**Architecture Analysis Using Optimized Implementations of Dot Product, Sobel Filter and Quick Sort kernels on Multi and Many-Core Architectures**

# Abstract

In recent years, High Performance Computing (HPC) has been used on wide variety of platforms ranging from embedded processors to massively parallel GPUs. It’s become highly critical for designers and developers to squeeze out every ounce of performance from their devices and at the same time decide which devices/architectures are suitable for what kind of applications. The focus of this paper is to underscore the strengths and shortcomings of three major device architectures namely Many Core CPUs, Multi-Core CPUs and GPUs using benchmarking. Different kernels having characteristics of being data-bound, compute-bound, etc., using their software optimized implementations, have been chosen for a fair assessment of architectures. Comparisons in terms of efficiency, power usage and speedups have been carried out using reliable performance metrics like Computational Density (CD) and Realizable Utilization (RU) which should provide designers useful info while choosing architectures for similar applications.

Index Terms: Computational Density (CD), Realizable Utilization (RU), Giga Operations per second (GOPS), Compute Unified Device Architecture (CUDA), ATLAS (Automatically Tuned Linear Algebra Software (ATLAS), Nvidia CUDA Basic Linear Algebra Subroutines (cuBLAS) library.

# Introduction

Designers always face a dilemma in choosing device architecture while developing a system for a single or given set of applications since all architectures are diverse in their performance characteristics. Also there is no dominant programming language or communication abstraction model which works equally well on all architectures and thus a parallel program implementation of an application that is best suited to one machine may not be appropriate for another.

Benchmarking is a technique used to evaluate the performance of an application on different system architectures and extrapolate from the results obtained. It helps developers with accurate data to compare architectures and choose the best platform among them during the device design stage for a given application with its own sets of characteristics. It also serves as a diagnostic tool to discover the cause of poor performance of a configuration and infer about what kinds of applications would work better or worse on such configuration.

Another advantage of benchmarking are usage of different performance metrics like Realizable Utilization that are used to measure efficiency of each architecture which would aid developers to compare applications that are undergoing optimizations to similar applications which have already been implemented. This info helps during optimization phase of development cycle which would aid developers to decide if further optimization is worth the time and cost. Finally apart from speedup and efficiency, other factors like power usage, number of resources used, communication overhead and overall cost can be obtained from this technique and can be considered while selecting architecture if the primary goal is to strike an optimum balance between speed and cost.

This study carries out architecture analysis of Intel Xeon E5-2670, Intel Xeon Phi 5110P and Nvidia Tesla K20X using optimized implementations of dot product, quick sort, sobel filter and linear spacing kernels. Kernels are well-defined parts of real applications like matrix factorization, FFTs etc. These kernels have been optimized using libraries like Armadillo, ATLAS, cuBLAS and parallel programming techniques to get best results on each of the architectures. Results of the analysis show, firstly, which kernel is suitable on which architecture at different data sizes and secondly a comparison of efficiency and power usage for each kernel on all architectures.

Study has been broken up into several sections. Presented first is motivation behind choosing the kernels and background of each of the kernels and the devices used in this study. It is followed by the section on implementation details like experimental setup, optimization details, results and analysis of each kernel on all architectures. The third section presents a thorough performance comparison analysis of each kernel on all architectures. Finally we make concluding remarks of our study.

# Motivation

The past decade has witnessed the advent of multi-core and many-core processors for desktops, laptops and even smart phones. All these parallel architectures are unique in their performance characteristics and programming models. To analyze the performance of three distinct parallel architectures, multi-core CPUs, many-core CPUs and GPUs, we have chosen 4 kernels namely dot product, linear spacing, sobel filter and quick sort. The chosen kernels are diverse in nature and find applications in various fields like image processing, remote sensing, database management, physics, astronomy etc. The nature of the kernels chosen are such that they are a mix of memory bound, bandwidth bound and computation intensive applications. These kernels are likely to be part of larger computations and it would be interesting to understand and analyze their behavior on various parallel architectures through benchmarking.

Moreover, benchmarking provides developers with accurate data to compare architectures and choose the best platform among the device design stage. This gives a fair idea of the device’s performance beforehand and will help in manufacturing faster processors. In addition to pairing applications with their most compatible architectures, benchmarking also helps in extrapolating the performance of kernels on various architectures which paves way for the evolution of parallel architectures.

# Background

## Kernel Description and Related Research

In the section below we discuss the major advancements and research that have been made in the field of parallel computing in each of kernels.

### Sobel Filter

Image processing domain presents ample amount of opportunities for parallelism. Some of the cases are embarrassingly parallel in terms of data and task flow. Sobel filter is one of the kernels that finds wide variety of application where edge detection is required in image processing. Different fields such as medical [18] or satellite images [19] processing where feature extraction is desired. Edge detection is an important step as it helps in identifying objects and minor details in the images.

The sobel filter performs an averaging step to reduce noise, followed by a differentiation step to extract the edges. The scale of the sobel mask can vary depending on the application requirement. However if the mask size is increased to bigger sizes smaller details might diminish during averaging. The averaging is required to smoothen the noise from the image. The convolution masks or kernels can be implemented as linear spatial filters. The convolution kernel for sobel [20] comprises of two parts, one in the directions of rows(x) and the other in column(y). These kernels are slid over the entire image and a new image with prominent edges is obtained. The two convolution kernels in x (row) and y (column) directions can be given as,

The new pixel is therefore the magnitude of the gradient calculated in x and y direction.

The orientation of the edge can be found by, . This is followed by thresholding the values which extend beyond the pixel value range.

The complexity of this algorithm can be evaluated as O(3\*3\*2\*N\*N + 2\*N2+ N2) i.e. O(N2). This is independent of the size of the kernel selected. One may also use the kernels of size 5x5 if the image needs high amount of noise elimination. It should be noted that the averaging step is similar to a basic Gaussian filter in characteristic.

Khalid et al.[10] studied the convolution kernel for multicore devices. They explored the performance in terms of CPU utilization by varying the image size, number of threads spawned and processor type. It was concluded that CPU utilization was highest when 8 threads were spawned for Intel Xeon E5420, quad core and Core 2 duo processors. A study by Alex et al.[11] analyzes how data distribution in MPI programming model can affect performance. If only the master does all the data distribution, the worker processors are idle for a long time. It showed an improved performance when the task of data distribution was divided among processors.

**Parallelism:** Sobel offers a high amount of parallelism. Intuitively we can see that there is no data dependence while convolving the image to find the x and y gradients. Hence these two steps can be done simultaneously, followed by the gradient calculation per pixel. So every time the processor finishes these steps we have a new pixel for the output image. This parallelism can further be augmented by considering the number of ALUs present in core, where they will be able to do multiple operations at once. These divisions can be further improved by using optimized libraries for particular applications.

It is critical to achieve appropriate load balancing in a distributed memory access. Also in the case of shared memory it is important to keep the number of remote accesses as low as possible. The various methods to be explored here are load balancing and to reduce the communication overhead. Since, the convolution masks slid over the entire image there is significant dependence on the neighboring pixels. This dependence can lead to overheads and hence an appropriate distribution has to be worked out.

### Dot Product

Dot Product is an algebraic/geometric operation that takes two finite length sequence numbers and returns a single number. Algebraically, Dot Product is sum of the products of the corresponding entries of the two sequences of numbers. It has real-world applications ranging from physics, astronomy, image processing, mechanics etc.

**Algebraic Formula**

[1]

Where a and b are two vectors a = [a1, a2, …..an] and b = [b1, b2, …..bn]

**Time Complexity and Number of Operations:** The dot product obtained from the algebraic formula has ‘n’ multiplications and ‘n-1’ additions. Assuming that both multiplication and additions are constant-time operations, the time complexity of a Dot Product is O(n) [O(n) +O(n)].

**Parallelism:** From equation [1], each element of one vector (sequence) is multiplied with the corresponding element of 2nd vector and there is no data dependency between elements. Thus each processor would be given same or set of same index elements and their tasks would be finding the products of these elements. After finding the products, all these elements can be added by single processor or multiple processors and final output can be obtained. Thus, dot product is embarrassingly parallel since it requires little effort to separate the problem into number of parallel tasks. Also there is no data dependency between these parallel tasks and so communication of results between tasks is not required. Only synchronization is to be provided (Case for Shared Memory Model) to make sure all processors are done with their product computations and next phase of summation of all products can be started.

Dot product as an operation is data-intensive since it accesses large amounts of data from cache/memory and performs relatively less computations. Also dot product implementation is bound by memory bandwidth for large vectors since it reads each value only once and so there is no scope of utilizing cache’s advantage of data reuse. Results of dot product implementation in subsequent sections would reflect this characteristic.

In [8], Dot product is implemented on CPU, GPU and FPGA using BLAS (Basic Linear Algebra Subroutine) library and comparisons are made regarding the performances of the kernel on these device architectures. For implementing dot product on many core CPU and GPU, Intel Math Kernel Library (MKL) and CUDA BLAS were used respectively to device the BLAS kernels. Results showed that MKL-PAR (Parallel implementation using MKL) was 1.8 to 88.4 times faster than the CUDA implementation on the GPUs since CUDA BLAS implementation is dominated by the system and driver overheads associated with data transfers.

Another study [9] focuses on reducing the cost of modular reductions (related to last step of dot product where summation of all products take place), which happens to be slow operation of the process. Two different algorithms have been devised to compute dot product in a finite field using floating-point arithmetic and parallel implementations have been carried out on GPUs. First algorithm does the summation part by lumping integers into packets and doing the final operation. Second algorithm enables summation to be done without any modular reductions and consequently this particular way of summation results in better performances over the latter. Final results show a speedup of more than 40 with the 2nd algorithm and about 10 with the first algorithm on GPUs.

### Quick Sort

The sort kernel is a core part of many computing applications. Earlier computer manufacturers considered that in most of the computing systems more than 25% of the computation time was spent on sorting. Sorting finds its usage in all spreadsheet programs, database applications and numerous other applications. Thus a lot of effort was put in to parallelize the sorting techniques. Considerable progress has been made in designing algorithms with excellent efficiency but not much of success has been achieved in implementing these algorithms on parallel machines due to the amount of data dependence in these algorithms.

Sorting is defined as the operation of arranging an unordered collection of elements into an order (either increasing or decreasing) collection. The sorting algorithms can be categorized into two major groups.1.Comparision based sorting algorithm order an ordered list of elements by comparing a pair of elements and exchanging them if they are out of order. The lower bound on the time complexity for such algorithms is O(n log n). Eg: quick sort, merge sort etc. 2. Non comparison based algorithms sort by using certain properties of the elements (like their binary representation).Eg: radix sort, bucket sort.

Quick sort is a divide and conquer, comparison sorting algorithm that on an average makes O(n log n) comparisons to sort n elements. In the worst case it makes O(n2) comparisons but it is almost always possible to avoid the worst case scenario by properly choosing the pivot. The performance of the quick sort algorithm depends on how it divides its list. If a sequence of size is split poorly into sub sequences 1 and k-1 then the time complexity is given by O(n2) whereas if it is split into equal halves of[n/2] and[n/2] elements then the complexity is O(n log n). Additionally it is has O(n log n) space complexity which means that it uses only O(n log n) additional spaces to sort the entire array.

Algorithm (Sequential Quick sort)

1. Choose an element from the array/list. The chosen element is called the pivot.
2. Partition the list into two sub lists/halves. The upper half consists of elements from the list that are greater than the pivot and they are placed after the pivot in the array. The other half consists of elements lesser than the pivot and they are placed before the pivot in the array. After the partition the pivot is in its final position.
3. Recursively apply the above steps to the two sub lists.

The recursion should be applied until the sub list size is 0 or 1.

Illustration ofQuicksort is shown in Figure 1*.*

Advantages of Quick sort

1. It is one of the fastest sorting algorithms.

2. It does in place sorting and hence doesn’t need any additional memory unlike other sorting algorithms like heap sort.

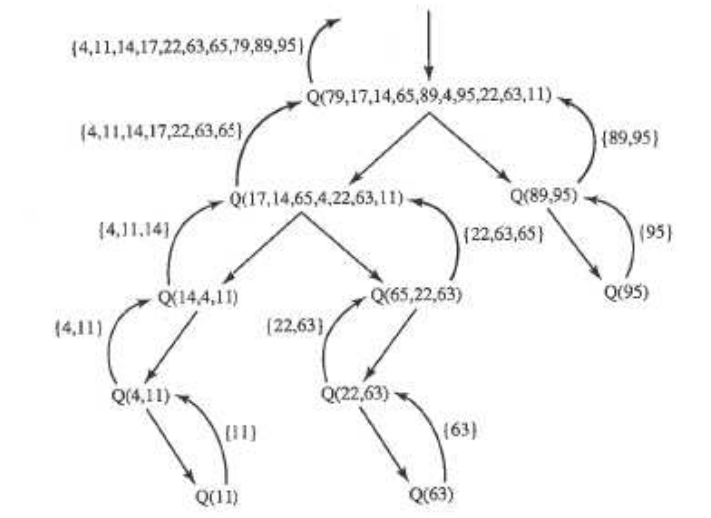


Figure 1: Illustration of Quick Sort [25]

#### Parallel Quick Sort Algorithm

The fundamental problem that one faces while designing parallel sorting algorithms is to collectively process and sort data owned by all the individual processors. An efficient parallel algorithm does so by utilising all the processors to sort the elements while minimising the communication overhead between the processors.

One of the possible methods to parallelize quick sort would be to divide the given array into two parts (sub arrays) and sort each part recursively. Since sorting the two sub arrays can be considered as two completely different problems they can be given to two different processors. Therefore one way is to execute quick sort on a single process and then when the algorithm performs recursive calls assign the sub processes to different processors. The algorithm terminates when the arrays cannot be partitioned further.

The major drawback of this method is that the partition of the array into smaller arrays is done by a single process. To obtain an efficient parallel quick sort we have to perform partitioning in parallel.

#### Parallel quick Sort algorithm for shared address space system

The algorithm starts by assigning each processor a consecutive block of N/P elements where N is the total number of elements in the array and P the total number of processors. The processors are labelled to define the global order of the sorted sequence. Let Ai be the block of elements given to process Pi. First we select a pivot element and broadcast it to all processors. Each processor then rearranges its assigned block of elements into two subsets one with elements smaller than and equal to the pivot placed in the first half of the array and other with elements greater than the pivot placed in the second half of the array. Each process in the upper half of the process list sends its “low list” to a partner process in the lower half of the process list and receives a “high list” in return. Thereafter the processes divide themselves into two groups and the algorithm recourses. After Log2P recursions every process has an unsorted list of values completely disjoint from the values held by the other processes. Each process can sort its list using sequential quick sort.

Illustration shown below in Figure 2(a) specifies that one processor broadcast initial pivot to all processors. Figure 2(b) shows that each processor in the upper half swaps with a partner in the lower half. Figure 2(c) specifies recursion on each half. 2(d) shows the swapping among partners in each half. In figure 2(e), each process uses quick sort to sort elements locally.

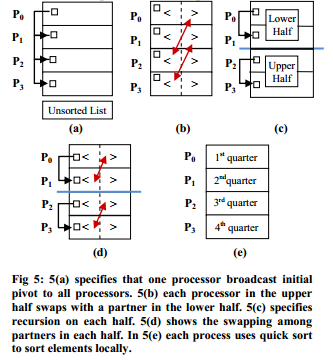


Figure 2: Parallel Quick Sort [12]

### Linear Spacing

Linear spacing generates linearly spaced vector between (and including) two values x1 and x2. To do this, we generate an interval length using the formula

where,

d is the interval length or the spacing between the vectors.

x1 and x2 are the starting and end points of the vector and the range in which the points in the vector will lie.

n is the number of points to be generated between x1 and x2. It is also referred to as data size.

We divide by (n-1) because there are (n-1) intervals in a sequence of n points. We then calculate each element in the vector using the formula

where,

vec[i] refers to the ith element of the vector where x1 < i < x2.

**Asymptotic Complexity**: Linear spacing has 2 subtractions and 1 division followed by (n-2) additions and multiplications. Therefore, the asymptotic complexity of the algorithm is O(N).

**Parallelism**: It can be clearly seen from the above formula that each element of the vector can be generated independently. Therefore, the kernel is embarrassingly parallel and does not have any inter-thread dependencies. It does not have any data sharing with the other threads if each thread is given the values of x1, x2 and n.

As a kernel linear spacing is computation intensive as it does more computations than memory accesses. There are 3+2\*(n-2) arithmetic operations and 3 memory access operation, one for x1, x2 and n. The data needed for computation will be contained within the processor’s cache and hence there would not be any overheads while fetching data.

The kernel finds application in exploring single dimensional spaces to find equally spaced points for sensor arrangements in remote sensing.

## The Various Devices

### CPU vs GPU

The CPU is the main hardware that controls a computer. It performs arithmetic and logical operations on the data and manages the input and output operations. They implement instruction level parallelism through pipelining. CPUs have a powerful Arithmetic and Logical Unit (ALU) that is designed to minimize operation latency. The ALU is supported by a cache that minimizes memory access latencies. They also have sophisticated control structures like branch prediction and data forwarding. These devices are focused on reducing the latency. The CPUs have evolved over time and all modern CPUs have multi-cores, i.e., they have more than one core on a single chip. They are also many-core chips that contain a large number of cores on one chip. The average CPU used in the laptops these days are multi-core. The many-core CPUs are used for highly computational applications.

A GPU is a co-processor or an accelerator that works in conjunction with one or more CPUs. They are not stand alone devices. GPUs are designed to have smaller caches and simple control structures. Their ALUs are designed for many, long latency operations that are heavily pipelined. They have a massive number of threads that compensate for the latency in terms of throughput. GPUs are focused on reducing the throughput of the task.

A CPU is a latency oriented device while a GPU is a throughput oriented device. The CPU acts as the host and the GPU takes orders from the CPU to perform parallel operations. An operation that has lesser latency on a CPU will have a higher latency on a GPU. But due to parallel operation, the GPU will have higher throughput. A CPU controls the GPU to perform an operation. It tells the GPU to launch as many threads as the number of parallel operations. The GPU is undoubtedly faster than the CPU as it has simpler control structures and caches. It gets the data it needs from the CPU and hence does not have memory access overheads. GPUs are highly efficient when working on parallel codes while CPUs perform well on serial code.

### Intel Xeon E5-2670 CPU

The Intel Xeon E5-2670 v1 is an 8 core 16 thread CPU with the 32 nm Sandy Bridge microarchitecture. It has features like on-die connect, high socket to socket bandwidth and high cache bandwidth. It uses Intel’s QuickPath Interconnect (QPI) which facilitates high speed point to point links within the processor and to the I/O hub. These are faster than parallel buses and enable in providing the maximum memory bandwidth of 51.2GB/s. These features make this processor 80% faster and 50% more power efficient than its predecessors [7]. Each core has its own L1 and L2 cache and a shared 20MB L3 cache as seen in the figure 2. The L3 cache is divided into slices, one associated with each core, although the cores access the entire L3 cache. It has all the components are linked with a ring interconnect. The bus is made up of four independent rings – data ring, request ring, acknowledgement ring and snoop ring. The ring architecture brings more bandwidth and scalability and reduces latency.

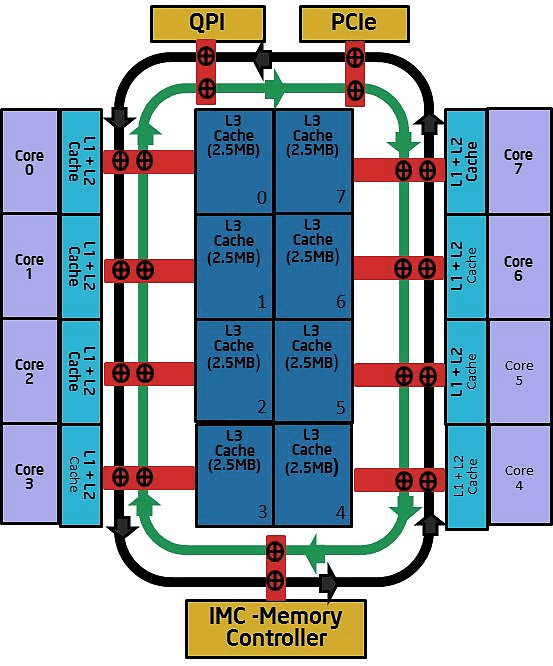


Figure 3: Intel Xeon E5-2670 CPU architecture

### Intel Xeon Phi 5110P

The Intel Xeon Phi 5110P is an accelerator based on Intel’s Many Integrated Core (MIC) Architecture that combines many cores on a chip. It can work as both a standalone processor as well as a co-processor.

The Xeon Phi 5110P is composed of processing cores, caches, memory controllers and a high bandwidth bidirectional ring controller (figure 4). It has a total 60 simple x86 architecture based processing cores each of which operate at 1.053 GHz. Each core consists of a L2 cache that is fully coherent with a global-distributed tag directory (TD). There are 64 tag based directories. The cache coherency is maintained by globally owned locally shared (GOLS) coherency protocols. At eight memory controllers the bidirectional memory with 5.5 GT/s (Giga transfers per second) offer a peak memory bandwidth of 352 GB/s. All the components are connected together by the ring interconnect as shown below [5].

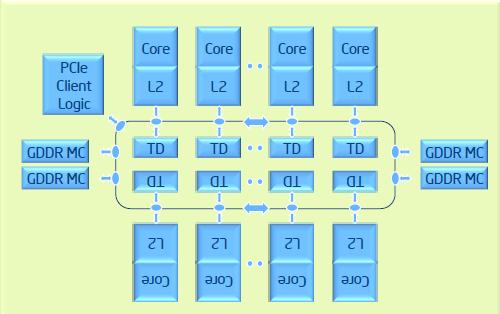


Figure 4: Micro Architecture [5]

An important component of the Intel Xeon Phi 5110P core is its vector processing unit (VPU). The VPU features a 512-bit SIMD instruction set. It can execute 16 single-precision (SP) or 8 double-precision (DP) operations per cycle. The VPU supports Fused Multiply-Add (FMA) instructions and can execute 32 SP or 16 DP floating point operations per cycle. It also supports integers. The VPU also has an Extended Math Unit (EMU) that operates by calculating polynomial approximations of the various functions [5].

### Nvidia Tesla K20X GPU

Nvidia Tesla K20X was designed for double precision float applications and delivers ten times the performance of a single GPU [1]. It contains one Kepler GK110 GPU and was designed exclusively for servers. The block diagram of the Tesla K20X is shown in figure 4.



Figure 5: Tesla K20X block diagram

With a total of 1.7 billion transistors, the Tesla is well suited for aggressive high performance computing. It is equipped with features like dynamic parallelism and Hyper-Q. These features enhance CPU utilization and simplifies parallel programming design. Dynamic parallelism [3] allows GPUs to automatically spawn new threads. It enhances parallel programming by adapting to the data without having to go back to CPU every time. The Hyper-Q [3] feature allows multiple CPUs to utilize the GPU simultaneously thereby reducing CPU waiting time and increasing programmability. The GPU has a hardware thread scheduler as opposed to the software based thread scheduling in the CPUs. It has 14 streaming multiprocessors (SMX), each with their 192 cores operating at 732 MHz. It is has a total of 2688 single precision CUDA cores and 896 double precision CUDA cores. The cores are designed to provide 3 times the performance per watt over the Fermi architecture.



Figure 6: Dynamic Parallelism [1]



Kepler – 32 simultaneous MPI tasks

Traditional GPU – 1 MPI task at a time

Figure 7: Hyper-Q[4]

## Performance Evaluation of the Devices

### Computational Density (CD) and Realizable Utilization (RU)

We would be using Computational Density metric to do comparison between different device architectures. CD, described in detail in [13] provides information about the relative performance in terms of bit, integer and floating-point operations and also includes power consumption and memory constraints. This information will aid developers in fast device exploration for efficient target device selection. CD is a measure of a device’s theoretical computational capacity.

Because of application characteristics, tools and user experience, it is not possible to achieve the device’s CD. Realization Utilization (RU) is another metric, described in [14], which we are employing, which allows developers to estimate application’s actual performance on a particular architecture. In other words, RU is the percentage of the theoretical performance of a device architecture a kernel is achieving. This information will provide valuable insight to developers during both the development cycle and coding stage of the application.

The peak performance parameters for the different devices selected have been summarized below.

Table 1: Peak computational density of the different devices

|  |  |  |  |
| --- | --- | --- | --- |
| Vendor | Intel | | NVIDIA |
| Device | **Xeon E5-2670** | **Xeon Phi 5110P** | **NVIDIA Tesla K20X** |
| Type | CPU | CPU | GPU |
| Process | 32 nm | 22 nm | 28 nm |
| Int8 (Peak CD) | 1267.2 | 1074.06 | 1897.72 |
| Int16 (Peak CD) | 633.6 | 1074.06 | 1897.72 |
| Int32 (Peak CD) | 316.8 | 1074.06 | 1897.72 |
| SPFP (Peak CD) | 422.4 | 1074.06 | 1897.72 |
| DPFP(Peak CD) | 211.2 | 568.62 | 632.58 |
| TDP (Watt) | 115 | 225 | 235 |

## Programming Models

A parallel programming model is a bridge between is a developer’s model of an application and implementation of that application on the hardware. These programming models exist as an abstraction above hardware and memory architectures. There are many ways to classify programming models but the one to our interest is on the basis of process interaction. Under this classification the programming models can be broadly divided into two groups.1.Message passing 2.Shared memory.

Message passing model-In a message passing model the exchange of data between processors takes place through explicit send and receive messages. In such a case the communication channel may prove out to be a bottleneck if the number of processors is very large since every processor would want to gain access to the communication channel.eg: MPI. MPI is a specification for message passing operations. It defines one process as the master node and the other as worker nodes. The worker management is done implicitly. One only needs to tell the MPI runtime the number of processors. The runtime infrastructure does the worker management on behalf of the users. MPI broadly classifies its message passing operations as point to point and collective. MPI\_Send/MPI\_Recv are point to point operations that facilitate communication between a pair of processors whereas MPI\_Bcast is a collective operation that facilitates communication between more than two processors.

Shared memory model-In a shared memory model parallel tasks share a global address space. The read and write to the shared memory is asynchronous i.e. tasks need to access the shared variable and only one task can access the shared variable at a time. This is done by protection mechanisms like locks and semaphores.eg UPC.UPC is a parallel programming language for shared memory architecture. The programmers view the address space as one global address space which is partitioned into a number of per thread address spaces. To improve the access to address spaces the UPC has the concept of thread affinity through which it optimizes memory access performance. The workload partitioning in UPC could be either implicit or explicit. There are several synchronization methods that UPC adopts like barriers and locks.

### CUDA

CUDA(Compute Unified Device Architecture) is a parallel programming platform invented by NVIDIA to exclusively program their GPUs. CUDA allows parts of an algorithm to be executed on the device as kernels. Only one kernel is executed at a time and these kernels have several threads doing computations in parallel. The primary advantage of CUDA threads is that they have very little creation overhead and facilitate instant switching.

A kernel is executed as a grid of thread blocks. A thread block is a batch of threads that can cooperate with each other. However, threads in different blocks cannot cooperate. Threads and blocks have IDs which help decide what data each thread should work on. It also simplifies memory addressing when handling multi-dimensional data.

CUDA’s memory model is divided into five parts: registers, local memory, shared memory, global memory and host memory. Registers and local memory can be accessed by the threads to which they belong. Shared memory can be accessed by threads that belong to the same block and global memory can be accessed by all threads. Host memory is the memory on the CPU. Data is transferred from the CPU to the GPU before the kernel is launched and copied back to the CPU after execution.

The CUDA is accessible to developers through CUDA-accelerated and extensions to standard programming languages, including C, C++ and Fortran. C/C++ programmers can use 'CUDA C/C++', compiled with "nvcc", NVIDIA's LLVM based C/C++ compiler and Fortran programmers can use 'CUDA Fortran', compiled with the PGI CUDA Fortran compiler from The Portland Group.

Moreover,CUDA platform supports other computational interfaces like Khronos Group's OpenCL Microsoft's DirectCompute, and C++ AMP. Third party wrappers are also available for Python, Perl, Fortran, Java, Ruby, Lua, Haskell, MATLAB, IDL, and native support in Mathematica.

### OpenMP

OpenMP(Open Multi-Processing) is a shared memory programming model that enables multi-processing in C, C++ and Fortran. OpenMP uses several parallelization directives to enable easy work sharing in for loops. It follows the fork and join pattern of execution. Initially a master thread starts the execution and until the parallel region is encountered. The master threads spawns worker threads and shares the work among them so that they can execute the parallel region simultaneously. After executing the parallel region the worker threads terminate and the master threads continues execution.

The data given to the threads can be either shared or private. Shared data is available in the shared memory and can be accessed by all threads. Private data can only be accessed by only that thread to which it belongs. OpenMP directives have an implied barrier for synchronizations which can also be disabled. OpenMP is a scalable, portable and easy to implement programming model.

# Implementation and Results

## Experimental Setup

### Hardware

Implementations of kernels are carried out on three devices of the NSF Center of High-Performance Reconfigurable Computing (CHREC). Specifications of these devices are shown below:

1. Intel Xeon E5-2670, Clock speed – 2.6Ghz, Maximum Memory Size = 384 GB, Maximum Memory Bandwidth = 51.2 GB/s.
2. Intel Xeon Phi 5110P, Clock Speed – 1.053 Ghz, Maximum Memory Size = 8 GB, Maximum Memory Bandwidth = 320 GB/s.
3. Nvidia Tesla K20X, Clock Speed – 732 Mhz, Total Memory Size = 6 GB, Maximum Memory Bandwidth = 240 GB/s.

### Software

1. Access to all devices was available through Nautilus cluster where remote access was done using putty and WinSCP.
2. Using APIs (Application Programming Interfaces) of CUDA and OpenMP for implementations on Tesla K20X and parallel implementations on Xeon E5-2670 and Xeon Phi 5110P.
3. Usage of ATLAS, MKL and cuBLAS libraries for code optimizations.
4. For Xeon Phi 5110P, Coprocessor native execution mode was used where the kernel was cross compiled on Host device (Intel Xeon E5-2670) before transferring it to Xeon Phi 5110P.

## Sobel Filter

### Code Optimizations and Calculating the Operationss

#### Memory optimizations

Since a grayscale pixel varies value from 0 to 255 the image has been stored as a ‘char’ to optimize memory usage such that the image can be accommodated in the main memory. To approximate a 2048x 2048 image would approximately take 222, i.e. 4 MB of memory. This is smaller than the main memories available on all the devices and hence there shouldn’t be any extra overheads while fetching the data from the main memory.

The image was tested for both 1-D and 2-D arrays for CPU implementation but it was observed that 1-D performed remarkably better in terms of throughput. This could be attributed to the fact that in a one dimensional array the memory will be contiguous with respect to a two dimensional array. As the kernel involves a lot of data movement, one dimensional allocation occupies cache in a better manner implying that two dimensional allocation has greater chances of facing conflict misses and hence leading to the increased execution time. For these reasons the image has been instantiated as a 1-Dimensional array

#### Parallelism

The kernel offers a huge amount of wide data level parallelism in the sense that it only reads data and doesn’t have to wait on any other pixels to be executed first. This means that there is no serialization in the code. The only serialization that is possible is while calculating the magnitude of the new pixel from the gradients obtained by the two masks. However, this does not happen as the two gradients are calculated by the same thread. Hence, the kernel does not face any inter thread dependencies. The sobel filter uses two 3x3 masks to perform the filtering operation in the X and Y axis. Task level parallelism observed as the gradient in X and Y direction can be calculated independently, this parallelism however has not been explicitly utilized. Loop unrolling has been done for both the mask operations for a particular pixel.

The kernel however is both memory and compute intensive. This mask operation for each pixel has to fetch 8 neighboring elements for processing one mask. This makes the kernel memory intensive as it may have to access threads which have the neighboring elements and causing bandwidth contention and eventually a slowdown.

Since, the application is also compute intensive, the operations that need to be done have to be counted appropriately. Since the boundary of the image cannot be subject to masks, the boundary of the edges are kept the same as the input image. This operation is just memory movement and does not involve any computation. Also, in each of the masks three of the values are zero, so from compiler optimization point of view, those calculations do not exist. To make sure that this does happen, it has been explicitly written as ‘0’ in the kernel description for the codes on all three devices. Incorporating such change did result in speedup on the same device as the masks wouldn’t be fetched from memory every time a gradient is calculated.

#### Total operations

The masks take a total of 21 operations in terms of additions and multiplications. The maximum number of operations occur while calculating the magnitude of the resultant output pixel. The individual gradients in X and Y direction have been squared and added to produce an intermediate result, which takes 3 operations. The square root of this intermediate value produces the magnitude. The square root function here takes the maximum number of operations. The operations are calculated using the basic mathematical operations [16] that would be required to obtain the result. The number of operations in square root will also depend on the number of places calculated before and after decimal. Since up to 360 whole squares are possible in the kernel for image pixels, the total number of operations have been taken in proportion to number of operations required for calculation of the outputs lying in the unit’s, ten’s or hundred’s place. Irrespective of the digits before decimal, the square root function always returns values till 6 places after decimal. Also, calculation for each individual digit requires 7 operations. Therefore the total number of operations for a square root can be given as, (9/360)\*(7\*1+7\*6) + (99/360)\*(7\*2+7\*6) + (252/360)\*(7\*3+7\*6) = 63 operations. The total number of operations turn out to be 86 (63+23) per pixel for CPU. Since in the GPUs fused multiply accumulate is counted as one operation, the total number of operations get reduced to 63 per pixel. Another factor that has been considered while calculating the total operations is the boundary treatment of the kernel. Since, the masks cannot be implemented on the edges of the image these pixels have been preserved and are same as the input image pixels. This also reduces the net area which is processed by the mask to (N-2)\*(N-2) for an N\*N size image.

Here, when the computation to communication ratio is calculated per pixel of the kernel it turns out to be 10.75, making it a compute intensive kernel. The same ratio for the GPU however turns out to be a little less 7.875, which can explain the results later. It should also be considered that the data movement from memory takes more cycles and is comparatively slower with respect to processor speed. The data sizes in the document are represented as ‘N’ for an image size of NxN, i.e. there are a total of N2 data points available.

### Sobel Filter on Intel Xeon E5-2670 CPU

As the size of the image increases the time taken to finish execution increases as expected, this trend remains constant as the number of threads increase.

As the number of threads increase for a given image size the execution times remain relatively constant. However for cases like extremely small image size when a large number of threads are spawned the overhead in spawning threads is much greater and hence results in higher execution times for such cases.

Since the kernel has a lot of data movement, it manifests itself in a way that it performs better at smaller number of threads, i.e. at 4 where all the communication is limited within at most 4 cores (assuming 1 thread per core is spawned). The fast QPI provides high speed data transfer at 25.6 GB/s in each direction and since only 4 cores need to communicate, the latency is pretty low. When it reaches 8 threads the performance is comparable to that of 4 threads but slightly lesser, this could be attributed to the fact that now the same data has to be moved across 8 cores (again, assuming OpenMP spawns one thread per core) and the overhead of additional thread. This data movement can lead to the increase in time. It is also possible that at lesser number of threads the communication is more coarse grained and uses the available bandwidth more efficiently. This assumption realizes itself as the execution time keeps increasing as the number of threads keep increasing. This memory intensive aspect of the core makes the execution time dependent on the image size and not the number of threads spawned. In addition to the data movement, the overhead of spawning larger threads also increases the execution time. For lower image size when higher number of threads are spawned they do not have enough work to keep them occupied.

Figure 8: Execution times for a 1024x1024 image on Intel E5-2670 CPU

Spawning 32 threads shows abnormal behavior and results in extremely high execution time. Even though there is an abrupt change in timings with respect to the other threads, the execution time increases as the image size increases. The distinct increase in execution time for 32 threads could be possibly related to the fact that the E5 has a maximum of 8 cores per chip and in the test setup there are two E5 chips on the motherboard which reaches its maximum hyper-threading limit at 32 threads. This communication pattern sees the inter-chip communication which contributes to the higher execution time when 32 threads orchestrate.

The highest number of GOPS achieved by the CPU implementation is 2.91 at four threads for an image size of 2048x2048 i.e. a total data size of 222, where the communication is limited within four chips and there is less traffic on the bus. This study concludes that the sobel filter kernel is not scalable on the CPU. Although the CPU computation is very fast the execution time is limited by the data movement involved. This is also the reason why a solely serial baseline (refer Table 2) with no parallel overhead performs way faster than the parallel OpenMP implementation on the CPU.

Figure 9: Execution time (ms) for the different image sizes and threads on Intel Xeon E5-2670 CPU

### Sobel Filter on Intel Xeon Phi 5110P

The parallel OpenMP code is compiled with the ‘mmic’ flag for the mic architecture and then natively run on the Xeon Phi 5110P coprocessor. The kernel has a lot of data movement and this does not make it suitable to be implemented on a ring bus architecture. Since each output element requires 8 other pixels from different rows as the image size increases the distance of these pixels from each other will increase and hence the probability of the data being in the neighbor’s cache decreases. This leads to all the more data movement. The application can be termed communication bound. Since in a ring bus it might take more time to for data to go around from one core to another this slows down the application further more. The many core architecture is capable of providing higher throughput but the performance is limited by the communication involved in the kernel. In worst case scenarios where the data is scattered absolutely randomly, i.e. in case of a 64x64 image size across 60 cores (assuming one thread per core), each core will approximately get one row of the image. Each output pixel would have to fetch approximately 6 bytes (each pixel is 1 byte and 2 of the pixels have higher chances of being in the same cache line) per computation. Although the ring bus provides a high memory bandwidth of 320GB/s spread across 60 individual cores, in this case where it has to transfer 6 byte/core it will face higher latency and will not be able to fully utilize the bandwidth. The latency will be both because of scheduling and randomness that might be present in spawning the threads. It should be noted that in this case only 3 of the bytes would come from the same core. The ring bus would face an extreme case of resource contention and the cores will not be able to deliver the throughput they are designed for, as latency forms a higher fraction of communication time for smaller transfers. This case can worsen where a row itself gets distributed among different cores. The architecture will then persistently see fine grain communication requests, which will render the high memory bandwidth useless. A study by Fang et. al. [23] shows that the memory access latencies across cores can vary by as 242 to 248 cycles in a shared memory state. This can also explain the relatively high latency and lower memory bandwidth. The memory bandwidth is also impacted by pre-fetch optimizations in the code, which have not been utilized in this setup.

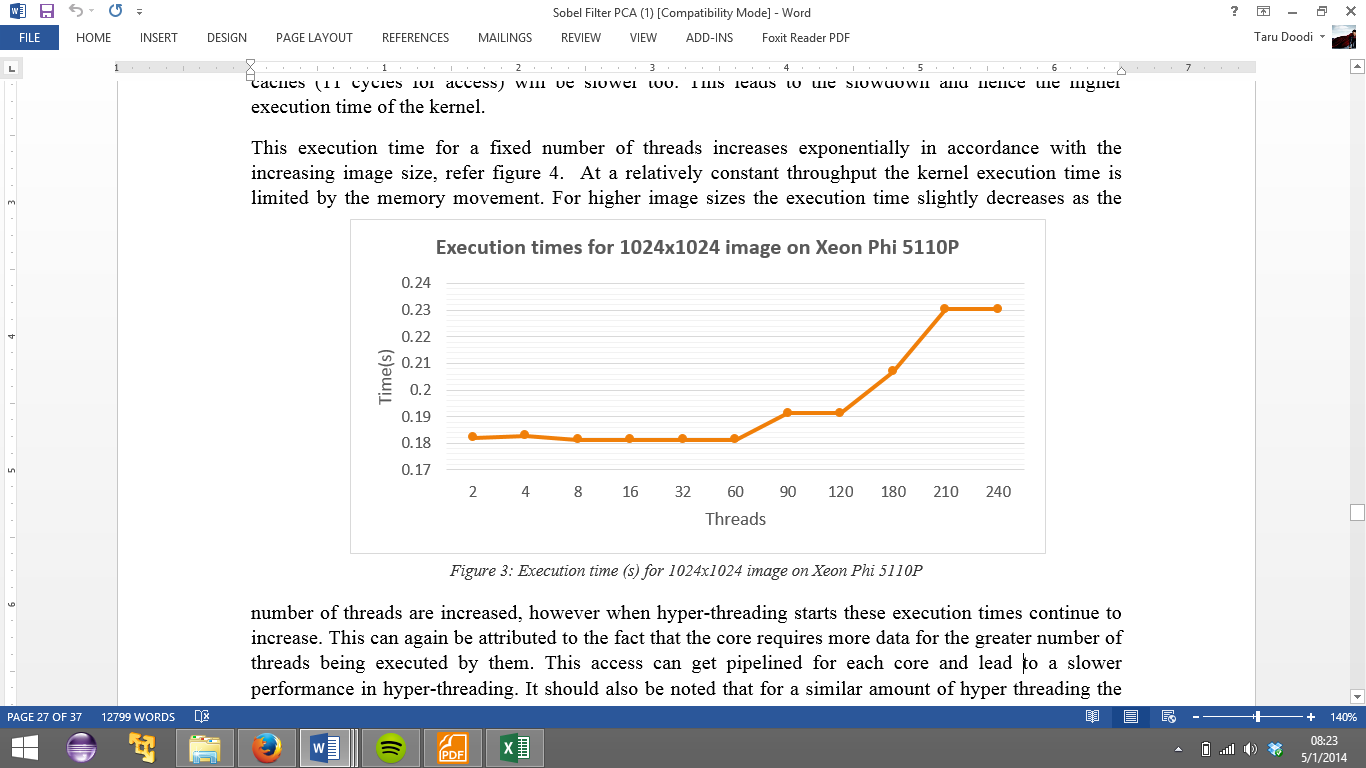


Figure 10: Execution time (s) for 1024x1024 image on Xeon Phi 5110P

While executing smaller image sizes the overhead in spawning threads is high and cannot be amortized, which results in higher execution times. The distributed tag directory based shared memory of Phi will have the data scattered across different processors which result in higher data fetch times on a ring bus. Since the top and bottom rows of the image will always need extraneous communication, a fixed amount of communication will always occur in all possible scenarios. Even though the kernel basically performs reads from the memory for different pixels, since the core clock is slower, the accesses from the L2 caches (11 cycles for access) will be slower too. This leads to the slowdown and hence the higher execution time of the kernel.

This execution time for a fixed number of threads increases exponentially in accordance with the increasing image size, refer figure 18. At a relatively constant throughput the kernel execution time is limited by the memory movement. For higher image sizes the execution time slightly decreases as the number of threads are increased, however when hyper-threading starts these execution times continue to increase. This can again be attributed to the fact that the core requires more data for the greater number of threads being executed by them. This access can get pipelined for each core and lead to a slower performance in hyper-threading. It should also be noted that for a similar amount of hyper threading the execution times remain constant. Even at a slightly smaller number of threads when the work division isn’t uniform there can be a slight increase in execution time as some cores would have to do more work than the others.

Figure 11: Execution time (ms) for all threads and image sizes on Intel Xeon Phi 5110P

Since the execution times for a thousand runs were very high for sizes beyond 2048x2048 images, the code was run for a 10 loops along with one non-timed section which would eliminate the higher execution times which could be attributed to the initial cold cache misses. Although this is true, tests were run for both cases i.e. with and without cache warming and the results did not show a very high variation. The resultant GOPS for such processing varied only by a 0.001 GOPS. Thus, it is safe to say that the overheads due to cold cache misses get amortized over a thousand loops.

Since the memory movement over the ring bus creates a major communication overhead it was attempted to reduce the overhead by implementing aligned memory accesses using the alignment feature made available by the ‘mkl\_malloc’ in the MKL library provided by Intel. This alignment isn’t often helpful as although it allows alignment of the overall dataset, it does not necessarily provide alignment for each thread, i.e., each thread working will not have aligned access [21]. This is true because each thread will access the different parts of the shared input data structure which wouldn’t be aligned with alignment provided for the main data structure. Hence, the ‘mkl\_malloc’ was found unsuitable and discarded for the sobel filter implementation.

### Sobel Filter on Nvidia Tesla K20X

The kernel runs miraculously fast on the K20X as if the architecture were custom made for it, which it indeed is! The GPU accelerators perform image processing kernels very fast because of the data level parallelism offered it. This wide parallelism can be exploited by the thousands of cores within the streaming multiprocessors which can simultaneously access the data in the global memory. The performance improves as the number of threads per block decrease. The performance improves as the threads change from 1024 to 64 per block but beyond that the execution time starts to increase again and the throughput decreases.

Figure 12: Execution time (ms) for a 1024x1024 image on Nvidia Tesla K20X GPU

As the blocks are more evenly spread out across the cores it reduces the contention to access the global memory per block. The communication in such scenario gets evenly spread out among different blocks when each block has less number of threads, which might get serialized if more number of threads are spawned within a block. A thorough research has been done by Lee and Sung [3] where they have profiled the DRAM accesses as the number of threads per block change. The execution time obtained in this experiment also follow a pattern similar to their results.

Figure 13: Execution time (ms) for all threads and image sizes on K20X

Even though the kernel has high communication requirement the execution times in the GPU are probably limited by computation for the best access times, since it has to do approximately 63 operations per pixel and 8 communications per pixel. The fast 240GB/s is capable of always making the data available for the next computation sequence. This keeps the GPU cores well fed with a lot of data to work on. GPUs are capable of sustaining a relatively high local bandwidth, i.e. the bandwidth available to each thread to fetch a word [15] [22]. The GPUs also show a high overlapped data reuse which can hide the data latency issues that could have been faced by this kernel. Also in situations memory accesses within a warp scheduler are coordinated memory coalescing happens [17], which would further reduce the time taken to fetch data for a thread till it is ready for the computation, with the next set of pixels. It should also be noted that the hardware warp scheduler for spawning threads does not generate as much overhead as the OS based software thread spawning in CPU would.

The results obtained in this experiment are without invoking the shared memory in the 64 KB L1 cache, a considerable speedup can be expected with the use of shared memory which would make GPU’s performance significantly better than the current performance.

### Performance across the different architectures

The CPU and the GPU show relative speedup as compared to the serial baseline. The Phi shows a heavy decline in performance, which is basically because of the extensive communication required by the kernel and the slower processor clock. The kernel performed the best on the Nvidia Tesla K20X where it can make use of the many accelerator cores without being limited by the high communication latencies. The following table shows the best performances across the various platforms which would be best suitable to be implemented across the various data sizes.

Table 2: Least Execution Times and Maximum GOPS for all data sizes for all devices.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N (Image  Size = N\*N) | Minimum Execution Time (ms) | | | | Maximum GOPS | | | |
| Serial | E5-2670 | Xeon Phi 5110P | K20X | Serial | E5-2670 | Xeon Phi 5110P | K20X |
| 4 | **0.001** | 0.004 | 0.023 | 0.013 | **0.344** | 0.086 | 0.015 | 0.020 |
| 8 | **0.005** | 0.005 | 0.023 | 0.013 | **0.619** | 0.619 | 0.135 | 0.176 |
| 16 | 0.01 | **0.009** | 0.058 | 0.013 | 1.686 | **1.873** | 0.291 | 0.959 |
| 32 | 0.042 | 0.031 | 0.176 | **0.013** | 1.843 | 2.497 | 0.440 | **4.472** |
| 64 | 0.173 | 0.12 | 0.689 | **0.013** | 1.911 | 2.755 | 0.480 | **18.550** |
| 128 | 0.701 | 0.478 | 2.781 | **0.014** | 1.948 | 2.856 | 0.491 | **70.041** |
| 256 | 2.822 | 1.915 | 11.229 | **0.02** | 1.966 | 2.897 | 0.494 | **201.553** |
| 512 | 11.486 | 7.698 | 45.206 | **0.044** | 1.947 | 2.906 | 0.495 | **376.040** |
| 1024 | 46.017 | 30.878 | 181.276 | **0.135** | 1.952 | 2.909 | 0.496 | **487.079** |
| 2048 | 184.282 | 123.713 | 726.237 | **0.501** | 1.954 | 2.910 | 0.496 | **526.804** |

For extremely small images, it is the best to use a serial implementation on the CPU which runs at a faster clock and has least memory movement for the kernel. For an intermediate size a parallel implementation on CPU provides a better performance. It can be noted that the kernel is embarrassingly parallel in terms of data level parallelism and at smaller sizes it doesn’t have enough data to work on. Once the sizes become considerably large, the GPU can exploit this and outperform all the other devices by a long stretch. The Xeon Phi 5110P however is not capable of exploiting data level parallelism so much as the ring bus turns out to be a bottleneck in terms of extensive memory movement. This degrades its performance tremendously as compared to the other platforms. Table 2 also throws lights on relative speedups possible on the different devices. For the larger data sizes (2048x2048) while the parallel Xeon Phi 5110P implementation sees a slowdown by 4x, a parallel CPU implementation can achieve best speedup of 1.48 over the serial implementation. The GPU however shows a huge leap in speedup by performing 368 times faster than the basic implementation. The detailed performances for individual threads have been put in the appendix. Figure 14 shows the maximum achievable GOPS for each of the devices.

Figure 14: Maximum achieved CD for different devices

## Dot Product

### Dot Product on Intel Xeon E5-2670 CPU

Dot Product Implementation was first carried out on Intel Xeon E5-2670 which is our baseline. It has 8 cores with 2 threads per cores resulting in total of 16 threads that can work in parallel. Three Implementations were carried out on E5 which are as follows:

1. Serial Optimized Implementation using cblas\_ddot (double precision dot product) function of ATLAS (Automatically Tuned Linear Algebra Software) library.
2. OpenMP implementation using cblas\_ddot (double precision dot product) function of ATLAS library.
3. OpenMP implementation using reduction clause[26].

Results for all the implementations are shown in below graph.

Note: Execution Times shown for OpenMP implementations are of the best performing threads.

Figure 15: Performances of various implementations on E5

Note: Here data sizes correspond to number of elements in each input vector.

As seen in the above chart, serial implementation (single core - single thread) using ATLAS library gives lowest execution times for data sizes ranging from 2 to 512. The primary reason for such relatively poor performance of OpenMP implementations in this range is the low data size which doesn’t fully exploit the parallelism available with 16 threads on this device. Thread spawning, data distribution, communication, ‘excess computation’ and synchronization among threads constitute the parallel overheads associated with OpenMP implementation that increase the execution time.

For data sizes in range of 1024 to 32768, performances of OpenMP reduction implementation is comparable to that of serial implementation while OpenMP ATLAS implementation takes longer time to complete execution because of inherent shortcoming of using ATLAS library in OpenMP. OpenMP, which is based on shared memory model, divides the data among the threads and performs the necessary operations. In case of this OpenMP ATLAS implementation, data distribution is time consuming since the library function doesn’t take selected individual elements (distributed by OpenMP) of vectors as inputs. Function is written as such to receive the entire vector at once and compute the dot product. Thus, significant code had to be written to divide the vector into smaller vectors and then give them as input to the library function for threads that were spawned. This takes up a lot of execution time and is the primary reason why the OpenMP ATLAS implementation fails to obtain optimal performance. An interesting observation here is when data size increased from 16384 to 32768, serial execution time increased by 2.5 times as compared to earlier data size which is because the data no longer fits in 288 KB size (L1+L2) cache (single core) size and it has to be fetched from L3 cache.

For data sizes beyond 32768, OpenMP reduction implementation performs better than serial implementation since each thread gets more data to work and perform more computations in parallel which amortizes the parallel overhead costs described in the above paragraph. Another reason is that since a single core is in charge of the serial implementation, it would have to fetch data from L3 cache as large data sizes (32768 and above) would not fit in 288 KB L1+L2 Cache (on core cache), resulting in increased latency. For parallel implementation, data associated with each core would be less and ring-style interconnect of Xeon E5 that links cores with L3 caches have high cache bandwidth that results in fast transfers and thus reduced latency[27]. The same interconnect would be used by single core during serial implementation but it would be accessing it far too often than its parallel counterpart. For data size of 1 MB, OpenMP reduction implementation achieves a speedup of 2.17 over serial implementation. Graph below shows the execution times for all data sizes for different threads using OpenMP reduction implementation.

Figure 16: Execution Times for different threads for all data sizes

Note: Trends for data sizes of 8,16,32,64 and128 were similar to those of 2 and 4. Hence they have not been shown in above graph for better visibility of other data sizes.

From the above graph, it can be seen that for a given data size as the number of threads is increased, the execution time understandably reduces. For data sizes below 8196, execution times for all threads are nearly same which is primarily due to small data size. Small data sizes don’t exploit the parallelism available with multiple threads and parallel overheads constitute majority of execution time. More the number of threads, more would be execution time for such data sizes as seen from spikes in above graph for data sizes of 256 and 512. But beyond 4096, advantage of parallelizing the application is clearly seen and execution time reduces as we move from 2 to 8 threads. Interestingly, the execution time increases as threads are increased from 8 to 16 for data sizes above 4096. This shows that when threads are increased from 8 to 16 for all data sizes in this range, (overhead) grows at a higher rate than the increase rate of processing elements (p) and problem size ( and so overall speedup of system goes down. Thus the dot product kernel is scalable up to 8 threads on Xeon E5-2670 architecture beyond which efficiency of the application goes down.

Another observation study found was when threads were increased to 32, execution time for all data sizes was very high. The abrupt increase in execution time for 32 threads could be possibly related to the fact that the E5 has a maximum of 8 cores per chip. In the test setup there are two E5 chips on the motherboard and at 32 threads it reaches its maximum hyper-threading limit. Also the inter-chip communication contributes to the higher execution time when 32 threads orchestrate. Below graph shows the execution time for data size of 1048576 for all threads. It can be seen that execution time for 32 threads is highest among all. Same observation is seen for other data sizes.

Figure 17: Execution Time for Data size, 220on Xeon E5-2670

### Dot Product on Intel Xeon Phi 5110P

Second dot product implementation was carried out on Intel’s many integrated core architecture having 60 cores with 4 hardware threads available in each core resulting in total 240 threads running in parallel. OpenMP reduction implementation was done on this device by varying number of threads from 2 to 240. The code used loops which were multithreaded using OpenMP pragmas and it was compiled to run in “native-MIC” mode involving heavy use of double-precision floating point arithmetic[28]. ICC (Intel C Compiler) was used to compile the code on Xeon Phi 5110P since GNU compilers are not suitable on Xeon Phi 5110P and the former possesses required optimization flags (mmic) for code to execute efficiently.

Figure 18: Execution Time (ms) for all data sizes for different threads

Note: Trends for data sizes of 8,16,32,64 and128 were similar to those of 2 and 4. Hence they have not been shown in above graph for better visibility of other data sizes.

As seen from above graph, for small to medium data sizes ranging from 2 to 32768, execution time for less number of threads is less than more number of threads. This is mainly because data size is not large enough to be divided among large number of threads and overhead cost related to thread spawning, thread communication, synchronization would be way too high for large number of threads. So as the number of threads is increased in this data range, execution time also increases. For data size of 65536, execution time for 2 threads is highest. Primary reason for this is each core’s cache size (L1 Data cache + L2 Unified Cache) which is 544 KB [29]. Two cores would spawn one thread each and (65536 x 8 bytes) input data distributed to each thread would not fit into core’s cache size resulting in main memory access through GDDR5 memory controller. This increases the latency as compared to the case when there are more cores and individual data would be resident in core’s cache. This trend continues as data size is increased beyond 65536 and execution time reduces as number of threads increase.

An observation seen from the above graph is for very large data sizes (above 131072), peak performances (least execution time) is seen for number of threads equal to 60 and 120. 240 threads are not able to obtain peak performances for any of data sizes. One of the reason for that is each core has limited resources in terms of two integer ALUs, one FP ALUs and one Vector processing unit. For 240 threads, each core would spawn 4 threads and so there would be resource contention among threads resulting in performance degradation as compared to 60 or 120 threads. Also data locality plays an important role in the bidirectional ring architecture of Xeon Phi 5110P. The nearest core from which a given core can get data is on an average 15 cores away thus increasing the latency as more number of threads is spawned. Lastly execution time is also increased due to increased latency associated with context switching (hyper-threading)[30] taking place between 4 threads.

For a data size of 1048576, execution times for different threads are shown in below chart.

Figure 19: Performance for different threads on Xeon Phi 5110P for N = 220

As the number of threads are increased, the execution time associated with a given data size reduces.  
From the above chart it is seen that execution time scales down consistently and achieves peak performance (lowest execution time) at 120 number of threads. For number of threads greater than 120, execution time increases as more threads are spawned per core resulting in performance degradation due to resource contention and data locality.

### Dot Product on Nvidia Tesla K20X

Dot product for the GPU was implemented in CUDA using cublas\_ddot (double precision dot product function) of cuBLAS (CUDA Basic Linear Algebra Space) library. Implementation included allocating GPU buffers and copying data into them and then calling library function. Execution time considered for analysis in this paper doesn’t include the data transfer time from host to device and versa. The execution time for data sizes ranging from 2 to 1048576 is shown below.

Figure 20: Execution times for data sizes using cuBLAS library for implementation

K20X has 14 Streaming Multiprocessors (SMX) which can spawn a total of 2688 cores (each core having maximum 1024 threads simultaneously. Execution time remains constant for data sizes ranging from 2 to 8196 since the required data would be resident in 256 KB of register file space and 1536 KB shared L2 cache of each SMX[31]. For the data sizes beyond 8196, data has to be fetched from 6 GB global memory resulting in higher execution time.

Execution times obtained with Tesla are lesser than that obtained for Xeon E5 and Xeon Phi 5110P but they are not as low as one would have expected from GPU. Tesla K20X having thousands of cores with data level parallelism should have performed much better for dot product kernel. Primary reason for such performance lies in inherent characteristic of the kernel. Dot product operation, especially involving large vectors, require global memory accesses without much computation on the data. In other words, dot product is bandwidth bound and will benefit from increased bandwidth resources. Each SMX has access to register file space of 256KB and 1536 KB L2 shared cache among all Streaming Multiprocessors. Thus cache bandwidth of Tesla K20X is lower than Xeon Phi 5110P and Xeon E5 and which is why performance of dot product kernel on GPU does not meet expectations. Also Tesla K20X has 6 40GB/s 64-bit memory controllers resulting in 240 GB/s memory bandwidth while Xeon Phi 5110P has 16 20GB/s 32-bit memory controllers amounting to 320 GB/s. That is why for large data sizes of vectors, Xeon Phi 5110P performs better than Tesla K20X.

### Performance Comparison between Xeon E5 2670, Xeon Phi 5110P and Tesla K20X

The graphs below show the performance comparison between architectures in terms of best execution times for different data sizes and speedups for Xeon Phi 5110P and Tesla K20X with respect to Xeon E5 respectively.

Figure 21: Comparison of execution times for different data sizes

Note: Trends for data sizes of 8,16,32,64 and128 were similar to those of 2 and 4. Hence they have not been shown in above graph for better visibility of other data sizes.

Figure 22: Speedup Comparisons w.r.t. baseline of Xeon Phi 5110P and Tesla K20X for all data sizes.

Note: Trends for data sizes of 8,16,32,64 and128 were similar to those of 2 and 4. Hence they have not been shown in above graph for better visibility of other data sizes.

For small data sizes between 2 to 16384, Xeon E5 performs better than the other devices. Xeon E5, although having half the cache bandwidth as Xeon Phi 5110P, has far lesser number of cores than the latter resulting in more close-knit compact architecture than Xeon Phi 5110P. Thus communication time between cores (or threads) would be lesser in E5 as compared to Xeon Phi 5110P. Also Xeon E5 executes instructions out-of-order, masking effects of cache misses as compared to the in-order Xeon Phi 5110P which on a cache miss would require hyper-threading to stall until data is available. When compared to GPU, Xeon E5 has higher cache bandwidth resulting in faster data access. For data sizes above 16384, Xeon E5 experiences performance degradation understandably because of lower number of threads and thus lack of task parallelizing ability than its counterparts. Xeon Phi 5110P performs better than Tesla K20X at data sizes of 524288 and above because of above explained reasons of higher cache bandwidth and higher memory bandwidth. Srinidhi, Davis and Williams in [8] had implemented Dot product using cuBLAS library function on Nvidia 9500 GT. Our results are similar to theirs of cuBLAS implementation being dominated by system and driver overheads associated with data transfers.

For any data size in dot product operation, total number of multiplication and addition operations add up to twice the data size. So total number of multiply and add operations for N is 2\*N. In case of Tesla K20X, multiply and add operation is considered as single Fused Multiply-Accumulate (FMA) instruction. Thus Total number of operations in case of GPU would be equal to data size itself. Performance comparison between the architectures in made for data size of 1 MB since it will involve maximum number of operations) in terms of Giga Operations per second (GOPS) and best execution times for each data size. Table and Graphs below shows performance comparison in terms of execution time and GOPS for data size of 1048576 respectively between the three devices.

Table 3: Least Execution Times and Maximum GOPS for all data sizes for all devices.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Data Size | Xeon E5 2670 | | | Xeon Phi 5110P | | Tesla K20X | |
| **Serial (ATLAS) Execution Time (ms)** | **Parallel Execution time (ms)** | **GOPS (Parallel)** | **Execution Time (ms)** | **GOPS** | **Execution Time (ms)** | **GOPS** |
| 2 | **0.0005** | 0.0050 | 0.0008 | 0.0430 | 0.0001 | 0.0160 | 0.0001 |
| 4 | **0.0005** | 0.0050 | 0.0002 | 0.0140 | 0.0005 | 0.0160 | 0.0003 |
| 8 | **0.0005** | 0.0030 | 0.0053 | 0.0140 | 0.0014 | 0.0150 | 0.0005 |
| 16 | **0.0007** | 0.0030 | 0.0100 | 0.0140 | 0.0029 | 0.0150 | 0.0011 |
| 32 | **0.0010** | 0.0030 | 0.0213 | 0.0150 | 0.0043 | 0.0150 | 0.0021 |
| 64 | **0.0010** | 0.0040 | 0.0320 | 0.0140 | 0.0091 | 0.0150 | 0.0042 |
| 128 | **0.0010** | 0.0040 | 0.0640 | 0.0140 | 0.0183 | 0.0160 | 0.0080 |
| 256 | **0.0010** | 0.0040 | 0.1280 | 0.0140 | 0.0366 | 0.0160 | 0.0160 |
| 512 | **0.0010** | 0.0040 | 0.2560 | 0.0150 | 0.0683 | 0.0160 | 0.0320 |
| 1024 | **0.0010** | 0.0050 | 0.4092 | 0.0150 | 0.1360 | 0.0160 | 0.0640 |
| 2048 | **0.0020** | 0.0050 | 0.8192 | 0.0160 | 0.2560 | 0.0160 | 0.1280 |
| 4096 | **0.0040** | 0.0060 | 1.3600 | 0.0200 | 0.4096 | 0.0160 | 0.2560 |
| 8192 | **0.0040** | 0.0090 | 1.8200 | 0.0240 | 0.6830 | 0.0170 | 0.4830 |
| 16384 | **0.0080** | 0.0130 | 2.5200 | 0.0280 | 1.1700 | 0.0180 | 0.9200 |
| 32768 | **0.0190** | **0.0190** | 3.5000 | 0.0320 | 2.0480 | 0.0210 | 1.5600 |
| 65536 | 0.0380 | 0.0310 | 4.2300 | 0.0390 | 3.3600 | **0.0220** | 2.9800 |
| 131072 | 0.0750 | 0.0550 | 4.7800 | 0.0460 | 5.7000 | **0.0310** | 4.2300 |
| 262144 | 0.1530 | 0.1040 | 5.0410 | 0.0550 | 9.5300 | **0.0420** | 6.2400 |
| 524288 | 0.3030 | 0.2050 | 5.1150 | **0.0650** | 16.1300 | **0.0650** | 8.0790 |
| 1048576 | 0.6060 | 0.2790 | **7.5200** | **0.0900** | **23.3000** | 0.1120 | **9.3600** |

As seen in the above table for data sizes from 2 to 32768, serial implementation using ATLAS library on Intel E5-2670 is the best choice for Dot Product implementation in terms of execution time. Between 65536 and 202144, Tesla K20X performs best while for sizes above that, Xeon Phi 5110P performs equally good and better than Tesla K20X.

Figure 23: Performance Comparison in terms of Giga Operations per second

## Quicksort Kernel

Quicksort is a sorting technique which involves a lot of comparisons. Data to be sorted is divided among the different threads and each thread sorts a smaller sequence available to it. For sorting the data the thread choses a pivot element at each step and moves the elements smaller than the pivot to the left of the pivot and the numbers greater than the pivot to the right of the pivot. After this step it chooses a pivot again for the left sequence and the right sequence [34]. Thus at each step the data depends on the data received from the previous step. Hence there is very little parallelism to exploit. Secondly after all the threads have completely sorted the smaller arrays, these arrays have to be merged together to form the final array of sorted elements. The final merge step can be done in series or in parallel. But for the merge part too data at each stage depends on the data in the previous stage which limits the amount of parallelism that can be exploited in this sorting technique. Thus this Kernel doesn’t scale as well as the other sorting methods. This Kernel may not blend into the different parallel architectures. There is a lot of inter thread communication taking place in the Kernel. Looking at this we can say that optimal performance for this Kernel should be when there is good work sharing among the threads and less communication between them so that computation time amortizes the communication time.

There are four different implementations of the Kernel on Xeon E-5 2670 and Xeon Phi 5110P.

Parallel Quicksort with Serial Merge – The original data is divided among the threads and sorting is performed on each of them in a parallel way. Finally the sorted elements from individual threads are compared and merged serially.

Parallel Quicksort with Parallel Merge- This is exactly like the parallel quicksort with serial merge except that the final compare and merge is done in parallel.

Parallel Quicksort with Serial Merge using MKL library.

Parallel Quicksort with Parallel Merge using MKL library.

### Complexity and number of Operations

Quicksort has an average complexity of N\*log2N and worst case complexity of N2.The sorting is done in place. Considering average case scenario, the number of operations being performed in the serial execution of the kernel is given by the equation

OPSCPU= N\*log2N + 2log2N  **(1)**

Where N is the input size.

The number of operations in the Parallel implementation with Serial Merge is given by

OPSE5/PHI= (threads-1)\*13 +N\*log2N +2log2N **(2)**

For Parallel implementation with Parallel Merge the equation is

OPSE-5/PHI= (N\*log2N + 2log2N) + (log2 (threads))\*[3 + (13/2)\*threads] **(3)**

For GPU the equation is

OPSGPU= BlockSize\*[5+ (N/BlockSize) \* (10 +2\*(N/BlockSize))] **(4)**

The number of operations are calculated to obtain Giga operations per second (GOPS), Computational density (CD) and Realizable Utilization (RU).

The Quick Sort kernel was implemented on a particular device by varying the number of inputs and the number of threads on the device. To evaluate the scalability of the kernel we also run the kernel across various devices.

### Quicksort on Intel Xeon E5-2670 CPU

Figure 24: Execution time (ms) for Quicksort with serial merge on Xeon E5-2670

These charts depict the performance of Quick sort on Xeon E5-2670. It can be clearly seen that for small data sizes serial baseline is better than the parallel implementations. The reason behind the ineffectiveness of parallel implementations for smaller data sizes is that the overhead time overpowers the computation time needed. This overhead time includes synchronization of threads and data movement between the cache and the memory and communication between the threads. Spawning more threads for smaller data sizes degrades the performance of the device as there is not enough data to be divided among threads and more time will spent in contention and communication between threads. But as we scale the data sizes the parallel implementations start showing an improvement over the serial counterpart. As the number of threads is increased from 2 to 4 for larger data sizes the performance of E5 improves and as the execution time decreases as seen in the second graph. This is an expected trend as the data is divided among the threads and each thread carries out considerable amount of computation with respect to the communication overhead. But when the threads are increased beyond 4 the computation time can no longer amortize the communication time. Hence we conclude that for this kernel, spawning 4 threads balances the amount of data distributed to each thread such that the computation time and communication time are similar resulting in lowest total execution time on the device. The merge part in Quicksort limits the performance that can be obtained on the device. Although the threads operate on their share of data in parallel, they need to wait a longer time during the serial merge which takes up a substantial part of the kernel execution time. Hence after a particular point increasing the number of threads actually degrades the performance. The high execution times for 32 threads could be due to movement of data between the chips which are connected by the QPI interconnect.

Figure 25: Execution time (ms) for Parallel Quicksort with serial merge for data size= 220

Figure 26: Execution time (ms) for Quicksort with parallel merge on Xeon E5-2670

This graph shows the results of the Quicksort kernel implementation with parallel sort and parallel merge parts as opposed to parallel sort and serial merge shown above. It can be seen that the parallel implementation works better than the serial baseline for big data sizes. This is because the bigger data is divided into small chunks and distributed among the threads; it can be accommodated in the L1 and L2 cache of the different cores on Xeon E5-2670. But for a serial implementation such big data cannot be in the cache and hence have to be fetched from lower L3 cache or memory. Small data values though can fit into the cache of a single core. Increasing the number of threads for small data values may lead to greater communication among threads with little computation to be done by individual threads. Thus serial baseline is better for smaller values of data. Running the merge over a thousand loops leads to a lot of communication within the threads which takes enormous amounts of time on the E5.Thus we the sort was executed over 1000 loops whereas the merge part over a single loop. This is done only to show the ratio of execution time spent on the sort part to the merge part. Since there are possibilities of interrupts occurring between the sort and merge part all other values have been obtained over a thousand loops for the merge and sort part together. The graph below compares two parallel implementations of the Quicksort kernel.

Figure 27: Comparison of execution time for the sort and merge parts and their impact on total execution time for Parallel Quicksort with Parallel and Serial Merge

It is clear from the graph plotted above that most part of the Quicksort kernel execution time is taken up by the merge part. The time spent on sorting in both the implementations is almost the same and is much lower than the time spent on merge. The second observation is that the serial merge implementation is slower than the parallel merge implementation. This is because the parallel Merge takes log2N steps while the serial merge implementations takes N steps.

The values were obtained for the best performing thread for each implementation. For very low data sizes the serial implementation is faster than the other parallel implementations. In parallel implementations communication between threads is significant as compared to computation and the MKL library too isn’t able to exploit much parallelism for such small data values. When the data size increases parallel merge with MKL implementation performs the best closely followed by the serial merge with MKL. The Quicksort routine used here accelerates the application performance by taking advantage of various optimization techniques (like prefetching and branch prediction) which cannot be exploited by the program without an optimized library. The complier identifies special instructions in the routine and optimizes the code accordingly. The gap between the execution time of the serial baseline and other parallel implementation widens as the data sizes increase because data beyond 288KB doesn’t fit into the (L1 + L2) cache [32]. So for a serial implementation a single core would have to fetch data from the lower L3 cache or the main memory, resulting in performance degradation for large data sizes due to capacity misses. While in a parallel implementation the data is divided among different cores and these smaller chunks can fit in caches of the individual cores or the 2 MB of shared L2 cache available thus increasing the cache hits.

Figure 28: Comparison of various implementations on E5-2670 based on their execution times with best performing number of threads for each implementation.

### Quicksort on Intel Xeon Phi 5110P

This implementation of Quicksort was done on another Intel device, Intel Xeon Phi 5110P which has 60 cores and each core has the ability to spawn 4 threads in parallel at maximum. OpenMP is used to take advantage of the distributed shared memory on the device. Although 240 threads are available the trend observed by varying threads from 2 to 128 was used to predict the behaviour for more threads. The code was complied with –mmic flag to take advantage of the mic architecture.

The Quick Sort kernel includes sorting of data and then merging the sorted values from different threads. The sort part of the kernel is parallelizable but is limited by the calculation of pivot at each step. At most different threads can work on smaller groups of the original data but there is huge data dependency at each step. Moreover in the second part of the implementation i.e. the merge part introduces a lot of data movement.

Xeon Phi 5110P shows an increase in the execution time as the number of threads for the same data size increases for the parallel Quicksort implementation with serial merge. This is because of a greater communication overhead. The fastest execution is seen for 2 threads for small data values and 4threads for larger data sizes, beyond that the performance degrades. This is due to the memory intensive and data dependent nature of the Quicksort kernel. Since at each step of merge, data has to be fetched from a different thread, communication time dictates the execution time. Ring bus architecture is unsuitable for such an application as the nearest node that a node can fetch data from is on an average 15 nodes away. Xeon Phi 5110P is optimum for kernels that are computation intensive and not memory intensive as it puts too much pressure on the ring bus. On a ring bus it might take longer time than on a normal bus to move data from a core at one end to another at the other end. Even though there are 60 cores on the Xeon Phi 5110P, the memory intensive kernel performance is limited by the communication involved. For small data sizes increasing the number of threads will lead to unnecessary overheads in spawning the threads. Also there isn’t much computation to be performed by each thread.

Figure 29: Execution time for Quicksort with Serial merge on Xeon Phi 5110P

When the data size is increased the best execution time is obtained with 4 threads. The computation and communication time balances out for 4 threads and an optimum performance is achieved. As we increase the threads the lack of scalability of memory/communication intensive kernels on Xeon Phi 5110P is seen. Since Xeon Phi 5110P has a distributed shared memory data could be scattered across different cores. Fetching of data from the shared cache which resides on another processor makes the implementation on Xeon Phi 5110P slower.

Figure 30: Execution time for Quicksort with parallel Merge on Xeon Phi 5110P

For parallel Quicksort implementation with parallel merge we can see a similar trend as the serial merge. For smaller data sizes increase in number of threads for a particular data size degrades the performance. But for larger data sizes the performance degrades slowly and the execution time is much lesser than the parallel implementation with serial merge. This is due to the fact that the amount communication between the threads after each stage drastically decreases. There are log2N communication stages and unlike the serial merge implementation a single thread doesn’t have to receive data from all other threads. Thus the communication bottleneck is reduced and hence an improvement over the serial merge implementation is seen. But even for parallel implementation it can be seen that for a given data size as the number of threads is increased the execution time increases. The kernel is limited by memory movement. As more threads are spawned data would be scattered among the threads and access to the data on threads on different cores can get serialized over the ring bus.

### Performance across the different architectures

In this section we discuss how each device performs with respect to another. The graphs below show the speedup obtained on different devices with respect to the serial baseline on Xeon E5 2670.

The results show that for small values of data increasing the number of threads doesn’t help in any architecture. The serial baseline always performs better for smaller data sizes. Increasing the number of threads doesn’t give any benefit. This is because the threads do not have sufficient data to work on and the overhead is more than the computation.

Intel’s E5-2670 is the best performing device for the Quicksort kernel. The best performance on the device is achieved when a small number of threads are used. Values obtained on Phi are always worse than on E5 2670. This could be because of several reasons. The clock speed of Xeon E5-2670 is 2.6 GHz whereas it is only 1.053 GHz for Xeon Phi 5110P [33]. The individual E5 cores are more powerful than the individual Xeon Phi 5110P cores. Also the Xeon Phi 5110P has a ring bus which is unsuitable for the large amount of data transfer as the latency can vary.

For larger data sizes there are more chances of data not fitting into the cache. The Xeon Phi 5110P has a smaller L1 cache than the E5. Moreover the memory access time to lower memory in Xeon Phi 5110P is more than E5-2670. If there is a miss in the L1 cache as well as L2 TLB on the Xeon Phi 5110P then the access time is huge as one will have to do through 4 levels of page tables [33]. Also the nearest node could be 15 nodes far on an average on Xeon Phi 5110P. The E5 has a bigger L1 cache and the memory access time to L2 and L3 cache is much lower.

The Tesla K20X performs similar to the Xeon Phi 5110P. Its execution time is much higher than the serial baseline but it performs relatively better than Xeon Phi 5110P for small data sizes. This is because the data can easily sit in the L1 and L2 caches and increasing number of threads will make the computation faster. But when the data size is increased Xeon Phi 5110P performs better than the GPU and Xeon E5-2670 performs the best. This is due to the reason that the kernel doesn’t have a lot of computation. Most of the execution time is spent in waiting for data from other threads and moving data among the threads. Each processor on the K20X has a L1 cache of 64KB and can access the 1536 KB shared L2 cache. Thus we can see that the cache size is smaller in GPUs than Xeon E5-2670 and Xeon Phi 5110P. Thus when the kernel uses bigger data sizes there will be cache misses due to capacity misses and hence GPU will have bad performance. Xeon Phi 5110P also has a 320GB/s memory bandwidth as compared to 240GB/s bandwidth in the GPU [REF]. So large values can travel form one core to other faster on Xeon Phi 5110P than on GPU. Xeon Phi 5110P also uses prefetching as an optimization technique which is not available on a GPU [33**].** Also the kernel on GPU is executed a bit differently. The sort part is done in parallel on the device (K20X) whereas the merge part is done serially on the host. Though the time for data movement between the host and device is not considered a major part of the execution time comes from the serial merge being implemented on the CPU which slows the GPU even more.

The table above shows the minimum execution time and maximum GOPS obtained on each device for the best performing threads. From the execution times it can be concluded that the kernel is non scalable due to huge amount of data dependencies. The serial baseline is the best for small values of data. But as the data size grows the Xeon E5-2670 performs better than the other devices. Xeon E5-2670 makes use of out of order execution and a faster clock rate to outperform the Xeon Phi 5110P. Xeon E5-2670 has lesser number of cores than the Xeon Phi 5110P.

Table 4: Minimum execution time and Maximum GOPS for three devices

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Data size | Execution Time(ms) | | | | GOPS(Giga operations per second) | | | |
| Serial | Xeon E5-2670 | Xeon Phi 5110P | Tesla K20X | Serial | Xeon E5-2670 | Xeon Phi 5110P | Tesla K20X |
| 4 | **0.001** | 0.003 | 0.035 | 0.007 | 0.012 | 0.008 | 0.001 | **0.769** |
| 8 | **0.001** | 0.002 | 0.026 | 0.007 | 0.032 | 0.023 | 0.001 | **0.776** |
| 16 | **0.001** | 0.002 | 0.027 | 0.007 | 0.080 | 0.047 | 0.003 | **0.788** |
| 32 | **0.001** | 0.003 | 0.028 | 0.007 | 0.192 | 0.068 | 0.007 | **0.813** |
| 64 | **0.003** | 0.003 | 0.030 | 0.007 | 0.149 | 0.154 | 0.015 | **0.859** |
| 128 | **0.005** | 0.004 | 0.036 | 0.007 | 0.205 | 0.259 | 0.029 | **0.961** |
| 256 | **0.005** | 0.006 | 0.045 | 0.007 | 0.461 | 0.386 | 0.051 | **1.164** |
| 512 | 0.012 | 0.009 | 0.065 | **0.007** | 0.427 | 0.570 | 0.079 | **1.603** |
| 1024 | 0.026 | 0.016 | 0.111 | **0.007** | 0.433 | 0.705 | 0.101 | **2.600** |
| 2048 | 0.058 | 0.030 | 0.200 | **0.016** | 0.424 | 0.820 | 0.123 | **2.167** |
| 4096 | 0.133 | 0.055 | 0.380 | **0.040** | 0.400 | 0.968 | 0.140 | **1.971** |
| 8192 | 0.274 | **0.111** | 0.735 | 0.128 | 0.419 | 1.033 | 0.156 | **1.698** |
| 16384 | 0.584 | **0.220** | 1.538 | 0.457 | 0.421 | 1.117 | 0.159 | **1.516** |
| 32768 | 1.250 | **0.443** | 3.489 | 1.741 | 0.419 | 1.184 | 0.150 | **1.396** |
| 65536 | 2.678 | **0.894** | 7.411 | 6.772 | 0.416 | 1.246 | 0.150 | **1.336** |
| 131072 | 5.670 | **1.881** | 15.130 | 26.696 | 0.416 | 1.254 | 0.156 | **1.306** |
| 262144 | 11.934 | **3.925** | 30.553 | 106.007 | 0.417 | 1.269 | 0.163 | **1.291** |

Thus the communication time between the threads on Xeon E5-2670 is much lesser than Xeon Phi. Xeon E-5 has a higher cache bandwidth than the Tesla K20X resulting in faster data access than the K20X. For small data values serial baseline is the best as the data is too small and communication overhead overpowers the computation time for such small data values. Xeon performs the worst when compared with K20X and Xeon E5-2670 for the Kernel. This could be attributed to its slower clock and very low cache size. Any misses in L1 and L2 cache result in costly access to lower memory. The ring bus interconnect in Xeon Phi 5110P 5110P is not efficient enough for the movement of data.

Figure 31 shows the comparison of different device architectures based on GOPS. GOPS depend on the number of operations being performed and the time taken to perform these operations. For Xeon E5-2670 the maximum GOPS were obtained for the Parallel Quicksort implementation with serial merge using the Math Kernel Library. For Tesla K20X the implementation was different from the parallel implementations on Xeon Phi 5110P .Hence the number of operations vary for each of the implementations on the devices. The number of operation taken into account for each of the implementation is the lower bound on the number of instructions. The GOPs obtained for all the devices are really low as the kernel is very memory intensive. The kernel is hugely data dependent and in almost all the implementations it was observed that threads spent more data waiting for the data to arrive or communicating with each other than on computations.

Figure 31: Max Giga Operations per Second (GOPS) achieved on each device

It can be seen that the Tesla K20X has better GOPS than the other devices when the data size is small. However when the data size is moderately large the performance of Tesla K20X and Xeon E5-2670 are comparable. The higher number of GOPS obtained for Tesla K20X is mostly due to the extra number of operations being performed in the implementation for that device .The Xeon E5-2670 shows improvement in performance as a result of decrease in execution time. Due to the memory intensive nature of the kernel, Xeon Phi 5110P and Tesla K20X are not able to extract any performance benefit in spite of having the capability to spawn large number of threads.

Although the implementation for Xeon E5-2670 and Xeon Phi 5110P were the same Xeon E5-2679 performed better. This result shows that recompiling the same implementation on Xeon Phi 5110 P will not give optimum performance on the device. Thus we can conclude that the kernel is not scalable at all on the parallel architectures considered here as there is a lot of dependency between data and very less amount of computation to be done. Even with the ability to spawn numerous threads and other optimisations like out of order execution the devices did not perform much better than the serial baseline.

Figure 32: Comparison of execution time for different implementations on Xeon Phi 5110P with best performing number of threads for each implementation

The comparison of various implementations of the kernel on Xeon Phi 5110P show that the performance of the kernel degrades substantially on Xeon Phi 5110P. The serial baseline on E5-2670 is better than any of the parallel implementations on Xeon Phi 5110P. Neither increasing number of cores nor using the optimized libraries help. Xeon Phi 5110P may support hyper - threading and improve the performance of computational kernels but they are very inefficient for communication bound kernels. The major reason behind this is the communication overhead and latency due to the bidirectional ring bus connection between the cores and the contention on execution units within a core with increasing number of threads. Access to data over the bus gets serialized and the communication between nodes (even neighbouring nodes) takes a lot of time. Thus on a whole performance on Xeon Phi 5110P is constrained by the amount of communication involved. Another important observation to be made here is that the MKL library doesn’t give the best performance on all the different architectures. These libraries are designed keeping in mind a particular architecture and do not necessarily work well on other architectures.

### Nvidia Tesla K20X GPU

The GPU is a device that can spawn thousands of threads at a time. Hence the capability to speedup kernels that are not data dependent is huge on such massively parallel devices. The kernel was implemented on Tesla K20X using Cuda.

Since the kernel is memory intensive it cannot exploit the fine grained parallelism available on the GPU. Here the implementation has been changed and the recursion in the Quicksort kernel has been replaced by an iterative call. A GPU gives maximum performance when the work is equally divided among the threads and each thread takes similar amount of time to do the work and there is enough data to utilize all the available threads. But in this implementation only one thread per block has been spawned and the number of blocks in the grid is increased. The thread count is limited to 1 as increasing the thread count would increase the amount of communication between the threads. If we increase the number of threads to large number then communication will have to take place between the threads on any of the 14 different SMXs. Moreover there is more contention among the threads on a single block than there is contention among threads on different blocks. Global memory is the largest memory area but suffers from high latency and low bandwidth. Access to the global memory should be minimized and this is done by giving each thread the total memory available to a block. Hence we increase the number of blocks in the grid. Since there is only one thread per block there is no contention within the same block. This way threads from the same block do not compete for the global memory. As the number of blocks keeps increasing the computation becomes fine grained. As each block spawns one thread the total work gets divided among a larger number of threads which is equal to number of blocks. Each thread thus works on smaller pieces of data and as can be seen from the results execution times decrease with increasing number of blocks.

Figure 33: Execution time (ms) for Quicksort with Serial Merge on Tesla K20X.The Merge part is done on the CPU.

For small data sizes increasing the number of blocks and hence the total number of threads increases the execution time as there is not enough data for each thread to work on. But for larger data sizes there is enough data for each thread to perform computation on, hence the performance improves. Another trend that can be seen here is that for small data values the timings remain same. But with increase in data sizes the execution time increases exponentially. This can be attributed to the fact that for small data sizes there is less communication between the threads. But as the data sizes increase the communication overhead between the threads overtakes the computation time as there are more number of merge steps that have to be carried out with larger data sizes. This communication is limited by 240 GB/s memory bandwidth available on K20X which decreases the performance of the K20X for the large data sizes for this kernel.

Since the kernel is high on communication and data at each step depends on the data at the previous step the kernel doesn’t scale that well on GPUs.

## Linear Spacing

### Linear Spacing on Intel Xeon E5 -2670 CPU

The Intel Xeon E5 2670 processor has 8 cores with 2 threads per core. Therefore there are totally 16 threads per chip. There are two Xeon E5s in the experimental setup. The programming language used is OpenMP. We analyse the performance of the kernel within one chip and across both chips.

Figure 34 shows the time taken taken to execute the kernel for each data size as the number of threads increase from 2 to 16 on one Xeon E5 chip while Figure 2 shows the time taken to execute the kernel by each thread group as the data size scales from 22 to 216.

In figure 34 it can be observed that for larger data sizes the time taken is greater for 2 threads and decreases as the number of threads increase. This is an expected trend as more number of threads are spawned, the work is shared and completed sooner. But for smaller data sizes on a large number of threads the overhead of spawning work is greater than the benefits of parallelism and hence tend to have higher execution times.

Figure 34: Performance on Xeon E5 - time taken by different threads for different data sizes

When there are 32 threads the time taken to execute the kernel increases significantly. This can be related to the fact that the computation is now done across two chips and involves hyper-threading. Since E5 has 8 cores per chip 16 threads can be spawned on one chip. At 32 threads the two Xeon E5s in use reach their maximum capacity.

### Linear Spacing on Xeon Phi 5110P

The Xeon Phi 5110P has Intel’s many-core architecture that has 60 cores and 4 threads per core. Linear Spacing is implemented on the Phi by varying threads from 2 to 240 for each data size. The performance of the kernel on Xeon Phi 5110P was mostly uniform. Thread groups showed a similar pattern while executing the kernel for various data sizes. As expected, they took less time to execute smaller data sizes and gradually increased execution time with the data size. There were a few spikes in time for smaller data sizes with larger number of threads which is due to the overhead of spawning work.

After analyzing the data four thread groups are chosen which were representative of the others – 2, 4, 8, 16, 32, 64, 128, 176 and 240. Figure 35 shows the execution times for these thread groups. The performance decreases as the number of threads increase. 64 performs best among them followed by 128, 176 and 240 (in that order). This can be attributed to hyper-threading. Assuming that the processor spawns one thread per core until it reaches 60 and then starts spawning two threads per core till it reaches 120 and so on, this would explain the decrease in performance as the resources will be shared among more number of threads as the total number of threads approaches 240.

Figure 35: Execution time for select thread groups for all data sizes on Xeon Phi 5110P

### Linear Spacing on Nvidia Tesla K20X

The kernel is implemented on the GPU using CUDA. For data sizes (N) from 22 to 210 one block with (N-1) number of threads was invoked. For data sizes 211 to 216 the block sizes were increased in multiples of 2 while the thread count was kept constant at 1024. This was done to not exceed the maximum threads/block limit for larger data sizes while not spawning more threads than required for smaller data sizes. While calculating the operations performed by each thread multiplication followed by an addition is considered as one operation (fused multiply accumulate). The time involved in copying data from the CPU memory to the device and back is not considered in this experiment.

Figure 36: Execution time (ms) for 216 on Nvidia K20X

An experiment was also conducted to analyze the performance of the kernel with varying threads per block. Data size 216 is chosen and the threads per block from 32 to 1024 while decreasing the number of blocks from 2048 to 64 respectively. It was observed that the time decreases as the threads per block increase from 32 to 512 but beyond that the time increases again for 1024 threads per block as shown in figure 36.

Figure 37 shows the execution times for all data sizes. It can be seen that the time is uniform from 23 to 29 and then it increases for 210 and remains constant till 216. As the threads per block reaches 1024, the performance slightly deteriorates. A through study has been conducted in [3] and 128 to 256 threads per block have been found to give best execution time. Also the time taken for 22 is slightly greater due to the overhead of work spawning.

Figure 37: Execution time (ms) for all data sizes on Nvidia K20X

### Comparison of performance across different architectures

The performance of the kernel on different architectures is compared based on the execution time, speedup and GOPS.

#### Execution Time

A comparison of the best execution times on each device for all the data sizes shows that the Xeon has the least execution time. A comparison of best execution times are shown below in figure 38.

Figure 38: Comparison of best execution times for Xeon E5, Xeon Phi 5110P and Nvidia K20X

#### Speedup

In this implementation the value of the interval is being calculated by every thread. This is an extra computation that is being performed in the parallel implementation. Alternatively it is also possible to perform that computation once and give the data to the threads. This will further reduce the number of operations and may result in a better performance. Comparing speed up between all the devices, Xeon E5-2670 has the best speedup followed by the GPU and then the Phi.

Figure 39: Comparison of Speedup between Xeon E5-2670, Xeon Phi 5110P and Nvidia Tesla K20X

#### GOPS

Table 1 shows a comparison of the GOPS for all data sizes for the serial, Xeon E5, Xeon Phi and Nvidia K20X implementations of the kernel. Serial implementation has better performance till 28 then E5 takes over from 29 to 215 and GPU has more GOPS for 216. An interesting point to note is that even though GPU has more GOPS for 216, Xeon E5 still has lesser execution time than the K20X. The K20X spawns 65536 threads and does all the computations in parallel. It also uses fused multiply accumulate that does multiplication and addition as one operation. Even though it does more number of operations in a second than the E5, it still takes more time to complete the execution. This may be due to the slower GPU core clock (732 MHz) and there are many operations to do in this kernel. Also as mentioned earlier there is an extra computation that every thread performs in this implementation which is increasing the number of operations to be performed per element generated. If the value of the interval is calculated in advance and given to each thread it can further reduce the execution time of the GPU.

Table 5: Comparison of Maximum GOPS between the Serial, Xeon E5, Xeon Phi 5110P and Nvidia K20X implementations

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Data Size | Minimum Execution Time (ms) | | | | Maximum GOPS | | | |
| Serial | Xeon E5-2670 | Xeon Phi 5110P | Nvidia Tesla K20X | Serial | Xeon E5-2670 | Xeon Phi 5110P | Nvidia Tesla K20X |
| 4 | **0.0000** | 0.0050 | 0.0140 | 0.0190 | **0.1890** | 0.0086 | 0.0104 | 0.0006 |
| 8 | **0.0001** | 0.0040 | 0.0140 | 0.0180 | **0.2419** | 0.0100 | 0.0126 | 0.0015 |
| 16 | **0.0001** | 0.0040 | 0.0140 | 0.0180 | **0.2500** | 0.0126 | 0.0992 | 0.0033 |
| 32 | **0.0001** | 0.0030 | 0.0140 | 0.0180 | **0.2635** | 0.0220 | 0.0314 | 0.0072 |
| 64 | **0.0008** | 0.0030 | 0.0150 | 0.0180 | **0.1510** | 0.0433 | 0.0413 | 0.0140 |
| 128 | **0.0010** | 0.0030 | 0.0170 | 0.0180 | **0.2550** | 0.0860 | 0.0164 | 0.0282 |
| 256 | **0.0020** | 0.0030 | 0.0190 | 0.0180 | **0.2555** | 0.1710 | 0.0270 | 0.0566 |
| 512 | 0.0030 | **0.0030** | 0.0220 | 0.0180 | 0.3410 | **0.3420** | 0.0474 | 0.1135 |
| 1024 | 0.0030 | **0.0030** | 0.0270 | 0.0200 | 0.6823 | **0.6830** | 0.0962 | 0.2046 |
| 2048 | 0.0050 | **0.0040** | 0.0320 | 0.0200 | 0.8190 | **1.0260** | 0.1293 | 0.4094 |
| 4096 | 0.0110 | **0.0050** | 0.0370 | 0.0200 | 0.7446 | **1.6400** | 0.2238 | 0.8190 |
| 8192 | 0.0220 | **0.0060** | 0.0410 | 0.0200 | 0.7446 | **2.7340** | 0.4018 | 1.6382 |
| 16384 | 0.0440 | **0.0070** | 0.0510 | 0.0200 | 0.7447 | **4.6840** | 0.6443 | 3.2770 |
| 32768 | 0.0880 | **0.0090** | 0.0630 | 0.0190 | 0.7447 | **7.2840** | 1.0462 | 6.8983 |
| 65536 | 0.1760 | **0.0140** | 0.0630 | 0.0230 | 0.7447 | 9.3630 | 1.7296 | **11.3973** |

Comparing the maximum GOPS achieved by each device, the K20X has the best GOPS closely followed by the Xeon E5. Xeon Phi 5110P performs only slightly better than the serial implementation of the kernel. Though linear spacing is a computation oriented kernel, it has only a few operations to be performed to find every element and hence the overhead of work spawning overcomes the benefits of parallelism for very large number of threads. Moreover, the Xeon E5 has a faster clock and more powerful cores than the Xeon Phi 5110P. This may also be another reason why the kernel performs better on the Xeon E5. Although the Xeon Phi 5110P has more number of cores than the Xeon E5 and has to be a competitive architecture, the work spawning overheads outrun the benefits of parallelism. This has been studied in detail in [24] and it has been concluded that the Xeon Phi 5110P can be competitive architecture only if the application provides enough parallelism for 240 threads and runtime allocation of fine grained tasks can be done with little overhead.

# Conclusion

Table 6: Realizable Utilization (%) on all the devices for different kernels

|  |  |  |  |
| --- | --- | --- | --- |
| Kernel | Devices | | |
| Xeon E5-2670 | Xeon Phi 5110P | Nvidia Tesla K20X |
| Dot Product | 3.56 | **4.1** | 1.48 |
| Sobel Filter | 0.44 | 0.04 | **27.75** |
| Linear Spacing | **4.43** | 0.3 | 1.8 |
| Quick Sort | **0.61** | 0.01 | 0.13 |

Table 7: Computational Density per Watt on all the devices for different kernels

|  |  |  |  |
| --- | --- | --- | --- |
| Kernel | Devices | | |
| Xeon E5-2670 | Xeon Phi 5110P | Nvidia Tesla K20X |
| Dot Product | 0.063 | **0.104** | 0.039 |
| Sobel Filter | 0.025 | 0.02 | **2.241** |
| Linear Spacing | **0.081** | 0.00776 | 0.048 |
| Quick Sort | **0.011** | 0.0007 | 0.011 |

Figure 40: Realizable Utilization (%) on all the devices for different kernels

The kernel shows maximum device utilization of 27% on the Nvidia K20X because of the numerous small cores capable of thoroughly exploiting the data level parallelism offered by the sobel filter kernel. Since it is compute intensive the latency offered by the slower cores at 732 MHz easily hides the intensive data fetch from the memory at 250 GB/s. Least amount of RU is obtained on the Xeon Phi where the ring bus cannot keep up with the data movement in the kernel and extremely slows down the performance. The case further deteriorates when the number of threads increase and the data is moved between farther cores.

Since, the square root function in the kernel takes only a double input, the peak CD for CPU has been taken proportional to the number of int8 and SPFP operations available in the device, which turn out to be 648 GOPS. This parameter has been used to calculate the RU for Intel’s E5-2670. Such a discrimination wasn’t necessary on the other devices as they offered similar number of peak GOPS for int8 and SPFP operations.

For dot product kernel, although the execution time obtained with Tesla K20X is lower than Intel Xeon Phi for most of the data sizes, its GOPS are much lower than the latter since GPU takes Multiply and Add as a single Fused Multiply Accumulate (FMA) instruction. Thus number of operations it performs would be half of that of each Xeon Phi 5110P and Xeon E5-2670 in the same time. Also GPU known for being huge data parallel device has maximum peak computational density for all types of instructions among all three devices. Thus Realizable Utilization of GPU is lower than that of Intel Xeon Phi 5110P and Intel Xeon E5-2670. Also for such high throughput device, TDP of Tesla K20X is also highest due to which CD/W value of GPU is lowest among the three.

# Acknowledgement

Successful completion of our study wouldn’t have been possible without the help of several key individuals. We would like to express our sincere gratitude to our mentor, Andrew Milluzzi, who met us weekly throughout the duration of project answering our innumerable queries, offering invaluable technical advice and wonderful insights from his experience in this domain. We would also like to thank Dr. Alan D. George for guiding us whenever we needed a direction, especially in emphasizing the importance of a good proposal to work on and motivating us to keep up the constant progress. In addition we would like to thank all guest lecturers who taught us different concepts of parallel programming which greatly helped us during implementation of kernels on devices.

# References

1. Nvidia,”Tesla Kepler GPU Accelerators”,www.nvidia.com.[Online].Available: http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf. [Accessed: April 3 2014].
2. Nvidia, “Tesla K20X GPU Accelerator”, www.nvidia.com.[Online].Available: http://www.nvidia.com/content/PDF/kepler/Tesla-K20X-BD-06397-001-v07.pdf. [Accessed: April 10 2014]
3. S. Lee and W. Sung, “DRAM access reduction in GPUs by thread-block scheduling for overlapped data reuse,” in *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2013, pp. 901–904.
4. Nvidia.”Nvidia Kepler GK110”, nvidia.com. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NV\_DS\_Tesla\_KCompute\_Arch\_May\_2012\_LR.pdf[Accessed: April 12 2014].
5. George Chrysos, Intel Corporation, ”Intel Xeon Phi Coprocessor – the Architecture”, [www.intel.com.[Online](http://www.intel.com.[Online)]. Available: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner [Accessed: March 23 2014].
6. G. van den Braak, B. Mesman, and H. Corporaal, “Compile-time GPU memory access optimizations,” in *2010 International Conference on Embedded Computer Systems (SAMOS)*, 2010, pp. 200–207.
7. Intel Xeon E5 Processor Family Technical Overview <http://software.intel.com/en-us/articles/intel-xeon-processor-e5-26004600-product-family-technical-overview>.
8. S. Kestur, J. D. Davis, and O. Williams, “BLAS Comparison on FPGA, CPU and GPU,” in Proceedings of the 2010 IEEE Annual Symposium on VLSI, Washington, DC, USA, 2010, pp. 288–293.
9. Stef Graillat, Jeremy Jean, "A Parallel Algorithm for dot product over word-size finite field using floating-point arithmetic", *in* *proceeding SYNASC '10 Proceedings of the 2010 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing,* pp. 80-87.
10. N.E.A.Khalid, S.A.Ahmad, N.M.Noor, A.F.A.Fadzil and M.N.Taib, “Parallel approach of Sobel Edge Detector on Multicore Platform”, *in International Journal Of Computers And Communications, Issue 4, Volume 5, 2011*, pp. 236-244.
11. Alex Merritt, Manan Mehta, Amit Warke, “Parallel Edge Detection”. [Online]. Available : http://www.cs.rit.edu/~ptt/courses/4003-531/student\_work/EdgeDetect.pdf .[Accessed: April 03, 2014].
12. Ishwari Singh Rajput, Bhawnesh Kumar, Tinku Singh, “Performance Comparison of Sequential Quick Sort and Parallel Quick Sort Algorithms”, *in* *International Journal of Computer Applications (0975 8887) Volume 57,No.9, November 2012*,pp. 14-22.
13. Jason Williams, Alan D. George, Justin Richardson, Kunal Gosrani, Siddarth Suresh, "Computational Density of Fixed and Reconfigurable Multi-core devices for application acceleration*", in ACM Transactions on Reconfigurable Technology and Systems (TRETS),* Volume 3 Issue 4, November 2010, Article No. 19.
14. Justin W. Richardson, Alan D. George, Herman Lam, “Performance Analysis of GPU Accelerators with Realizable Utilization of Computational Density”, *in SAAHPC '12 Proceedings of the 2012 Symposium on Application Accelerators in High Performance Computing,* pp. 137-140.
15. [John D. McCalpin](http://www.tacc.utexas.edu/%7Emccalpin/) (2012, Nov. 17) “Some comments on the Xeon Phi coprocessor”. [John McCalpin's blog](http://blogs.utexas.edu/jdm4372), Available: <http://blogs.utexas.edu/jdm4372/tag/memory-bandwidth/> [Accessed April 26, 2014].
16. Edison Heng (2013, April 14), “Square Root algorithm for C”. Code Project, Available: http://www.codeproject.com/Articles/570700/SquareplusRootplusalgorithmplusforplusC. [Accessed April 26, 2014].
17. S. Ramos and T. Hoefler, “Modeling Communication in Cache-coherent SMP Systems: A Case-study with Xeon Phi,” in *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, New York, NY, USA, 2013, pp. 97–108.
18. M. Bhatia, D. Yadav, Madhurima, P. Gupta, G. Kaur, J. Singh, M. Gandhi, and A. Singh, “Implementing Edge Detection for Medical Diagnosis of a Bone in Matlab,” in *2013 5th International Conference on Computational Intelligence and Communication Networks (CICN)*, 2013, pp. 270–274.
19. S. J. H. Pirzada and A. Siddiqui, “Analysis of edge detection algorithms for feature extraction in satellite images,” in *2013 IEEE International Conference on Space Science and Communication (IconSpace)*, 2013, pp. 238–242.
20. *Digital Image Processing*, Gonzalez and Woods, Second Edition, Prentice-Hall, 2002.
21. R. Rahman, *Intel® Xeon PhiTM Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, 2013.
22. Oak Ridge *Leadership* Computing *Facility* “[Accelerated Computing Overview](https://www.olcf.ornl.gov/kb_articles/accelerated-computing-overview/)”. Available <https://www.olcf.ornl.gov/support/system-user-guides/accelerated-computing-guide/>. [Accessed April 28, 2014].
23. J. Fang, A. L. Varbanescu, H. Sips, L. Zhang, Y. Che, and C. Xu, “An Empirical Study of Intel Xeon Phi,” *arXiv:1310.5842 [cs]*, Oct. 2013.
24. J. V. F. Lima, F. Broquedis, T. Gautier, and B. Raffin, “Preliminary Experiments with XKaapi on Intel Xeon Phi Coprocessor,” 2013, pp. 105–112.
25. University of Oslo, “Parallel Quicksort Algorithms”, www.uio.no.[Online].Available: http://www.uio.no/studier/emner/matnat/ifi/INF3380/v10/undervisningsmateriale/inf3380-week12.pdf [Accessed: February 17, 2014]
26. Blaise Barney, Lawrence Livermore National Laboratory. ”OpenMP”. Available: https://computing.llnl.gov/tutorials/openMP/#REDUCTION. [Accessed: February 27, 2014]
27. Reza Rahman, Intel Software and Services Group. “Intel Xeon Phi Programming Environment”. Available: http://goparallel.sourceforge.net/wp-content/uploads/2013/07/Intel\_-Xeon-Phi-Programming-Environment.pdf. [Accessed: March 10, 2014]
28. CPU-world.com “Intel Xeon E5-2670 Specifications”. [Online]. Available: http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2670.html. [Accessed : March 2, 2014]
29. ark.intel.com “Intel Xeon Phi Coprocessor 5110P”. [Online]. Available: http://ark.intel.com/products/71992/intel-xeon-phi-coprocessor-5110p-8gb-1\_053-ghz-60-core. [Accessed: March 29, 2014]
30. threadingbuildingblocks.org,”Intel Xeon Phi”.[Online].Available: http://www.threadingbuildingblocks.org/faq/26. [Accessed : April 04, 2014]
31. nvidia.com, “Nvidia Kepler GK110 Architecture”. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf. [Accessed: April 10, 2014]
32. Cpuboss.com,”Intel Xeon E5-2670” [Online].Available: http://cpuboss.com/cpu/Intel-Xeon-E5-2670 [ Accessed: April 1,2014]
33. Software.intel.com,”Intel Xeon Phi Core Micro-architecture”. [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture[Accessed: March 28,2014]
34. Wikipedia.org,”Quicksort”, [Online]. Available: en.wikipedia.org/wiki/Quicksort [Accessed: February15,2014]

e