AN ABSTRACT OF THE MASTERS PROJECT OF

William Leslie for the degree of Master of Science in Computer Science presented on September 1, 2016.

Title: Benchmarking and Programming Practices for the Intel Xeon Phi Coprocessor.

Abstract approved:

_________________________________________________
Mike Bailey

ABSTRACT: The Intel Xeon Phi is a relative newcomer to the scientific computing scene. In the recent years, GPUs have been used extensively for mathematical simulations. The Xeon Phi is Intel’s response to the use of these cards. Like the GPU, it is highly parallelizable but can be programmed like a CPU. Since the card is relatively new, little literature is written about it. The focus of this paper is to benchmark the card and find programming practices that are conducive to higher performance and to compare those techniques and their performance to those of more mainstream processors.
Benchmarking and Programming Practices for the Intel Xeon Phi Coprocessor

by

William Leslie

A MASTERS PROJECT
submitted to
Oregon State University

in partial fulfillment of
the requirements for the
degree of
Master of Science

Presented September 1, 2016
Commencement June 2017
Master of Science project of William Leslie presented on September 1, 2016

APPROVED:

________________________
Mike Bailey, representing Computer Science

________________________
Head of the Department of Computer Science

________________________
Dean of the Graduate School

I understand that my paper will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my paper to any reader upon request.

________________________
William Leslie
ACKNOWLEDGEMENTS

I would like to acknowledge the people who made my education possible. First, I would like to thank my parents. They have supported me wherever life took me, especially in my educational decisions. Next, I would like to thank my girlfriend, Colleen, for standing by me in my life, both personally and academically. I would also like to thank my major advisor, Dr. Mike Bailey. He has stood behind me through my entire Masters degree; his support (and his hardware) made this paper possible. Thirdly, I would like to thank Kevin McGrath, who has been a boss and advisor to me throughout my studies. Kevin is the second member of my Masters defense committee. Finally, I would like to thank Dr. Eugene Zhang, a wonderful teacher and the third member of my committee.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>7</td>
</tr>
<tr>
<td>2 Idealized Performance</td>
<td>10</td>
</tr>
<tr>
<td>3 Literature Review</td>
<td>12</td>
</tr>
<tr>
<td>4 Process</td>
<td>16</td>
</tr>
<tr>
<td>4a Xeon Phi Programming</td>
<td>16</td>
</tr>
<tr>
<td>4b Comparison with Xeon</td>
<td>18</td>
</tr>
<tr>
<td>4c CUDA And GPU Programming</td>
<td>19</td>
</tr>
<tr>
<td>5 Results</td>
<td>20</td>
</tr>
<tr>
<td>6 Analysis of Trends</td>
<td>28</td>
</tr>
<tr>
<td>7 Conclusions</td>
<td>30</td>
</tr>
<tr>
<td>8 Future Work</td>
<td>31</td>
</tr>
<tr>
<td>References</td>
<td>32</td>
</tr>
<tr>
<td>Appendix A</td>
<td>33</td>
</tr>
<tr>
<td>Appendix B</td>
<td>36</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

Moore’s Law states that transistor density doubles approximately every two years. This trend is directly linked to computer processor clock rates. Below is a graph over the years of (from top to bottom) transistors, clock speed in MHz, power consumption in Watts, and performance per clock cycle. One can see that in recent history, Moore’s Law continues its trend in transistors, but the other plots have levelled out.

Computer processors have been plateauing in clock speed for some time now, due to increasing heat dissipation and power requirements. Power consumption is proportional to the square of the clock speed. Below is an image
demonstrating the heat involved with increasing clock rates.

**CPU Architecture Today**

Heat becoming an unmanageable problem

![Graph showing power density in CPU architecture]

**Figure 1.** In CPU architecture today, heat is becoming an unmanageable problem. (Courtesy of Pat Gelsinger, Intel Developer Forum, Spring 2004)

Most would agree that the heat of a rocket nozzle or the surface of the sun is excessive for a computer processor, which would be a reality if clock speeds were to continue increasing at its previous rate. The cooling would have to be equally drastic in order to keep the circuitry from melting. In order to increase computing power without increasing the clock speeds (and thus the problems with doing so), manufacturers have turned to increasing the number of processors or “cores” within one device.

The purpose of this project was benchmarking the “rabbit” server’s Intel Xeon Phi co-processor at Oregon State University. This testing helped determine useful techniques and configurations to harness the card’s full power and potential. The card at Oregon State has 56 multi-purpose programmable cores, four-way hyperthreading and 16 floating-point SIMD lanes per core.

Hypothetically, one may use all these features together to achieve extremely high performance computing.
Each core is significantly less powerful than a single core of a conventional processor; the Xeon Phi is rated a little over 1 GHz per core and does not support out-of-order instruction processing. This makes the processor something between a CPU and a GPU. The programming style is virtually identical to programming a typical x86 processor with C or C++, with the difference that it can support many more threads concurrently, due to the number of cores and the high degree of hyperthreading.

Since the co-processor has such specific architecture, one can hypothesize that there are some specific types of programs that would benefit from it, namely those that can be easily broken into many threads. This is why the tests include array multiplication and an n-body simulation. Due to the parallel nature of this card, there was no testing of serial programs since an x86 CPU would work significantly better for the job.

The hypothesis before any testing was that using the SIMD registers to their full potential and using the maximum number of threads is the best way to utilize this card. If there is any blocking involved in the program, then the number of working threads could be up to 4 times the number of physical cores, since it would take full advantage of the hyperthreading. The assumption is that under the optimal configuration, this card performs significantly better than most alternatives.

Similar tests were run on a 32 core Xeon server processor and an NVIDIA Titan Black to compare the performance. Not only did the project find the optimal configurations for running a Xeon Phi, it tested its viability compared to less niche technology.
2. IDEALIZED PERFORMANCE

The biggest beneficiaries of this research would mostly be those who wish to run similar applications, including scientists or engineers running massive numerical simulations or other calculations done on large data sets.

A good preliminary calculation is the ideal performance of each processor, a completely theoretical expectation based on the hardware’s specifications. Each core of the Xeon Phi runs at about 1.1 GHz and there are 56 programmable cores. Multiplying would give the ideal performance of the Xeon Phi:

\[ 1.1 \text{ GHz/core} \times 56 \text{ cores} \times 1 \text{ floats/cycle} = 61.1 \text{ gigaflops} \]

When compared to the Xeon server that holds the card (32 cores at 2.4 GHz) we have the calculation:

\[ 2.4 \text{ GHz/core} \times 32 \text{ cores} \times 1 \text{ floats/cycle} = 76.8 \text{ gigaflops} \]

The server is winning now, but when one takes into account the large SIMD registers, the calculations look like this:

\[ 1.1 \text{ GHz/core} \times 56 \text{ cores} \times 16 \text{ floats/cycle} = 977.6 \text{ gigaflops} \]

\[ 2.4 \text{ GHz/core} \times 32 \text{ cores} \times 4 \text{ floats/cycle} = 307.2 \text{ gigaflops} \]

As one can see, the Xeon processor is expected to be marginally better without vectorization since the clock rate is much higher and the number of cores is relatively high. With the SIMD registers working, the Phi can do a significantly higher number of operations per second. Of course, these calculations are ideal and assume that 100% of the program is both parallelizable and vectorizable.

Amdahl’s Law on the maximal speedup of a parallelized program is as follows:

\[ S = \frac{N}{(1 - p) \times N + p} \]
Where $S$ is the speedup, $N$ is the number of processors, and $p$ is the fraction of the program that is parallelizable. The speedup only approaches the number of processors as the program becomes completely parallel.

The architecture of a GPU is significantly different, where it has many, much slower “CUDA cores”. The ideal speed of the Titan Black can be calculated similarly:

$$0.889 \text{ GHz/core} \times 2880 \text{ cores} \times 1 \text{ float/cycle} = 2560.3 \text{ gigaflops}$$

It may seem that this is the clear winner, and in many situations it is. Graphical transformations are particularly good on a graphics card, as the name would indicate. Differences in architecture and programming style may not make this the best choice for general purpose programming. For instance, memory cannot be trivially shared between different blocks of threads, like they can on a CPU or a Xeon Phi. The time it takes to synchronize all threads and moving memory across the card’s bus should also be taken into account, since the memory must be initialized on the CPU and brought back over the bus again to be viewable via command line.

<table>
<thead>
<tr>
<th>Device</th>
<th>Clock Rate</th>
<th>Cores</th>
<th>Register Size</th>
<th>Performance</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon Phi</td>
<td>1.1 GHz</td>
<td>56</td>
<td>16 floats</td>
<td>61.1 gflops</td>
<td>977.6 gflops</td>
</tr>
<tr>
<td>Xeon</td>
<td>2.4 GHz</td>
<td>32</td>
<td>4 floats</td>
<td>76.8 gflops</td>
<td>307.2 gflops</td>
</tr>
<tr>
<td>Titan</td>
<td>0.889 GHz</td>
<td>2880</td>
<td>1 float</td>
<td>2560.3 gflops</td>
<td>2560.3 gflops</td>
</tr>
</tbody>
</table>
3. LITERATURE REVIEW

*Intel Xeon Phi Coprocessor High-Performance Programming* by Jim Jeffers and James Reinders is a comprehensive book written by Intel employees. Intel insiders may have some low-level architectural insights that make this a good resource. This book touches on the architecture, performance and programming style of the card. The publication date is 2013, which makes it a fairly recent resource, though some newer configurations of these cards would not be discussed.

Another helpful resource is *Benchmarking the Intel Xeon Phi Coprocessor* put forward by the Infrared Processing and Analysis Center at Caltech. The article was written by F. Masci in 2013. This resource is useful since a party not related to Intel is responsible for testing the card. Unfortunately, the publication year is the same as the first book, so any recent changes will not be in this paper either.

*CUDA by Example: An Introduction to General-Purpose GPU Programming* is a valuable reference for learning to program NVIDIA cards with CUDA. The authors are Jason Sanders and Edward Kandrot, two NVIDIA senior software engineers. This resource is useful since the programming style of GPUs is significantly different from the way a CPU is.

While the introduction of this paper has one calculation of performance, other sources have different metrics and test devices. The book *Intel Xeon Phi Coprocessor High-Performance Programming* states that the theoretical performance of the Xeon Phi is about 2.130 teraflops. This is calculated using a Xeon Phi with 61 cores clocked at 1.091 GHz; this calculation also takes into account a technique called fused multiply-add (FMA) whereby the processor can apply a multiply and add in the same instruction. [Jeffers 2013] This essentially doubles the number operations, assuming there are multiplies and adds together. The calculation is as follows:
1.091GHz*61 cores*16 lanes *2 (FMA) = 2.1296 teraflops

Without the FMA, the calculation is as follows:

1.091GHz*61 cores*16 lanes = 1.0648 teraflops

*Benchmarking the Intel Xeon Phi Coprocessor* has another setup. Their performance is calculated using 60 1.052 GHz cores. [Masci 2013] Thus their performance would be:

1.052GHz*60 cores*16 lanes = 1.010 teraflops or 2.020 teraflops with FMA

Of course, all these numbers are ideal and theoretical, and not attained when running an actual program [Masci 2013]. Graphs comparing performance to an 8 core Xeon processor using these metrics shows the performance climbing faster on the Xeon but leveling out much quicker than on the Xeon Phi. [Jeffers 2013] In practice, when compared to a Xeon with 16 2.6 GHz, the Xeon Phi only has a speedup of about 2 to 2.5 times. Graphs of a program optimized to use the Xeon Phi shows performance leveling out a little under 1000 gigaflops. [Masci 2013]

GNU compilers are not suited for compiling Xeon Phi Code and only Intel Compilers have the necessary flags to efficiently run code on the device. [Masci 2013] The ICC compiler flags are set as follows [Jeffers 2013]:

```
icc -openmp -mmic -vec-report=3 -O3 program.c -o program
```

icc is the Intel c Compiler, openmp tells enables OpenMP for parallelism, vec-report creates a file that discusses the effect of the vectorization, and -O3 is the optimization level; O3 included automated vectorization.

OpenMP and its pragmas are often used for assistance in multithreading when using the Xeon Phi. [Masci 2013, Jeffers 2013] The alternative to using omp is directly using pthreads, which is what OpenMP uses under the surface. In addition to setting the parallelism of the program, OpenMP has the SIMD
pragma which explicitly tells the compiler where in the program the arrays can be vectorized. [Masci 2016]

To make the most use of the 64 byte wide vector lanes, 16 floats or 8 doubles can be operated on simultaneously. This can be achieved by aligning the array on 64 byte cache-line boundaries. There are several ways to align the arrays to cache lines, such as posix-memalign() or malloc_aligned(). [Masci 2013]
Alternatively, the arrays can be setup like this:

```c
float a[SIZE] __attribute__((align(64)));
```

Another pragma that is useful to for programming a Xeon Phi is “ivdep” which allows the compiler to ignore dependences that are not obvious to the programmer. [Jeffers 2013] The program must also have a long runtime to make complete use of so many threads. It is suggested that at least 120 threads (2 times number of cores) should be used to take full advantage of the Xeon Phi’s cores. [Masci 2016]

An alternative to this sort of parallel programming is through the use of GPUs. Since NVIDIA created the available card, CUDA was the chosen programming language. There are more than a few differences between the programming languages:

```c
__global__ void add(int *a, int *b, int *c){
    int ti = blockIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

Above there is an array addition program written in CUDA [Sanders 2011].

BlockIdx.x is the block index, assuming that we will use one thread per block. The program will spawn threads with different block indexes until the Nth one is reached. This is lower level than simply using OpenMP which generates threads based on loop indices.
While threads within blocks share memory, blocks do not, by default, share resources. The `__share__` keyword creates a copy of a variable for every block. [Sanders 2011] Each copy is independent of each other and have the same issues as any shared resource, where race conditions are a problem. One must be careful when resources are written and read.
4. PROCESS
   
   a. XEON PHI PROGRAMMING

The first program to test the Xeon Phi was an n-body gravity simulation. The program keeps track of the bodies’ current positions and velocities; these values are updated with every step based on the distance from other bodies and their masses. This algorithm is based on Newton’s Law of Universal Gravitation:

\[ F = \frac{G m_1 m_2}{r^2} \]

Using this formula, one can calculate the force each body applies on another and then further isolate the acceleration in each dimension. These accelerations are then used to calculate the new positions and velocities for the body and the process repeats. The final version of this code can be found in Appendix B.

At the beginning, a struct for each body seemed like good programming practice. This way, a body consisted of a vector of positional information, a vector for velocity, and a float for the body’s mass. This is intuitive, since an array can be made of bodies; however, this setup is inefficient since the large structure means that alike memory is not contiguous. Memory continuity is especially important for the Xeon Phi SIMD registers.

The original algorithm is called an array of structures and the modification is called a structure of arrays. The way the program’s structure of arrays works is that each variable and vector component has its own array. In conjunction with the 16 float wide lanes, 16 floats can be read and updated simultaneously. This works because the same calculation is applied to each float in the vector.

To achieve maximal performance, the cache alignment was set to 64 bytes, representing 16 floats for the Phi’s extra wide registers, which is a complete cache line. O3 level optimization was chosen to improve performance. As an
added bonus, the compiler attempts to automatically vectorize loops that work over arrays; this is reflected in all performance tests.

At one point in testing, the efficacy of the compiler’s automated vectorization came into question and another run of tests emerged. This test set revolved around explicitly programmed SIMD pragmas using OpenMP. Upon analysis of produced assembly code, one could see a significant size difference in the code, but analysis of the performance was inconclusive due to the similarities in graph shapes and limits.

Multicore parallelism was achieved via OpenMP, which is a good choice since it does not require thread-level programming. A script can easily change the number of threads that OpenMP uses. In order to test the hyperthreading, the number of threads tested exceeded 4 times the number of cores.

Many of the references discuss using the “ivdep” pragma to insure that the compiler doesn’t assume any vector dependencies. This is helpful for the program, since the loop dependencies are taken care of in the program. Using ivdep reduces the logic that must be added to the code in the form of these dependency checks.

Similar steps were taken for the array multiplication program. This program was much simpler than the gravity simulation, but the same process was applied to it. The program should have little to no blocking in it since there was no competition for resources; every element of each array should only be accessed once per run. It was a good candidate to test since hyperthreading should do very little for the program. This is in stark contrast to the gravity program that appeared to heavily use hyperthreading.

The multiplication program is very simple compared to the gravity simulation. It simply creates two arrays and fills them with values, then multiplies the elements of those two arrays together and puts the output into a third array.
The full code for this program is within Appendix B, under the Array Multiplication header.

b. COMPARISON WITH XEON

A decision was made that the Xeon would be used to compare performance. This hardware is more like desktop processors and is what most programmers have experience with. The Xeon is a powerful processor and the Xeon Phi’s efficacy can be better determined by comparison to a more traditional processor.

The n-body gravity simulation was chosen to run on the Xeon processor. This was done for two reasons. First, it is the more complicated program and therefore something that someone would actually run on either device. The second is that the Titan cannot trivially do this type of calculation, since it would require sharing memory between blocks and much more thread cooperation. Only the vectorized version of the program was run for purposes of comparison. The maximal performance was of interest when comparing these two pieces of hardware.

The process for making a program on the Xeon that compares with the Xeon Phi is very straight-forward, since the code was the same. Compiling for the Phi works similarly to most x86 programs. Using the Intel C++ compiler (ICPC), a normal Linux executable was created. Running the program was as one would expect, too. The script runs over the same values as the Xeon Phi one, even the higher thread counts that would be unnecessary on such a device. This is done to illustrate how such a high thread count would affect the performance after the theoretical plateau or peak is reached.
c. CUDA AND GPU PROGRAMMING

The GPU is another processor that was often used for scientific computation. A Titan Black is connected to the same system that houses the Xeon and Xeon Phi. This is by far the most powerful GPU available for the purposes of this paper.

The array multiplication program was chosen to run on the Titan GPU since, as mentioned before, the gravity simulation requires much more thread cooperation that would slow down the performance. This is due to the fact that threads in different blocks cannot easily share memory. Since one would want to make as much use of blocks as possible and an n-body problem, by definition, must take into account a large number of bodies in the system, the gravity simulation was not a good metric for a GPU’s performance because there was a lot more synchronization involved.

The GPU code looks pretty different from the CPU code this paper has discussed so far. The full program can be found in Appendix B under the CUDA Array Multiplication heading. A key portion of the code is the kernel call which looks like this in the program:

add<<<(N/512), 512>>>(dev_a, dev_b, dev_c);

This may seem strange, but the breakdown is like this: the values in the triple brackets (from left to right) are the number of blocks and the number of threads per block, respectively. The dev variables represent the two input arrays and one output array that are currently allocated on the GPU. To test the GPU, the value of N ranged up to as high as possible before the GPU couldn’t process anymore threads. This gives the largest number of calculations without looping over values.
5. RESULTS

Each graph is plotted twice, first using the number of OpenMP threads as the independent variable, with the data set size as the series, and then using the data set size as the independent variable, with the number of threads as the series. Together, all this data constitutes a series of 3D graphs, but was presented like this for convenient viewing on paper. Anything that has a single core does not use OpenMP.
Values for this graph are in Appendix A, under table 1.

In the graphs above, one can see the performance starting to level out after about a quarter million bodies. While larger data set sizes do increase in performance after this point, the slope becomes much less dramatic. One can also see a large drop-off in performance after 224 threads, 4 times the number of cores. The performance gain up to this point would indicate that the hyperthreading is working on this program.
Values for this graph are in Appendix A, under table 2.

From the graph above, one can deduce that performance is virtually identical between programmer-defined vectorization and that which is generated automatically by the compiler; the two curves look very similar and peak around the same point.
Values for this graph are in Appendix A, under table 3.

The scale of these graphs has been divided by 10 on the Y axis to show detail, since the performance is significantly lower. Performance decreases between 10 and 12 times without vectorization. Again, one can see a dramatic drop-off in performance after 224 threads.
Values for this graph are in Appendix A, under table 4.

Here is the same program run on the Xeon CPU. One can see performance starting to cap out around 32 threads, which is the number of Xeon processor cores. Performance improves slightly between 32 and 64 threads, indicating that hyperthreading helps slightly but not as significantly as on the Xeon Phi. The performance does not plummet at higher thread counts.
Unlike the gravity simulation, in the simple array multiplication program performance drops off significantly with more threads, especially after 56 threads. This would indicate that hyperthreading does nothing for the performance with such a simple program.
Here is a nonvectorized version of the multiplication program. One can see these curves are very similar to those of the vectorized program and deduce that vectorization doesn’t affect performance at all. This result is quite surprising, because it seems very easy to vectorize such a loop.

Values for this graph are in Appendix A, under table 6.
Values for this graph are in Appendix A, under table 7.

The Titan Black only has two plots since in this situation the card is always using the maximal number of threads. One plot was calculated using timing that includes the transfer of data over the bus in both directions; this is labeled “with setup”. Performance from the Titan is consistent after an array size of a quarter million. It plateaus around the middle of Xeon Phi graphs. This may be surprising, since the GPU has so many CUDA cores, but one must remember that the performance takes into account transferring the memory across a card bus.

The second plot is without any of the transfer time and is labeled “without”. This performance is significantly higher with a positive trend as array sizes increase. With very large data sets, the Titan is the clear winner, as hypothesized in the introduction. This is the reason GPUs are commonly used for this type of work; although, an issue arises when the arrays have to be sent over the card’s bus back to the main memory. In a program like this, where there is little computation length, the timing portion for transferring data is quite significant.
6. ANALYSIS OF TRENDS

If one were to compare the speedups of different hyperthread configurations, some interesting observations emerge. Taking the first set of graphs, with the values in table 1, one can see with the thread count equaling the number of cores, the performance is 2.47 gigaflops. Two times number of cores gives us 3.61 gigaflops. Four times number of cores gives us 5.25 gigaflops. Below is a table of performance with their relative speedups. The first column represents the threads assigned per core (essentially the amount of hyperthreading), the second column is the recorded performance for that number of threads, and the third is the ratio of performance over that without hyperthreading (i.e. row 1).

<table>
<thead>
<tr>
<th>Threads/Core</th>
<th>Performance</th>
<th>Perf/Perf at 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.47</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>3.61</td>
<td>1.46</td>
</tr>
<tr>
<td>4</td>
<td>5.25</td>
<td>2.13</td>
</tr>
</tbody>
</table>

Here is the same table without vectorization:

<table>
<thead>
<tr>
<th>Threads/Core</th>
<th>Performance</th>
<th>Perf/Perf at 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.202</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.316</td>
<td>1.56</td>
</tr>
<tr>
<td>4</td>
<td>0.445</td>
<td>2.20</td>
</tr>
</tbody>
</table>

These trends are similar to those of a modified “Amdahl’s Law”:

\[ SU = \frac{1 - (1 - p)^n}{p} \]

If one were to substitute \( p \) (which is the fraction of the code that is serial) for the fraction of a core that a hyperthread can keep busy without blocking and \( n \) for the number of hyperthreads per core, then the formula can be graphed like this:
This graph is very similar to the numbers calculated above. One can see that the speedups around .4 busy-fraction are essentially the same as those in the nonvectorized hyperthread speedup table; this would indicate that with the gravity simulation any given thread is in use approximately 40% of the time. Using the .4 busy-fraction, if one doubles the number of hyperthreads to 8, the speedup is 2.45. Quadrupling to 16 gives 2.5. Eight times also gives 2.5, indicating that performance converges to 2.5. As one can see, beyond 4 hyperthreads speedup becomes negligible for a program with moderate blocking. Perhaps doubling to 8-way hyperthreading would be useful, but beyond that there is virtually no speedup.
7. CONCLUSIONS

Performance definitely increases with larger array sizes, as indicated through the graphs. This is expected, since the time the program spends in parallel increases, thus improving the proportion of time running parallel code to serial code.

The gravity simulation greatly benefited from 224 threads, but the multiplication program peaked at 56. This indicates that 224 threads are suggested for any code that blocks, to take advantage of the hyperthreading; otherwise 56 threads are recommended.

O3 optimization with vectorization turned on is the best configuration for SIMD. Since performance is virtually identical between manually set OpenMP and compiler set vectorization, O3 optimization is the simpler solution. As The results show, the performance can increase up to 12 times via the use of O3 vectorization.

To take full advantage of SIMD in the program, one must use a structure-of-arrays technique to ensure that memory is effectively being transferred to the registers. Explicitly set SIMD pragmas do not significantly affect performance. This is another reason O3 optimization is suggested, which makes the vectorization simpler to implement without extra programming.

A definite advantage of the Xeon Phi is the fact that almost no extra programming expertise is needed to program the card. Other than running and compiling, a Xeon Phi program is identical to that of any other C or C++ program. A GPU is another option for such problems, but the programming style has some obvious differences. Threads must be more explicitly programmed on a GPU. Libraries such as OpenMP make multicore parallelizing a program very easy and work as expected with the Xeon Phi.
8. FUTURE WORK

The future of these Xeon Phi chips is going to be in on-board processors. Ideally this research would continue and include benchmarking that processor. A major advantage of such a chip involves the use of RAM on the motherboard (and even virtual memory). This should be a major expansion of memory over that which is on the card. Tasks larger than the memory capacity of the card have to transfer memory over the card’s bus to run thus slowing the performance considerably.

Another change of the next generation Xeon Phi is the addition of out-of-order instruction processing. This would allow the program to look ahead and minimize blocking on threads since they can be kept busy doing other parts of the program out-of-order and significantly reduce the blocking that occurs in the gravity program, as evidenced by the heavy use of hyperthreading. Hypothetically, this should improve performance and reduce the reliance on hyperthreading in the program.
REFERENCES


## Appendix A – Tables

### Table 1:

<table>
<thead>
<tr>
<th>Bodies</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>32000</td>
<td>1 16 28 32 56 64 80 96 112 128 144 160 184 200 224 230 256</td>
</tr>
<tr>
<td>64000</td>
<td>0.05 0.60 0.99 1.04 1.41 1.43 1.46 1.58 1.68 1.62 1.64 1.65 1.66 1.67 1.79 1.57 1.61</td>
</tr>
<tr>
<td>96000</td>
<td>0.05 0.69 1.17 1.22 1.80 1.84 1.85 2.12 2.37 2.32 2.46 2.51 2.57 2.75 2.87 2.39 2.69</td>
</tr>
<tr>
<td>128000</td>
<td>0.05 0.74 1.22 1.21 2.00 2.06 2.12 2.49 2.76 2.90 3.07 2.99 3.24 3.42 3.62 2.76 3.13</td>
</tr>
<tr>
<td>160000</td>
<td>0.05 0.74 1.24 1.29 2.08 1.88 2.46 2.62 2.97 2.86 3.32 3.28 3.50 3.71 4.01 3.05 3.41</td>
</tr>
<tr>
<td>192000</td>
<td>0.05 0.70 1.18 1.28 2.16 2.19 2.32 2.99 3.11 2.96 3.18 3.41 3.74 3.93 4.29 3.23 3.67</td>
</tr>
<tr>
<td>224000</td>
<td>0.05 0.75 1.30 1.46 2.22 1.98 2.67 2.79 3.20 2.95 3.20 3.56 3.97 4.22 4.47 3.31 3.70</td>
</tr>
<tr>
<td>256000</td>
<td>0.05 0.76 1.29 1.43 2.28 2.24 2.49 2.84 3.24 3.21 3.39 3.75 4.02 4.24 4.66 3.39 3.79</td>
</tr>
</tbody>
</table>

### Table 2:

<table>
<thead>
<tr>
<th>Bodies</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>32000</td>
<td>1 16 28 32 56 64 80 96 112 128 144 160 184 200 224 230 256</td>
</tr>
<tr>
<td>64000</td>
<td>0.05 0.63 0.96 0.92 1.36 1.36 1.46 1.60 1.40 1.50 1.34 1.71 1.65 1.68 1.71 1.59 1.76</td>
</tr>
<tr>
<td>96000</td>
<td>0.05 0.71 1.05 1.27 1.77 1.91 1.96 2.14 2.38 2.46 2.25 2.78 2.72 2.84 2.84 2.31 2.59</td>
</tr>
<tr>
<td>128000</td>
<td>0.05 0.65 1.08 1.25 1.92 1.94 2.37 2.57 2.97 2.81 2.83 3.05 3.08 3.45 3.66 2.85 3.16</td>
</tr>
<tr>
<td>160000</td>
<td>0.05 0.74 1.23 1.29 2.09 2.01 2.31 2.85 2.96 2.84 3.02 3.56 3.59 3.71 4.03 3.02 3.30</td>
</tr>
<tr>
<td>192000</td>
<td>0.05 0.75 1.23 1.33 2.18 2.12 2.35 2.71 3.07 2.92 3.21 3.42 3.81 3.98 4.30 3.10 3.54</td>
</tr>
<tr>
<td>224000</td>
<td>0.05 0.69 1.15 1.47 2.32 2.07 2.41 2.80 3.20 3.01 3.32 3.59 3.92 4.14 4.51 3.41 3.62</td>
</tr>
<tr>
<td>256000</td>
<td>0.05 0.71 1.18 1.49 2.28 2.28 2.66 2.90 3.23 3.44 3.79 3.93 4.00 4.17 4.65 3.37 3.70</td>
</tr>
</tbody>
</table>

Note: The tables summarize the performance of various configurations in terms of threads and bodies.
Table 3:

<table>
<thead>
<tr>
<th>Threads</th>
<th>1</th>
<th>16</th>
<th>28</th>
<th>32</th>
<th>56</th>
<th>64</th>
<th>80</th>
<th>96</th>
<th>112</th>
<th>128</th>
<th>144</th>
<th>160</th>
<th>184</th>
<th>200</th>
<th>224</th>
<th>230</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32000</td>
<td>0.24</td>
<td>1.40</td>
<td>2.14</td>
<td>1.78</td>
<td>2.03</td>
<td>1.85</td>
<td>2.15</td>
<td>2.05</td>
<td>1.94</td>
<td>2.12</td>
<td>2.36</td>
<td>2.08</td>
<td>1.94</td>
<td>2.12</td>
<td>2.48</td>
<td>2.08</td>
<td>2.34</td>
</tr>
<tr>
<td>64000</td>
<td>0.27</td>
<td>2.14</td>
<td>3.51</td>
<td>2.67</td>
<td>2.97</td>
<td>3.11</td>
<td>2.97</td>
<td>3.20</td>
<td>3.12</td>
<td>3.21</td>
<td>3.06</td>
<td>3.14</td>
<td>3.10</td>
<td>3.04</td>
<td>3.24</td>
<td>3.09</td>
<td>3.02</td>
</tr>
<tr>
<td>96000</td>
<td>0.27</td>
<td>2.58</td>
<td>3.31</td>
<td>3.53</td>
<td>3.60</td>
<td>3.53</td>
<td>3.61</td>
<td>3.93</td>
<td>3.81</td>
<td>3.56</td>
<td>3.34</td>
<td>3.58</td>
<td>3.45</td>
<td>3.51</td>
<td>3.78</td>
<td>3.73</td>
<td>3.61</td>
</tr>
<tr>
<td>128000</td>
<td>0.27</td>
<td>3.17</td>
<td>3.41</td>
<td>3.36</td>
<td>3.79</td>
<td>3.53</td>
<td>3.94</td>
<td>3.49</td>
<td>3.78</td>
<td>4.15</td>
<td>3.86</td>
<td>3.88</td>
<td>3.74</td>
<td>3.87</td>
<td>3.84</td>
<td>4.04</td>
<td>3.81</td>
</tr>
<tr>
<td>160000</td>
<td>0.27</td>
<td>3.53</td>
<td>3.50</td>
<td>3.71</td>
<td>3.92</td>
<td>3.98</td>
<td>3.91</td>
<td>3.93</td>
<td>3.87</td>
<td>3.97</td>
<td>3.99</td>
<td>4.02</td>
<td>4.01</td>
<td>3.95</td>
<td>4.00</td>
<td>3.95</td>
<td>3.94</td>
</tr>
<tr>
<td>192000</td>
<td>0.28</td>
<td>2.85</td>
<td>3.72</td>
<td>3.62</td>
<td>4.10</td>
<td>4.18</td>
<td>3.91</td>
<td>3.95</td>
<td>3.98</td>
<td>4.06</td>
<td>4.00</td>
<td>4.05</td>
<td>3.99</td>
<td>4.07</td>
<td>4.04</td>
<td>4.09</td>
<td>4.00</td>
</tr>
<tr>
<td>224000</td>
<td>0.28</td>
<td>2.80</td>
<td>3.69</td>
<td>4.22</td>
<td>4.12</td>
<td>4.03</td>
<td>4.06</td>
<td>4.05</td>
<td>4.03</td>
<td>4.06</td>
<td>4.06</td>
<td>4.04</td>
<td>4.08</td>
<td>4.02</td>
<td>4.11</td>
<td>4.07</td>
<td>4.09</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4:

<table>
<thead>
<tr>
<th>Threads</th>
<th>1</th>
<th>16</th>
<th>28</th>
<th>32</th>
<th>56</th>
<th>64</th>
<th>80</th>
<th>96</th>
<th>112</th>
<th>128</th>
<th>144</th>
<th>160</th>
<th>184</th>
<th>200</th>
<th>224</th>
<th>230</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32000</td>
<td>0.28</td>
<td>3.47</td>
<td>3.78</td>
<td>4.31</td>
<td>4.05</td>
<td>3.98</td>
<td>4.00</td>
<td>3.99</td>
<td>4.05</td>
<td>4.10</td>
<td>4.08</td>
<td>4.11</td>
<td>4.09</td>
<td>4.13</td>
<td>4.12</td>
<td>4.13</td>
<td>4.12</td>
</tr>
<tr>
<td>64000</td>
<td>0.28</td>
<td>3.22</td>
<td>3.83</td>
<td>3.77</td>
<td>4.12</td>
<td>3.95</td>
<td>4.03</td>
<td>4.07</td>
<td>4.16</td>
<td>4.16</td>
<td>4.13</td>
<td>4.16</td>
<td>4.16</td>
<td>4.20</td>
<td>4.18</td>
<td>4.13</td>
<td>4.14</td>
</tr>
<tr>
<td>96000</td>
<td>0.28</td>
<td>3.46</td>
<td>3.73</td>
<td>3.87</td>
<td>4.17</td>
<td>4.00</td>
<td>4.10</td>
<td>4.15</td>
<td>4.15</td>
<td>4.16</td>
<td>4.21</td>
<td>4.17</td>
<td>4.23</td>
<td>4.20</td>
<td>4.24</td>
<td>4.19</td>
<td>4.11</td>
</tr>
<tr>
<td>128000</td>
<td>0.28</td>
<td>2.65</td>
<td>3.81</td>
<td>3.88</td>
<td>4.14</td>
<td>4.14</td>
<td>4.14</td>
<td>4.16</td>
<td>4.18</td>
<td>4.21</td>
<td>4.21</td>
<td>4.20</td>
<td>4.21</td>
<td>4.25</td>
<td>4.23</td>
<td>4.27</td>
<td>4.27</td>
</tr>
<tr>
<td>160000</td>
<td>0.28</td>
<td>3.29</td>
<td>3.75</td>
<td>3.85</td>
<td>4.16</td>
<td>4.07</td>
<td>4.13</td>
<td>4.16</td>
<td>4.23</td>
<td>4.23</td>
<td>4.21</td>
<td>4.25</td>
<td>4.24</td>
<td>4.26</td>
<td>4.22</td>
<td>4.27</td>
<td>4.20</td>
</tr>
<tr>
<td>192000</td>
<td>0.28</td>
<td>3.03</td>
<td>3.89</td>
<td>3.93</td>
<td>4.14</td>
<td>4.11</td>
<td>4.17</td>
<td>4.19</td>
<td>4.24</td>
<td>4.20</td>
<td>4.21</td>
<td>4.27</td>
<td>4.23</td>
<td>4.29</td>
<td>4.28</td>
<td>4.30</td>
<td>4.30</td>
</tr>
<tr>
<td>224000</td>
<td>0.28</td>
<td>3.47</td>
<td>3.84</td>
<td>3.87</td>
<td>4.15</td>
<td>4.14</td>
<td>4.18</td>
<td>4.22</td>
<td>4.18</td>
<td>4.22</td>
<td>4.30</td>
<td>4.29</td>
<td>4.23</td>
<td>4.28</td>
<td>4.26</td>
<td>4.27</td>
<td>4.30</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Leslie 34
Table 5:

<table>
<thead>
<tr>
<th>Threads</th>
<th>1</th>
<th>28</th>
<th>32</th>
<th>56</th>
<th>64</th>
<th>96</th>
<th>112</th>
<th>128</th>
<th>184</th>
<th>224</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>25600000</td>
<td>0.16</td>
<td>0.50</td>
<td>0.48</td>
<td>0.28</td>
<td>0.31</td>
<td>0.23</td>
<td>0.18</td>
<td>0.19</td>
<td>0.13</td>
<td>0.11</td>
<td>0.09</td>
</tr>
<tr>
<td>51200000</td>
<td>0.16</td>
<td>0.86</td>
<td>0.78</td>
<td>0.65</td>
<td>0.63</td>
<td>0.45</td>
<td>0.40</td>
<td>0.38</td>
<td>0.26</td>
<td>0.22</td>
<td>0.19</td>
</tr>
<tr>
<td>76800000</td>
<td>0.16</td>
<td>1.19</td>
<td>1.17</td>
<td>0.91</td>
<td>0.86</td>
<td>0.68</td>
<td>0.62</td>
<td>0.52</td>
<td>0.40</td>
<td>0.34</td>
<td>0.28</td>
</tr>
<tr>
<td>10240000</td>
<td>0.16</td>
<td>1.44</td>
<td>1.39</td>
<td>1.17</td>
<td>1.02</td>
<td>0.86</td>
<td>0.75</td>
<td>0.68</td>
<td>0.49</td>
<td>0.42</td>
<td>0.34</td>
</tr>
<tr>
<td>12800000</td>
<td>0.16</td>
<td>1.65</td>
<td>1.61</td>
<td>1.45</td>
<td>1.28</td>
<td>1.04</td>
<td>0.90</td>
<td>0.88</td>
<td>0.66</td>
<td>0.54</td>
<td>0.48</td>
</tr>
<tr>
<td>15360000</td>
<td>0.16</td>
<td>1.78</td>
<td>1.83</td>
<td>1.46</td>
<td>1.47</td>
<td>1.23</td>
<td>1.03</td>
<td>1.00</td>
<td>0.76</td>
<td>0.60</td>
<td>0.55</td>
</tr>
<tr>
<td>17920000</td>
<td>0.16</td>
<td>2.07</td>
<td>2.02</td>
<td>1.88</td>
<td>1.64</td>
<td>1.32</td>
<td>1.22</td>
<td>1.13</td>
<td>0.85</td>
<td>0.74</td>
<td>0.62</td>
</tr>
<tr>
<td>20480000</td>
<td>0.16</td>
<td>2.09</td>
<td>2.10</td>
<td>2.05</td>
<td>1.74</td>
<td>1.58</td>
<td>1.44</td>
<td>1.22</td>
<td>0.92</td>
<td>0.82</td>
<td>0.70</td>
</tr>
<tr>
<td>23040000</td>
<td>0.16</td>
<td>2.18</td>
<td>2.17</td>
<td>2.16</td>
<td>1.92</td>
<td>1.49</td>
<td>1.57</td>
<td>1.20</td>
<td>1.08</td>
<td>0.90</td>
<td>0.81</td>
</tr>
<tr>
<td>25600000</td>
<td>0.16</td>
<td>2.27</td>
<td>2.09</td>
<td>2.24</td>
<td>2.06</td>
<td>1.83</td>
<td>1.67</td>
<td>1.49</td>
<td>1.15</td>
<td>1.00</td>
<td>0.67</td>
</tr>
</tbody>
</table>

Table 6:

<table>
<thead>
<tr>
<th>Threads</th>
<th>1</th>
<th>28</th>
<th>32</th>
<th>56</th>
<th>64</th>
<th>96</th>
<th>112</th>
<th>128</th>
<th>184</th>
<th>224</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>25600000</td>
<td>0.06</td>
<td>0.47</td>
<td>0.43</td>
<td>0.35</td>
<td>0.30</td>
<td>0.23</td>
<td>0.20</td>
<td>0.19</td>
<td>0.14</td>
<td>0.11</td>
<td>0.10</td>
</tr>
<tr>
<td>51200000</td>
<td>0.06</td>
<td>0.80</td>
<td>0.76</td>
<td>0.62</td>
<td>0.55</td>
<td>0.46</td>
<td>0.41</td>
<td>0.37</td>
<td>0.27</td>
<td>0.22</td>
<td>0.19</td>
</tr>
<tr>
<td>76800000</td>
<td>0.06</td>
<td>1.03</td>
<td>1.07</td>
<td>0.93</td>
<td>0.80</td>
<td>0.68</td>
<td>0.57</td>
<td>0.54</td>
<td>0.39</td>
<td>0.33</td>
<td>0.29</td>
</tr>
<tr>
<td>10240000</td>
<td>0.06</td>
<td>1.22</td>
<td>1.20</td>
<td>1.06</td>
<td>1.02</td>
<td>0.87</td>
<td>0.79</td>
<td>0.69</td>
<td>0.47</td>
<td>0.40</td>
<td>0.37</td>
</tr>
<tr>
<td>12800000</td>
<td>0.06</td>
<td>1.40</td>
<td>1.38</td>
<td>1.34</td>
<td>1.19</td>
<td>0.98</td>
<td>0.90</td>
<td>0.54</td>
<td>0.62</td>
<td>0.44</td>
<td>0.40</td>
</tr>
<tr>
<td>15360000</td>
<td>0.06</td>
<td>1.46</td>
<td>1.49</td>
<td>1.48</td>
<td>1.45</td>
<td>1.16</td>
<td>1.06</td>
<td>0.90</td>
<td>0.73</td>
<td>0.60</td>
<td>0.54</td>
</tr>
<tr>
<td>17920000</td>
<td>0.06</td>
<td>1.62</td>
<td>1.60</td>
<td>1.65</td>
<td>1.52</td>
<td>1.39</td>
<td>1.19</td>
<td>1.08</td>
<td>0.83</td>
<td>0.71</td>
<td>0.62</td>
</tr>
<tr>
<td>20480000</td>
<td>0.06</td>
<td>1.64</td>
<td>1.67</td>
<td>1.69</td>
<td>1.58</td>
<td>1.50</td>
<td>1.41</td>
<td>1.16</td>
<td>0.90</td>
<td>0.78</td>
<td>0.69</td>
</tr>
<tr>
<td>23040000</td>
<td>0.06</td>
<td>1.66</td>
<td>1.84</td>
<td>1.93</td>
<td>1.81</td>
<td>1.56</td>
<td>1.50</td>
<td>1.28</td>
<td>1.02</td>
<td>0.87</td>
<td>0.76</td>
</tr>
<tr>
<td>25600000</td>
<td>0.06</td>
<td>1.82</td>
<td>1.86</td>
<td>2.12</td>
<td>1.84</td>
<td>1.70</td>
<td>1.18</td>
<td>1.35</td>
<td>1.09</td>
<td>0.93</td>
<td>0.71</td>
</tr>
</tbody>
</table>

Table 7:

<table>
<thead>
<tr>
<th>Elements</th>
<th>Without</th>
<th>With setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>25600000</td>
<td>1.90E+01</td>
<td>0.80</td>
</tr>
<tr>
<td>38400000</td>
<td>2.61E+04</td>
<td>0.85</td>
</tr>
<tr>
<td>51200000</td>
<td>3.48E+04</td>
<td>0.84</td>
</tr>
<tr>
<td>64000000</td>
<td>4.44E+04</td>
<td>0.84</td>
</tr>
<tr>
<td>76800000</td>
<td>5.45E+04</td>
<td>0.84</td>
</tr>
<tr>
<td>89600000</td>
<td>6.22E+04</td>
<td>0.85</td>
</tr>
<tr>
<td>102400000</td>
<td>6.96E+04</td>
<td>0.85</td>
</tr>
<tr>
<td>115200000</td>
<td>8.18E+04</td>
<td>0.84</td>
</tr>
<tr>
<td>128000000</td>
<td>9.09E+04</td>
<td>0.84</td>
</tr>
</tbody>
</table>
Appendix B – Code
Gravity Simulation:
#include <iostream>
#include <fstream>
#include <omp.h>
#include <math.h>
#include <string.h>
#include <stdio.h>
using namespace std;
//#define BODIES 1000
#define TIMES 2
#define STEP 10000
//#define THREADS 4
#define SEED 123
#define G 0.0000000000667408

float oldxv[BODIES] __attribute__((aligned(64)));  
float oldyv[BODIES] __attribute__((aligned(64)));  
float oldzv[BODIES] __attribute__((aligned(64)));  
float oldxp[BODIES] __attribute__((aligned(64)));  
float oldyp[BODIES] __attribute__((aligned(64)));  
float oldzp[BODIES] __attribute__((aligned(64)));  
float newxv[BODIES] __attribute__((aligned(64)));  
float newyv[BODIES] __attribute__((aligned(64)));  
float newzv[BODIES] __attribute__((aligned(64)));  
float newxp[BODIES] __attribute__((aligned(64)));  
float newyp[BODIES] __attribute__((aligned(64)));  
float newzp[BODIES] __attribute__((aligned(64)));  
float mass[BODIES] __attribute__((aligned(64)));

void fillArray(){
    srand(SEED);
    #pragma omp parallel for
    for (long i = 1; i < BODIES; i++){
        oldxp[i] = ((rand() % 30000) - 15000) / 15000.0;
        oldyv[i] = ((rand() % 30000) - 15000) / 15000.0;
        oldzv[i] = ((rand() % 30000) - 15000) / 1000000.0;
        oldxv[i] = ((rand() % 30000) - 15000) / 1000000.0;
        oldyp[i] = ((rand() % 30000) - 15000) / 1000000.0;
        oldzp[i] = ((rand() % 30000) - 15000) / 1000000.0;
        mass[i] = (rand() % 400) + 100.0;
    }
    oldxp[0] = 0;
    oldyv[0] = 0;
    oldzv[0] = 0;
    oldxv[0] = 0;
    oldyp[0] = 0;
    oldzp[0] = 0;
    mass[0] = 200000.0;
}

inline void copyArray(){
    memcpy(oldxv, newxv, BODIES*sizeof(float));
    memcpy(oldyv, newyv, BODIES*sizeof(float));
    memcpy(oldzv, newzv, BODIES*sizeof(float));
    memcpy(oldxp, newxp, BODIES*sizeof(float));
    memcpy(oldyp, newyp, BODIES*sizeof(float));
    memcpy(oldzp, newzp, BODIES*sizeof(float));
}
inline float findDist(int b1, int b2) {
    return sqrtf((oldxp[b1] - oldxp[b2])*(oldxp[b1] - oldxp[b2]) + (oldyp[b1] - oldyp[b2])*(oldyp[b1] - oldyp[b2]) + (oldzp[b1] - oldzp[b2])*(oldzp[b1] - oldzp[b2]));
}

int main(int argc, char **argv){
    #ifdef PARALLEL
   omp_set_num_threads(THREADS);
    #endif
    double time0 = omp_get_wtime();
    fillArray();
    for (int q = 0; q < TIMES; q++) {
        #ifdef PARALLEL
        #pragma omp parallel for
        #endif
        #pragma ivdep
        for (int i = 0; i < BODIES; i++) {
            float accx = 0;
            float accy = 0;
            float accz = 0;
            #ifdef SIMD
            #pragma omp simd reduction(+:accx) reduction(+:accy) reduction(+:accz)
            #endif
            #pragma ivdep
            for (int j = 0; j < BODIES; j++) {
                if (i != j) {
                    float distance = findDist(i, j);
                    float acc = ((G*mass[j]) / (distance*distance));
                    accx += acc*(oldxp[j] - oldxp[i]) / distance;
                    accy += acc*(oldyp[j] - oldyp[i]) / distance;
                    accz += acc*(oldzp[j] - oldzp[i]) / distance;
                }
            }
            newxp[i] = oldxp[i]+oldxv[i]*STEP + .5*STEP*STEP*accx;
            newyp[i] = oldyp[i]+oldyv[i]*STEP + .5*STEP*STEP*accy;
            newzp[i] = oldzp[i]+oldzv[i]*STEP + .5*STEP*STEP*accz;
            newxv[i] = oldxv[i] + STEP*accx;
            newyv[i] = oldyv[i] + STEP*accy;
            newzv[i] = oldzv[i] + STEP*accz;
        }
        copyArray();
    }
    double time1 = omp_get_wtime();
    cout << BODIES << "", "<"<THREARDS<<", "<(double)TIMES*(double)BODIES*BODIES/(time1-time0);
    return 0;
}
Array Multiplication

```cpp
#include <iostream>
#include <omp.h>
#include <math.h>
#include <stdlib.h>
using namespace std;

#define BODIES 1000
#define THREADS 4

float array1[BODIES] __attribute__((aligned(64)));
float array2[BODIES] __attribute__((aligned(64)));
float array3[BODIES] __attribute__((aligned(64)));

void fillArrays(){
    for(int i=0; i < BODIES;i++){
        array1[i] = -i*1.4;
        array2[i] = i*i;
    }
}

int main(int argc, char **argv){
    #ifdef PARALLEL
    omp_set_num_threads(THREADS);
    #endif

    fillArrays();
    double time0 = omp_get_wtime();
    #ifdef PARALLEL
    #pragma omp parallel for
    #endif
    #pragma ivdep
    for(int i=0; i < BODIES;i++){
        array3[i] = array1[i]*array2[i];
    }
    double time1 = omp_get_wtime();
    cout << BODIES << " , " << THREADS << " , " << (double)BODIES / (time1 - time0);
}
```
CUDA Array Multiplication:

```c
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>

#define RUNS 2000
float a[N], b[N], c[N];

__global__ void add(float *a, float *b, float *c) {
  int tid = blockIdx.x*blockDim.x+threadIdx.x; // handle the data at this index
  if (tid < N)
    c[tid] = a[tid] * b[tid];
}

int main(void) {

  float *dev_a, *dev_b, *dev_c;
  // allocate the memory on the GPU
  cudaMalloc((void**)&dev_a, N * sizeof(float));
  cudaMalloc((void**)&dev_b, N * sizeof(float));
  cudaMalloc((void**)&dev_c, N * sizeof(float));
  // fill the arrays 'a' and 'b' on the CPU
  for (int i = 0; i<N; i++) {
    a[i] = -i*1.4;
    b[i] = i * i;
  }
  cudaMemcpy(dev_a, a, N * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(dev_b, b, N * sizeof(float), cudaMemcpyHostToDevice);
  add<<<(N/512), 512>>>(dev_a, dev_b, dev_c);
  // copy the array 'c' back from the GPU to the CPU
  cudaMemcpy(c, dev_c, N * sizeof(float), cudaMemcpyDeviceToHost);
}
```

```c
// display the results
  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  float time;
  cudaEventElapsedTime(&time, start, stop);
  printf("%d,%f\n", N, RUNS*float(N)/(time/1000));
  // free the memory allocated on the GPU
  cudaFree(dev_a);
  cudaFree(dev_b);
  cudaFree(dev_c);
  return 0;
}
```