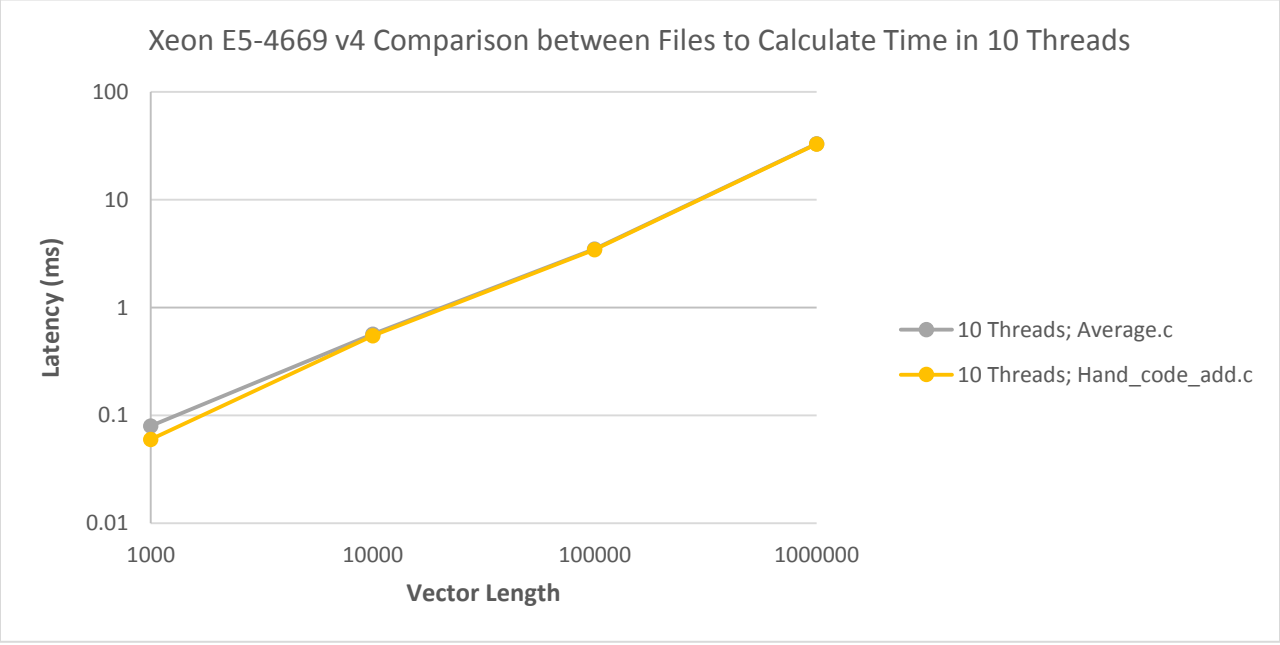
[48] P. Pacheco. (2010) *Shared Memory Programming with Pthreads* [Online] Available: https://people.cs.pitt.edu/~melhem/courses/xx45p/pthread.pdf

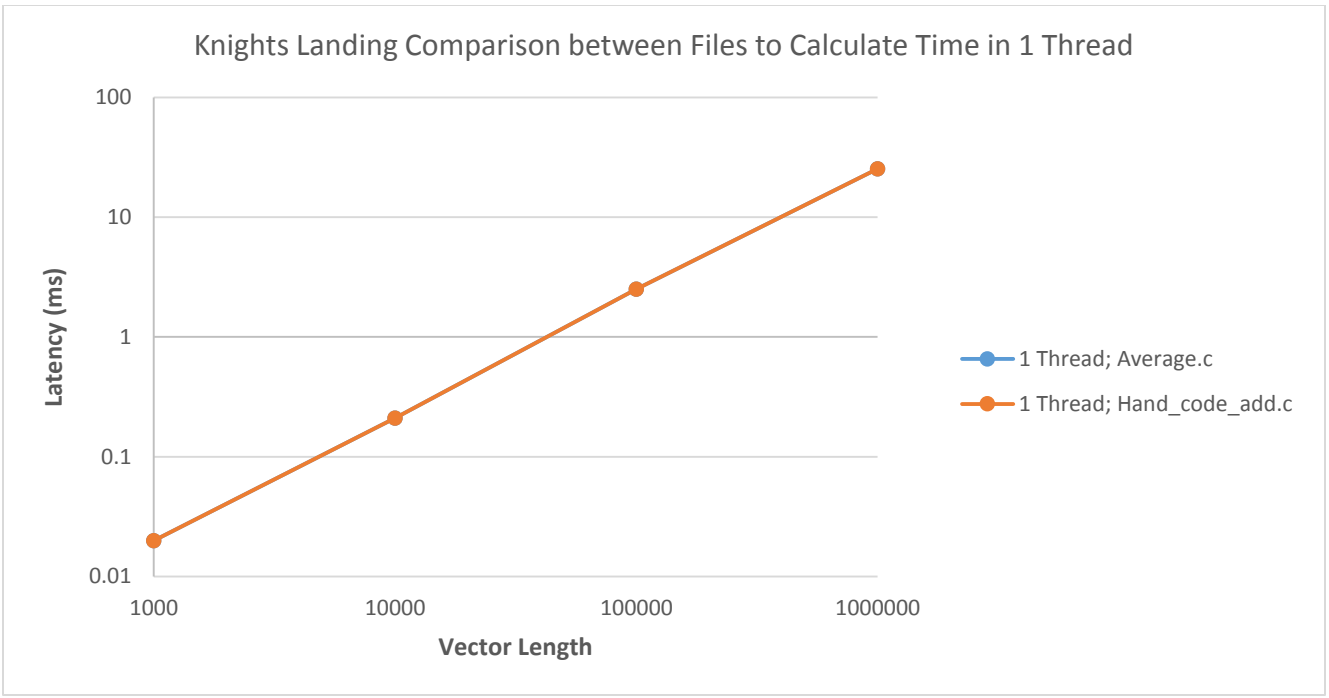
[49] J. Jeffers and J. Reinders, Intel Xeon Phi Coprocessor High Performance Programming, Waltham, MA: Elsevier Inc., 2013.

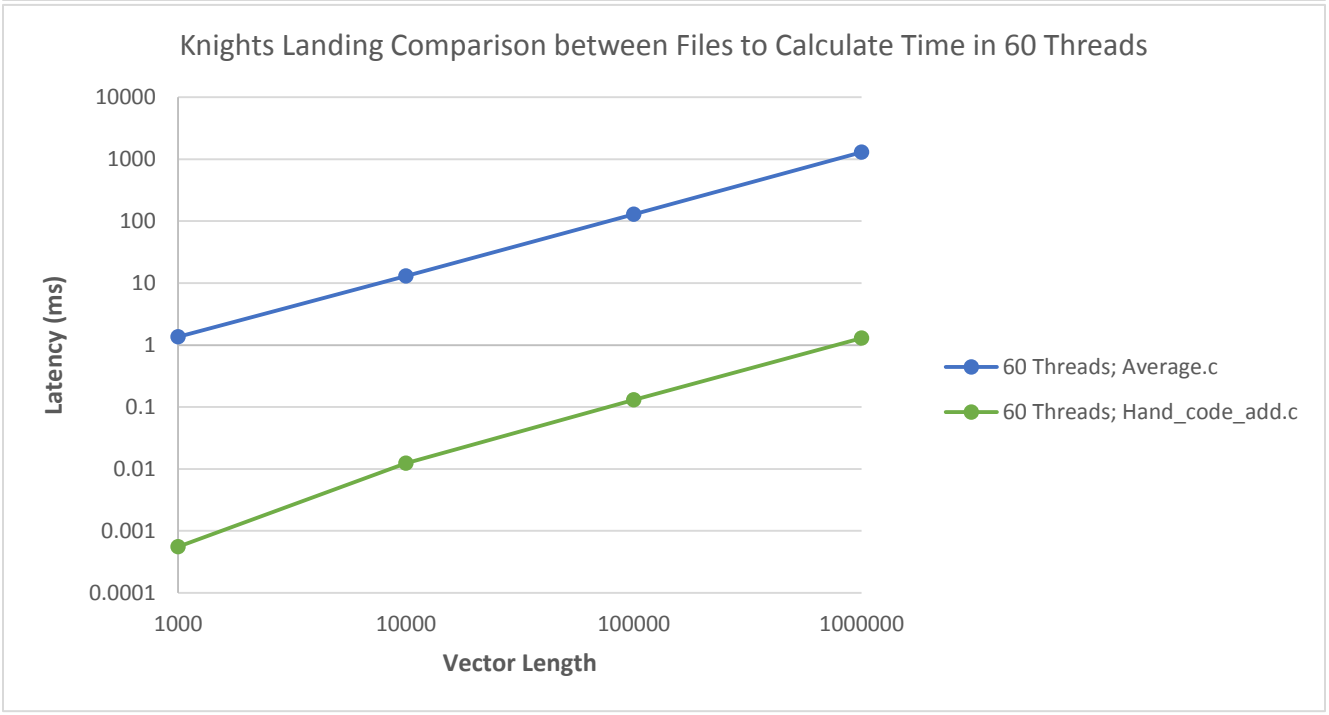
Appendix A: Graphs of including and excluding thread create and termination for the Knights Landing and the Xeon E5-4669 v4

All of the graphs either show that excluding the thread create and termination of threads bettered the latency or made no change to the latency. In all graphs if the time was calculated in average.c the create and termination of threads was included in the latency and in the hand_code_add.c did not include the create and termination of threads in the latency.



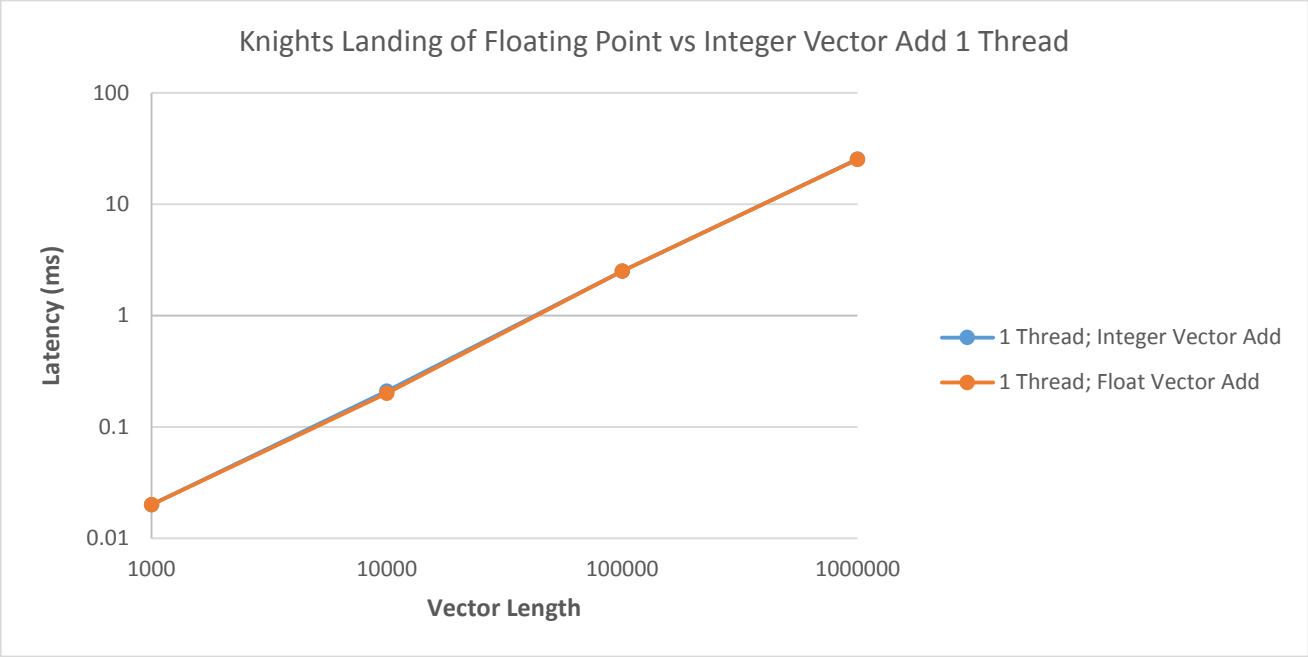


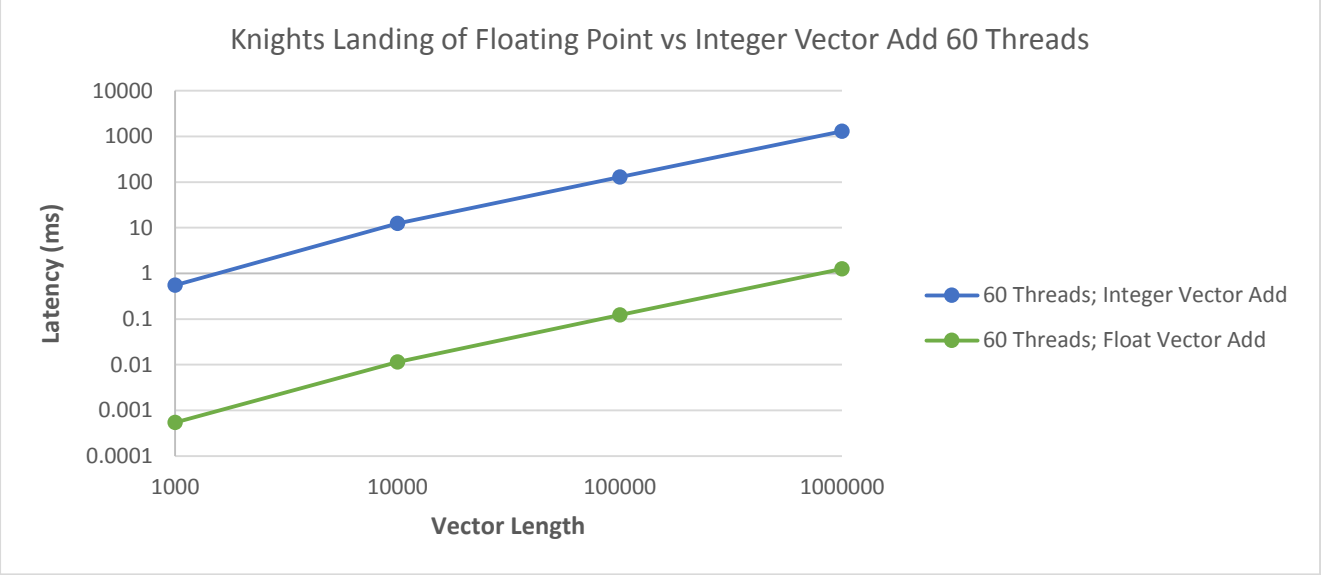

Knights Landing Comparison between Files to Calculate Time in 1 Thread
100
10
1
0.1
0.01
Latency (ms)
1000
10000
100000
1000000
Vector Length
1 Thread; Average.c
1 Thread; Hand_code_add.c

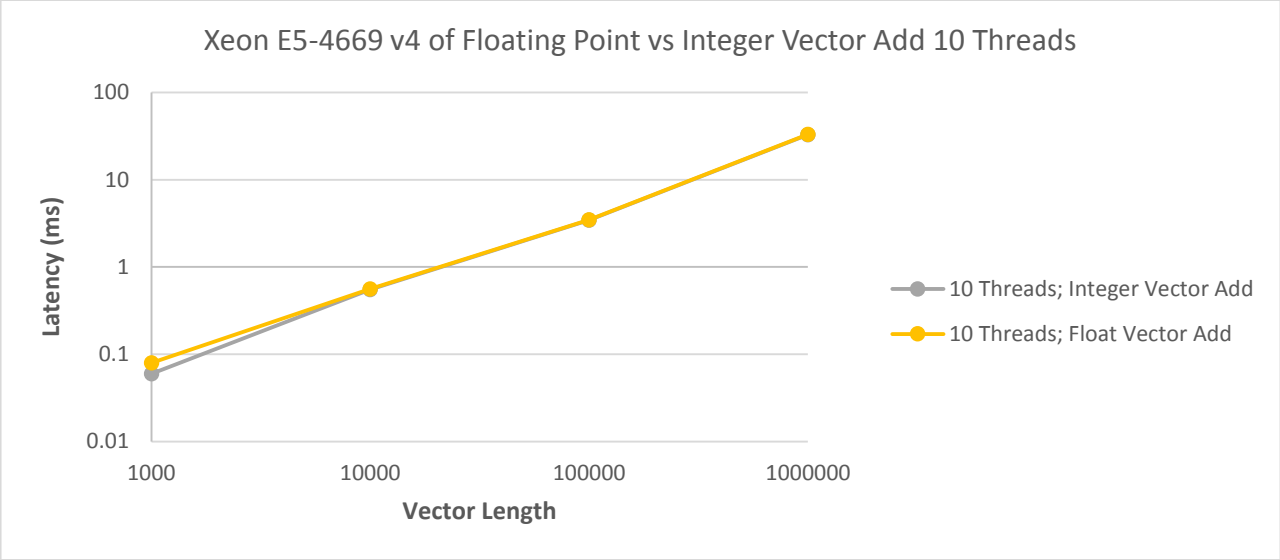

Knights Landing Comparison between Files to Calculate Time in 60 Threads
10000
1000
100
10
1
0.1
0.01
0.001
0.0001
Latency (ms)
1000
10000
100000
1000000
Vector Length
60 Threads; Average.c
60 Threads; Hand_code_add.c

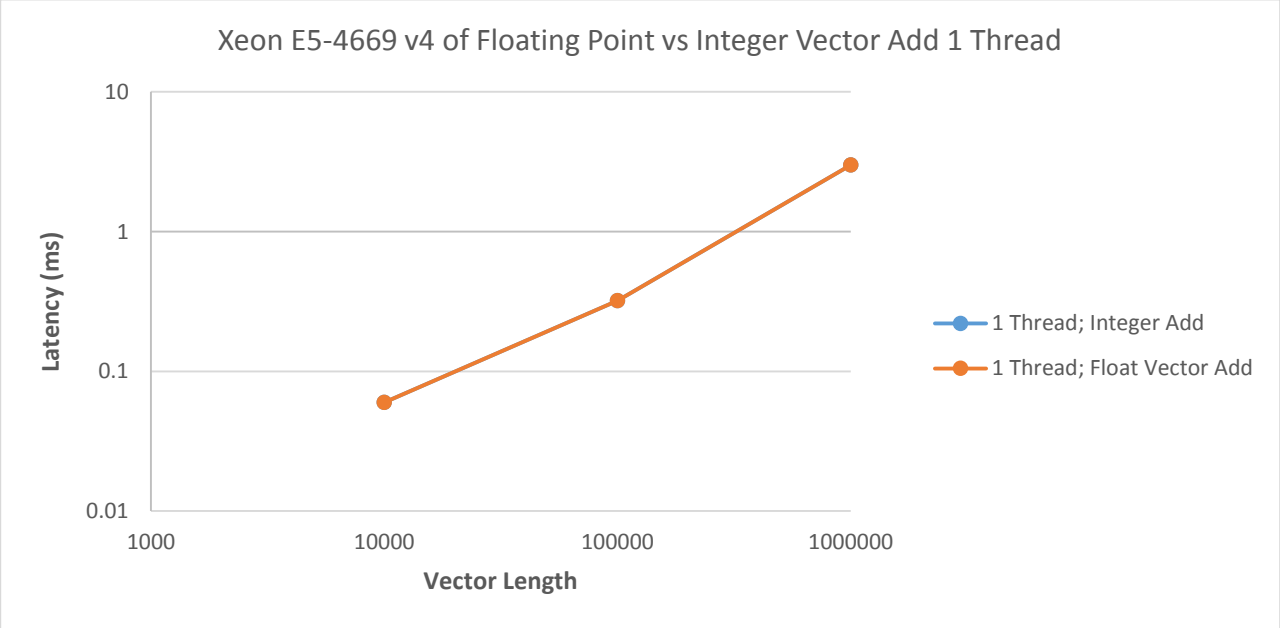
Appendix B: Graphs comparing floating point vector add and integer vector add Knights Landing and the Xeon E5-4669 v4

Almost all of the Knights Landing scenarios improved the latency greatly especially on 60 threadsThis improved latency is expected as the ISA of the Knights Landing is meant to handle floating point numbers. On the contrary, the Xeon E5-4669 v4 obtained slower latency in almost all scenarios when using the floating point vector add.






Xeon E5-4669 v4 of Floating Point vs Integer Vector Add 10 Threads
100
10
1
0.1
0.01
Latency (ms)
1000
10000
100000
1000000
Vector Length
10 Threads; Integer Vector Add
10 Threads; Float Vector Add


Xeon E5-4669 v4 of Floating Point vs Integer Vector Add 1 Thread
10
1
0.1
0.01
Latency (ms)
1000
10000
100000
1000000
Vector Length
1 Thread; Integer Add
1 Thread; Float Vector Add

Appendix C: Signal Processing Comparison Tool Code

The main function asked and scanned for the user inputs.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "average.h"
//#include <sys/prctl.h>

int VEC_NUM = 4;
int THREAD_NUM = 3;

void average_func(int threads[THREAD_NUM], int num_vecLength[VEC_NUM],
int iteration, int op);
int main()
{
	int threads[THREAD_NUM];
	int vec_Length[VEC_NUM];
	int iteration,op;
	printf("Comparison tool\n");
	printf("For each prompt press enter after you type it. If you are
asked for more than one value for a prompt press enter after each
value.\n");
	printf("Please enter 3 thread values: \n");
	scanf("%i %i %i",&threads[0], &threads[1], &threads[2]);
	printf("Please enter 4 vector length values: \n");
	scanf("%i %i %i %i",&vec_Length[0], &vec_Length[1],
&vec_Length[2], &vec_Length[3]);
	printf("Please enter how many iterations you would like to run:
\n");
	scanf("%i", &iteration);
	printf("Please enter the operation number you would like to run:
\n");
	scanf("%i", &op);
	average_func(threads, vec_Length, iteration, op);
	return(0);
}
```

The average function handled the user inputs and determined which vector function should be called.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#include "hand_code_add.h"
#include "average.h"
//#include "mkl_control.h"
```

```
//This c file takes the number of iterations the user wants to run for
each thread and sample count then runs the operation desired for that
amount of time.
//The file will produce a processing curve graph of the given
information

void create_threads(int num_threads, int vec_length, int iteration);

void average_func(int threads[THREAD_NUM], int num_vecLength[VEC_NUM],
int iteration, int op)
{
      FILE *fp;
      int i,j,m, count;
      count = 0;//initialize counter
      int max_count = VEC_NUM * THREAD_NUM;
      double time_test;
      double avgtime[max_count];
      double stdDEV[max_count];
      for(i =0; i<THREAD_NUM; i++){
            for(j=0; j<VEC_NUM; j++){
                  if (op == 1 | op == 2 |op==3){
                        create_threads(threads[i], num_vecLength[j],
iteration,op); //pass values to create threads
                        time_test = global_time;
                  //    printf("time test %.10lf\n", time_test);
                        avgtime[count] = global_time;
                        stdDEV[count] = global_stdDev;
                  }else if (op ==4)
                        printf("Single MKL thread mode");
                  else{
                        printf("The operation number you chose does not
have a correlate with a operation in the comparison tool\n");
                        printf("The operation numbers and operations are
as follows: \n");
                        printf("Operation: hand coded vector add, Op #:
1\n");
                        printf("Operation: hand coded scalar-vector
multiply, Op #: 2\n");
                        printf("Operation: hand coded vector add and
scalar-vector multiply, Op #: 3\n");
                        printf("Operation: MKL single thread, Op #:
4\n");
                        printf("To run an operation rerun the tool and
use one of the operation numbers provided. \n");
                        break;//exit loop
                  }
                  count++; //increments the number of times called;
should equal THREAD_NUM*VEC_NUM
            }
      }
      count = 0; //reset count for putting the values into the text
file
```

```
      char filename[32];
      snprintf(filename,sizeof(filename)*32,"XeonPhi_op%i.txt",op);//
change title when switches between servers
      fp = fopen(filename, "w+");
      fprintf(fp, "Number of Threads  Vector Length  Operation #
Number of Iterations  Avg Latency  Standard Deviation\n");
      for(i =0; i<THREAD_NUM; i++){
            for(j=0; j<VEC_NUM; j++){
                  fprintf(fp, "%9i %17i %14i %20i %22.12lf %20.12lf\n",
threads[i], num_vecLength[j],op, iteration,
avgtime[count],stdDEV[count]);
                  count++;//increment counter
            }
      }
fclose(fp);
return;
}
```

The average.cpp header file.

```
//Header file for the average data
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#include "hand_code_add.h"
//#include "mkl_control.h"
extern int THREAD_NUM;
extern int VEC_NUM;

void average_func(int threads[THREAD_NUM], int num_vecLength[VEC_NUM],
int iteration, int op);
```

The hand_coode_add.cpp file which contains the thread create, thread terminate, the vector add function, and the function to compute the averages.

```
# define _POSIX_C_SOURCE 199309L
# include <stdio.h>
# include <stdlib.h>
# include <sched.h>
# include <assert.h>
# include <pthread.h>
# include <string.h>
# include <unistd.h>
# include <stdint.h>
# include <time.h>
# include <math.h>
```

```
# include <errno.h>
//# include <aligned_new>
# include "tbb/tbb_allocator.h"
# include "hand_code_add.h"

# define BILLION 1E9
# define handle_error_en(en, msg)\
      do {errno = en; perror(msg);exit(EXIT_FAILURE);}while(0)
# define handle_error(msg)\
      do {perror(msg);exit(EXIT_FAILURE);}while(0)

//global variables for this source file
int vector_length; //length of vectors
int iterations; //how many iterations each function should be called
int local_num_threads; // number of threads to be created for each
call
pthread_barrier_t threadWait;//block to wait for all threads to be
created
double *arrayTimes;
double *arrayStdDev;
double *arrayOfIDs;
//global variable to be used in multiple source files
double global_time;
double global_stdDev;

//instantiate functions
void stand_dev_avg(double *time_array, int ID);
void* multiply_code(void *arg);
void* multiplyAdd_code(void *arg);
void *waitThreads(void *arg);
void *add_code(void *arguments);
double timeConvert(struct timespec time);

struct Myargs{
      float * restrict a;
      float * restrict b;
      float * restrict c;
      double *all_times;
      int ID;
};


// function that creates the number of threads the user asks and
computes the vector add, vector multiply, or vector multiply and aall:
main


// function for the number of samples and iterations given by the user
void  create_threads(int num_threads, int vec_length, int iteration,
int op){
      printf("Threads %i Vec Length %i\n", num_threads, vec_length);
      //declare variables
```

```
      int t, i, ret;
      Myargs *args;
      struct sched_param* param;
      cpu_set_t cpusetp;


      // set main thread to one cpu
      pthread_t current_thread = pthread_self();
      CPU_ZERO(&cpusetp);//clear the previous sets
      CPU_SET(num_threads, &cpusetp);//set main thread to cpu ID
greater than number of threads
      pthread_setaffinity_np(current_thread, sizeof(cpusetp),
&cpusetp);

      //thread values
      pthread_t *threads; //thread array
      pthread_attr_t tattr; //thread attributes
      pthread_barrierattr_t battr;

      //variables for average time and standard deviation
      double sumMultipleThreads = 0.0; //sum of mutliple sum times
      double avgMultipleThreads = 0.0; //average of multiple sum times
      double sumMultipleStdDev = 0.0; //sum of multiple standard
deviation
      double avgMultipleStdDev = 0.0; //average of multiple standard
devaitions

      //allocate memory for array of average time and standard
deviation
      arrayTimes = (double *)malloc(sizeof(double)*num_threads);
      arrayOfIDs = (double *)malloc(sizeof(double)*num_threads);
      arrayStdDev = (double
*)malloc(sizeof(double)*num_threads);//NOTE****change to num threads
if global is used
      threads = (pthread_t *)malloc(sizeof(pthread_t)*num_threads);


      //set globals
      vector_length = vec_length;
      iterations = iteration;
      local_num_threads =num_threads;

      //set up for threads and barrier
      pthread_attr_init(&tattr);// initialize attributes
      pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);//set scope
of bound thread

      ret = pthread_barrierattr_init(&battr);
      if (ret != 0){
            handle_error_en(ret, "pthread_barrierattr_init");
      }
```

```
      ret = pthread_barrierattr_setpshared(&battr,
PTHREAD_PROCESS_PRIVATE); //set so it can only be operated upon in the
main thread
      if (ret != 0){
            handle_error_en(ret, "pthread_barrierattr_setpshared");
      }
      ret =
pthread_barrier_init(&threadWait,&battr,num_threads+1);//wait fot all
threads to be created
      if (ret != 0){
            handle_error_en(ret, "pthread_barrier_int");
      }

      //shortened length
      int len_vec = 256;

      //create theads for appropriate operation
      if (op ==1){
      //vector add operation

            for(t=0; t<num_threads; t++){
                  //allocate memory for structure
                  args = (Myargs *)malloc(sizeof(Myargs));
                  param = (sched_param *)malloc(sizeof(sched_param));
                  (*args).a = (float *)_mm_malloc(sizeof(float)*len_vec,
64);
                  (*args).b = (float *)_mm_malloc(sizeof(float)*len_vec,
64);
                  (*args).c = (float *)_mm_malloc(sizeof(float)*len_vec,
64);
                  (*args).all_times = (double
*)malloc(sizeof(double)*iterations);
                  (*args).ID = t; // set ID for each thread
                  (*param).sched_priority = t;// sched_priority will be
the priority of the threads

                  pthread_create(&threads[t], &tattr, &add_code, (void
*)args);//vector add func}

            }
      }else if (op ==2){
      //vector multiply operation
      }else{
      //vector mulitply and add operation

      }

      ret = pthread_attr_destroy(&tattr);// destroy attributes
      if (ret != 0){
            handle_error_en(ret, "pthread_attr_destroy");
      }
```

```
	//join threads
	pthread_barrier_wait(&threadWait);
	for(t=0; t<num_threads; t++){
		ret = pthread_join(threads[t],NULL);//defined attributes
		if (ret != 0){
			handle_error_en(ret, "pthread_join");
		}
	}

	//loop through array of sum of times and standard deviations
//	#pragma loop count min(256)
	for(t=0;t<num_threads;t++){
		sumMultipleThreads = sumMultipleThreads + arrayTimes[t];
		sumMultipleStdDev = sumMultipleStdDev + arrayStdDev[t];
	}
	//take average of times across all threads
	avgMultipleThreads = sumMultipleThreads / (num_threads *
iterations);
	avgMultipleStdDev = sumMultipleStdDev / num_threads;
	global_time = avgMultipleThreads;
	global_stdDev = avgMultipleStdDev;

	//clear barrier
	pthread_barrier_destroy(&threadWait);
	free(arrayTimes);
	free(arrayOfIDs);
	free(arrayStdDev);
	free(threads);
}

//vector add function
//int add_code(){
void *add_code(void* arguments){
	int LEN =256;
	struct Myargs *add_args = (struct Myargs*)arguments;
	int threadID = add_args->ID;//thread ID number and index number
	struct sched_param param;
	//variables for vector add
	float *a = add_args->a;
	float *b = add_args->b;
	float *c = add_args->c;
	double *all_times = add_args->all_times;
	int i, t,s,k, vec_length;
	struct timespec start, end;
	double diff;
	double startTime = 0.0;
	double endTime = 0.0;
	//cpu values
	int num_cpus = local_num_threads;
	cpu_set_t cpusetp;
	int idx[LEN];
```

```
	param.sched_priority = threadID;// sched_priority will be the
priority of the thread
	pthread_t current_thread = pthread_self();

	//set values for thread
	CPU_ZERO(&cpusetp);//clear the previous sets
	CPU_SET(threadID, &cpusetp);//add cpus to the set;
	pthread_setaffinity_np(current_thread, sizeof(cpusetp),
&cpusetp);
	s =pthread_getaffinity_np(current_thread, sizeof(cpusetp),
&cpusetp);

	//set up vector length divide vector length given by the user by
LEN to make it run one cycycle per iteration to remove bottlenecking
	vec_length = vector_length/LEN;


	//wait for all threads to be created
	pthread_barrier_wait(&threadWait);
	for( t=0; t<iterations; t++){
		clock_gettime(CLOCK_REALTIME,&start);
		for(i=0; i< vec_length; i++){
			#pragma prefetch
			#pragma ivdep
			#pragma vector aligned
			//#pragma loop count min(255)
			for(k=0; k<LEN ;k++){
				c[k] = a[k] +b[k];
			}
		}

		clock_gettime(CLOCK_REALTIME,&end);
		//covert times and take difference
		startTime = timeConvert(start);
		endTime = timeConvert(end);
		diff = endTime - startTime;
		all_times[t]= diff;
	}

	stand_dev_avg(all_times, threadID);//send array of times to
average and standard deviation function
	//release vectors
	_mm_free(add_args->c);
	_mm_free(add_args->a);
	_mm_free(add_args->b);
	free(add_args->all_times);
	free(add_args);
	pthread_exit(0);
	return(0);
}
```

```
//multiply vector function
void* multiply_code(void *arg){
	return(0);
}

//vector multiply and add function
void* multiplyAdd_code(void *arg){
	return(0);

}

//this function determines the standard deviation and average of the
time
void stand_dev_avg(double time_array[], int ID){
	double timeSum = 0.0; //sum of all time for one thread
	double avg_time = 0.0 ; // average of all time for one thread
	double stdDev = 0.0; //standard deviation for one thread
	int i;

	//sum of all times of one thread
	//#pragma loop count min(512)
	for (i = 0;i<iterations;i++){
		timeSum = timeSum + time_array[i];
	}
	avg_time = timeSum / iterations;

	//standard deviation of all times of one thread
	//#pragma loop count min(512)
	for(i = 0 ;i <iterations; i++){
		stdDev = stdDev + pow(time_array[i] - avg_time,2);
	}


	arrayTimes[ID] = timeSum;
	arrayStdDev[ID]= stdDev;

	return;
}

double timeConvert(struct timespec time){
	double magnitude = (time.tv_sec + (time.tv_nsec * 1E-9));
	if (magnitude < 0){
		printf("less than zero\n");
	}
	return(magnitude);
}
```

The hand_code_add.cpp header file.

```
# include <stdio.h>
# include <stdlib.h>
# include <sched.h>
```

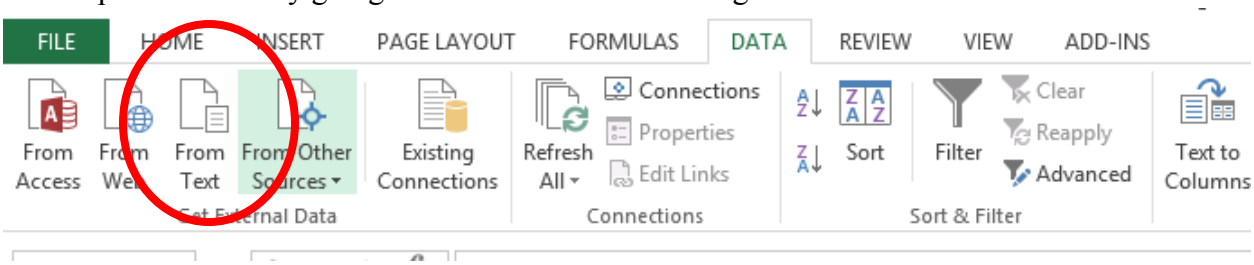
```
# include <assert.h>
# include <pthread.h>
# include <string.h>
# include <unistd.h>
# include <stdint.h>
# include <time.h>
# include <math.h>
# include <errno.h>

# include "tbb/tbb_allocator.h"
//header file for the function that creates threads, a function for
the vector add, as well as a function that finds the standard
deviation and average of the latency
extern double global_time;
extern double global_stdDev;
void create_threads(int num_threads, int vec_length, int iteration,
int op);
```

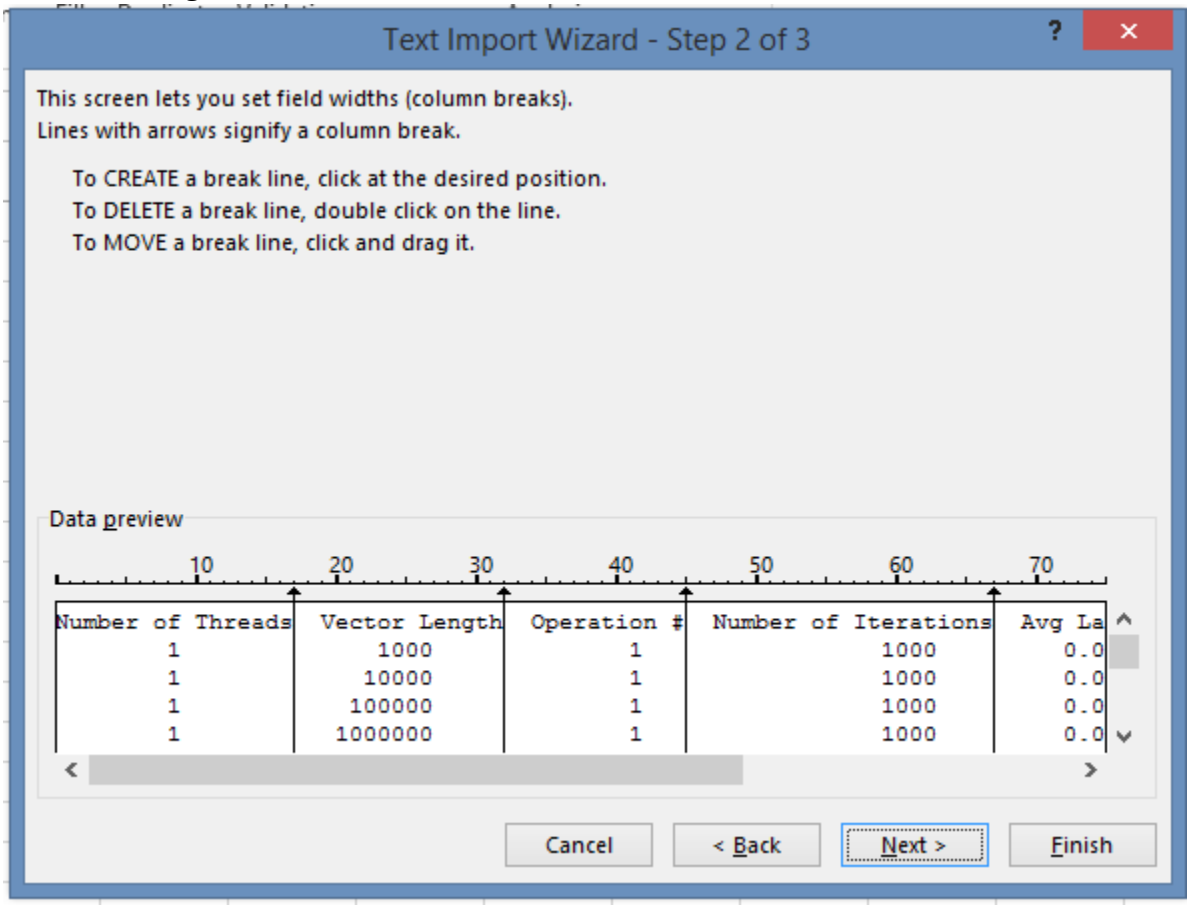
Appendix D: How to run the signal processing comparison tool

How to run the Signal Processing Tool

1. In the Linux command window type: ‘Make clean’ then ‘Make’ then ‘./main’
2. Enter responses to each of the questions prompted by the program. After each entry press enter to submit the answer to the program.
    a. Enter three thread values
    b. Enter four vector lengths
    c. How many iterations would you like the function to run on?
    d. What operation number do you want to run?
3. Text file will be created, save it, and import it to Excel
4. Import to Excel by going to the data tab and selecting “From Text”



5. Follow the pop up window. One step two of the import you may have to move the cursor for column alignment



6. Then your data will be imported. Copy the table from the chartTemplate sheet with the GFLOP/s calculation
7. Copy the average latency from the imported text file to the Latency (ns) column. This will give you the GFLOP/s per core.

Important to know

- The only working function is the vector add function, which is operation number 1
- All files saved as XeonPhi_op1.txt (the number of the file will change depending on the file called)
- The makefile currently has the MIC-AVX512. There are two saved makefiles in the folder. One called makefileMIC with the MIC-AVX512 and one makefileCORE with the CORE-AVX2.
- Build on Phi and copy the executable to Xeon
    - Phi MIC-AVX512
    - Xeon CORE-AVX2

Appendix E: Makefile with CORE-AVX2 compiler options

```
all: main

main:  hand_code_add.o average.o main.o
     icpc hand_code_add.o average.o main.o -o main -O3 -xAVX -axCORE-
AVX2 -ansi-alias -lpthread

main.o: main.cpp average.h
     icpc -c -O3  -xAVX -axCORE-AVX2 -ansi-alias -qopt-report=3 -
std=c++17 main.cpp


hand_code_add.o: hand_code_add.cpp hand_code_add.h
     icpc -c -O3  -xAVX -axCORE-AVX2  -ansi-alias  -restrict -lpthread
-parallel -qopt-prefetch -qopt-report=3 -std=c++17 hand_code_add.cpp


average.o: average.cpp hand_code_add.h average.h
     icpc -c -O3  -xAVX -axCORE-AVX2 -ansi-alias -qopt-report=3 -
std=c++17 average.cpp

clean:
     rm -rf *o main
```

Appendix F: Makefile with MIC-AVX512 compiler options

```
all: main

main:  hand_code_add.o average.o main.o
     icpc -O3 -xMIC-AVX512 -axMIC-AVX512  -ansi-alias hand_code_add.o
average.o main.o -o main -lpthread -lm

main.o: main.cpp average.h
     icpc -c -O3  -xMIC-AVX512 -axMIC-AVX512  -ansi-alias -qopt-
report=3 -std=c++17 main.cpp


hand_code_add.o: hand_code_add.cpp hand_code_add.h
     icpc -c -O3  -xMIC-AVX512 -axMIC-AVX512   -ansi-alias  -restrict
-L -lpthread -parallel -qopt-prefetch -qopt-report=3 -std=c++17
hand_code_add.cpp


average.o: average.cpp hand_code_add.h average.h
     icpc -c -O3  -xMIC-AVX512 -axMIC-AVX512 -ansi-alias -qopt-
report=3 -std=c++17 average.cpp

clean:
     rm -rf *o main
```