Getting the Most Out of Your Graphics Card

Final Report

by
Kyle Von Koepping
Justin Walsh
Robert Wrisley

Prepared to fulfill the requirements for ECE402

Department of Electrical and Computer Engineering
Colorado State University
Fort Collins, Colorado 80523

Report Approved:
Project Advisor
Senior Design Coordinator
ABSTRACT

As the potential for shrinking transistor sizes comes to an end microprocessor manufactures have moved away from Moore’s Law as it applies to transistor densities in a single processor. The direction they are heading in now is increasing the number of processor cores on a single chip. So, Moore’s Law is beginning to be applied to chips instead of individual processors. There are also considerations such as the Memory Wall and the Power Wall that are putting a limit on the performance of modern microprocessor chips, even multi-core chips.

The Graphics Processor Unit (hereafter GPU) offers a unique alternative to multi-core processors. These GPUs are packed into graphics cards and were initially designed to handle the massive quantity of floating point operations required to render a 3-dimensional object on a 2-dimensional screen. Recently new programming languages and language libraries have been developed to take advantage of these architectures and their massive parallelism. Our goal was to develop parallelized versions of applications that will run on a graphics card and to measure the increase in performance. We began with a study of the CUDA extension of the C language, which works with NVIDIA GPUs and can be used to write parallel programs. During this project we successfully developed our own implementation of the Matrix Multiply problem (simply multiplying two matrices together to get a result matrix), and saw performance results on par with those obtained by the NVIDIA developers. We also created our own implementation of triangular matrix multiplication (which multiplies two triangular filled matrices together), successfully implemented optimizations and saw a considerable performance increase over the CPU.

These exercises helped us increase our understanding of the relationship between the CPU and GPU, which is important when creating programs that will port their computations to the GPU and get a result back. We also learned about the strict memory management that must take place and the general development algorithm for these types of programs. We have listed some ideas we have about where this project may go in the future, and we recommend continuing the project based on our limited performance results. What we show is that with a small amount of knowledge and the right type of application there is the potential for major performance gains by utilizing the processing power of the graphics card.

We also examined the marketability and ethical concerns that could come into play as high performance computing continues to develop.
Table of Contents

Title i
Abstract ii
Table of Contents iii
List of Figures and Tables v

I. Introduction 1
   A. The Problem 1
   B. The Project Goals 2
   C. The Graphics Card & CUDA 2

II. Project Management & Phases 3
   A. Project Management and Flow 3
   B. Education Phase 4
   C. Simple Implementations Phase 4
   D. Performance Evaluation Phase 4
   E. Optimization Phase 5
   F. Triangular Matrix Multiply Phase 5

III. Matrix Multiplication 5
   A. Preliminary Application & Results 5
   B. Significant Lessons 7

IV. Triangular Matrix Multiply 9
   A. The Problem 9
   B. The Trick 9
   C. The Algorithm 9
   D. The Results 10

V. Project Continuation 12
   A. Rationale 12
   B. Potential Applications 12

VI. Additional Concerns 14
   A. Manufacturability and Marketability of CUDA Projects 14
   B. Ethical Issues and Concerns 16
VI. Conclusion

APPENDIX A – Abbreviations

APPENDIX B – Budget

Acknowledgements
List of Figures

Figure 1  Graph of Moore’s Law                  1
Figure 2  GPU performance trends compared to CPUs  3
Figure 3  GPU Performance                 6
Figure 4  CPU Performance                 7
Figure 5  GPU and CPU Performance            8
Figure 6  TMM GPU Performance            11
Figure 7  Laundry Without Pipelining        13
Figure 8  Laundry With Pipelining           14
Figure 9  CPU/GPU H.264 Video Encoding Comparison  15

List of Tables

Table 1  Average Run Times and Performance for the GPU and CPU  8
Table 2  TMM Average Run Times and Performance for the GPU and CPU  11
Table 3  Budget  B-1
Chapter I – INTRODUCTION

The Problem

Moore’s law, the infamous trend first observed by Gordon Moore, the co-founder of Intel in 1965, has proven to be extremely important over the past 45 years. The observation is a very accurate trend in computer hardware, which states that the number of transistors placed on an integrated circuit was growing at an exponential rate. To be more specific, the combination of increased performance of transistors themselves and the exponential increase of transistors on a circuit board brings about a double of performance roughly every 18 months.

First accepted as a natural occurrence, Moore’s law has now become an industry standard. It has become a driving force of processor manufacturing companies such as Intel and AMD to produce computational devices that follow this trend. If companies do not follow Moore’s law, they will quickly be weeded out by their competition. However, there is an ultimate limit to Moore’s law: as transistors reach the size of atomic levels
they will be eventually limited by the laws of physics itself. Furthermore, Moore’s law
does not apply to Random Access Memory (RAM) speeds and hard drive seek times,
which sets off a rising gap between processor and memory speeds. Additionally there are
power and heat constraints that limit the total capacity of a processor. These limitations
and the memory and power walls have created a need for a complete redesign of
processing architecture. Recently, manufacturers have been driven to design
computational devices with multiple independent cores. These computational devices
are able to process independent computations at the same time.

The performance gained by the addition of processing units is completely
dependent on the algorithm of a problem itself and how a program is implemented. If a
computation is perfectly parallel, then a computer with two cores, each of which is twice
as slow as a single processor will have the same performance as the single processor.
Programmers are now inclined to design programs that run computations in parallel as
effectively as possible. This change in architecture has had an enormous effect on
application industries and software developers and applications are now developed with
multi-threaded designs to attain maximum performance in multi-processor machines.

The Project Goals

The focus of our project has been to research and write applications that are run
on the GPU instead of the CPU, and to compare their performances. Programmers at
NVIDIA have provided sample applications, such as a matrix multiply program, and
other various computationally extensive applications which have allowed us to get a
glimpse of the power of the GPU. However, the CUDA compiler and GPGPU are both
recent developments in computing technology, which leave much to be discovered and
explored. After the project introduction in Chapter I, Chapter II discusses project
management and phases, and Chapters III and IV take an in-depth look at our processes
and results. Chapter V discusses the future plans for this project, and Chapter VI pertains
to ethical concerns, manufacturing and marketability of CUDA applications. Finally
Chapter VII summarizes our findings and concludes our paper.

The Graphics Card & CUDA

Contrary to popular belief, the traditional single-core CPU and even multi-core
CPUs are not the only processors found in a typical personal computer. The graphics
card is a vital hardware component responsible for outputting images to the monitor. The
graphics card has a graphics processing unit (GPU) that is responsible for rendering 3-
dimensional models to a 2-dimensional display; which is a highly floating point
computational task. To have interactive graphics, especially in gaming, requires the GPU
to be incredibly fast at floating point operations. This is due to the need for each pixel’s
certain characteristics, such as color, brightness, shadows, and reflections, to be
calculated and refreshed 60-120 times per second. Thus, as computer technology and
interactive graphics have evolved, the GPU has also evolved in its graphics specific
architecture to become a floating point processing powerhouse. Below is a comparison of
the ever-rising performances of NVIDIA’s GPUs and Intel’s CPUs between January
2003 and November 2006. It can easily be seen that the GPUs have been advancing at a much greater rate, and will most likely continue to in the near future.

Graphics companies have encouraged general purpose computing on the GPU (GPGPU) by creating compilers and development tools to allow programmers to write general purpose applications to be run on the GPU. NVIDIA, one of the leading graphics companies in the market, has developed CUDA (Compute Unified Device Architecture), and a variation of a C based compiler. CUDA is an extension of the C programming language, and not a completely new language, which allows programmers to develop on the GPU more effectively.

**CHAPTER II - PROJECT MANAGEMENT & PHASES**

**Project Management and Flow**

The project, like almost all others, needed its course of action planned out. We collectively came up with a general timeline for the project, with time allotted for initial debugging, implementing fixes and performance speedups, and the corresponding performance measurements. This process would be continued in an iterative loop until all parallel optimizations were implemented and the desired performance was achieved. Dr. Rajopadhye determined that it would be best if we individually created implementations of the programs we set out to parallelize. After writing our initial code, we compared performance and code methodologies to determine what was making a certain efficient implementation, and what was crippling another.

Resource allocation and management could have played a bigger role in our project than it did, especially if we have had to share the graphics cards with many more research groups or people. Many of the times we did our programming and performance measuring, there were few, if any, other people using the machines. Although there
wasn’t any sort of load leveler program in place to queue jobs, the graphics card would not try to simultaneously run two jobs, implementing an inherent queue system (for a few users). Also, due to the nature of how Dr. Connors set up the systems for CUDA, one was specifically for writing and debugging code, and the other was for compiling and running the programs. This created a separated system where the computer doing the computations would not become slowed down by everyone else editing and saving their files, transferring files in and out of the machine, etc.

Education Phase

The education phase of the project included the first several weeks of the first semester. We had to locate equipment that would run our programs which also took up the first several weeks. During that time we read through the CUDA documentation and also worked through several presentations from a CUDA course at University of Illinois Urbana-Champaign. We did not begin coding right away because the machines were not available, although had we been able to, this ramp-up phase may have been cut a lot shorter – from 8 weeks to 3 or 4 weeks.

Simple Implementations Phase

Once we had machines and had spent a week learning how to do the build for the CUDA SDK (Software Development Kit) we moved on to the Matrix Multiply program. This implementation took two matrices and multiplied them together, allowing each core on the GPU to calculate a separate element for the result matrix. We started by examining why this program was good for this project – high level of data independence (every position in a result matrix can be calculated independent of all other results) and a high number of computations. We then began development, which was assigned to each of us individually rather than as a team. While this furthered our understanding of the CUDA library and also the function of the kernel program used to run the device (GPU), working individually rather than as a group led to some frustrations and delays. We wrapped up the initial implementation of the Matrix Multiply program at the end of the first semester.

Performance Evaluation Phase

Now that we had a working implementation of the Matrix Multiply program that would run on the GPUs, we had to determine what kind of performance we were getting and what sort of baseline to compare it to. We used a sequential version that runs on the CPU that consists of three nested loops: the outer loop iterates over the rows of the matrix, the middle loop loops over the columns of the matrix, and the inner loop does the calculations for each position. This algorithm is not the most efficient available, but gave us an idea of how long the CPU would take to do the Matrix Multiply program and also gave us the ability to compute a reference matrix to compare with our GPU results and ensure mathematical correctness.
Optimization Phase

Next, we optimized our Graphics implementation by tiling: a process where the result being calculated is broken down into blocks, and each of these sub-blocks computes its individual result and adds it to the total result. By doing this we were able to tweak our block size so that we were not going beyond the number of threads that could be executed together at once. This also allowed us to take advantage of the shared memory that each GPU has, thereby reducing memory latency times. Each sub-block could load into its shared memory the corresponding sub-blocks of the source matrices, and thus run all of its computations without further accessing the global memory on the device.

Triangular Matrix Multiply Phase

Once we were finally able to complete and successfully execute a tiled version of the Matrix Multiply program, we were challenged to try and develop a version for Triangular Matrix Multiply: a more complicated but still highly parallelizable problem. Again we were tasked to each tackle this problem individually instead of as a group. This approach again led to delays and slower development, and we would suggest that if the Matrix Multiply program absolutely must be done individually a next program such as this be done as a group. The biggest reason is that developing as a group will foster teamwork and also allows group members to share their strengths in understanding and reduce the opportunities for mistakes and frustration. Towards the very end of the project we began to work together on this problem and together we developed the sequential algorithm on which our work is based.

Chapter III – MATRIX MULTIPLICATION

Preliminary Application & Results

Once we began programming with CUDA, our focus was on writing and analyzing a matrix multiply program. The matrix multiply program is a pleasantly parallel problem, with the computation of each element of the result matrix being independent of all other computations. Being a pleasantly parallel problem, matrix multiply was a good starting point as it allowed us to program using CUDA without much trouble. Also, the matrix multiply program has a set number of calculations, calculated from the matrix sizes, which makes it easy to measure the performance of the program.

Matrix multiply is essentially multiplying an M*N matrix, A, by an N*P matrix, B, to yield an M*P matrix, C. This multiplication entails multiplying each element in row i of matrix A by each element of row j of matrix B, then summing each of those products to yield the i,j-th element of matrix C. In our project, we set M = N = P in order better take advantage of the hardware structure of the GPU.
The basic matrix multiply algorithm that we initially wrote consisted of three nested loops: the outermost loop, i, traverses the rows of the A and C matrices, the middle loop, j, traverses the columns of the B and C matrices, and the innermost loop, k, iterates through each element of the i-th row of A and the j-th column of B. Outside of each iteration of the k-loop, a sum variable is initialized to 0. Inside the k-loop, the sum variable is used to store the sum of the products of the i,k-th element of the A matrix and the k,j-th element of the B matrix. After each iteration of the k-loop, the value of the sum variable is written to the i,j-th element of the C matrix. Each iteration of the inner loop contains two floating point operations, a multiply of the float-type values of the two matrices and then an addition of the sum to the current value of the result matrix. Therefore, if square matrices of size N x N are used, the number of computations will be $2*N^3$. From this, we can calculate the performance, which will be $\frac{2*N^3}{T}$ Gigaflops/sec, where T is the run time.

From the programs we wrote to solve the matrix multiply program, we got surprising results that highlight the capabilities of CUDA. As seen in Figure 3 above, we achieved a peak GPU performance of 131 Gigaflops/s. Although this is only 14% of the 933 Gflops/s peak potential of the GTX280, it is still 13 times the peak potential for current quad-core processors. Figure 4 below shows the performance of the CPU in terms of number of calculations, from which it can be seen that the CPU performs considerably worse as the matrix sizes increase. Table 1 below shows the average run times and performances of the GTX280 and the CPU over a variety of different values for N.
A noted optimization for the matrix multiplication program was to tile the result utilizing shared memory into smaller sub-matrices and has each thread block compute a sub-matrix. By tiling the matrices to be multiplied, each thread block can load each tile of data from the first matrix once, and load the tiles of the second matrix $N$/tile-size times into shared memory, considerably reducing the number of memory loads from the naïve triple-nested loop implementation of the matrix multiply algorithm. From this optimization, it can be determined that reducing the amount of data that needs to be loaded into the registers will result in better performance from the GPU.

**Significant Lessons**

There are two important lessons we learned about the memory when dealing with CUDA. First, the main program that is being executed on the CPU must allocate memory on the GPU. This is important since it is required to be done before executing the kernel on the GPU, which is different from regular ANSI C, in which you can allocate memory at any time. Also, CUDA is an extension of ANSI C, not a separate language. At first glance, CUDA appears very intimidating, but once this observation is noted, it becomes easier to grasp. Having experience with programming in C, learning CUDA is about understanding how to use its functions to parallelize existing programs written in C. This requires an understanding of how the GPU hardware works in addition to syntactical concerns.
Table 1: Average Run Times and Performance for the GPU and CPU

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Number of Operations</th>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>8,192</td>
<td>0.0550</td>
<td>0.1490</td>
</tr>
<tr>
<td>32</td>
<td>65,536</td>
<td>0.0613</td>
<td>1.0686</td>
</tr>
<tr>
<td>48</td>
<td>221,184</td>
<td>0.0710</td>
<td>3.1157</td>
</tr>
<tr>
<td>64</td>
<td>534,388</td>
<td>0.0857</td>
<td>6.1203</td>
</tr>
<tr>
<td>80</td>
<td>1,024,000</td>
<td>0.1010</td>
<td>10.1386</td>
</tr>
<tr>
<td>96</td>
<td>1,769,472</td>
<td>0.1253</td>
<td>14.1183</td>
</tr>
<tr>
<td>112</td>
<td>2,809,856</td>
<td>0.1480</td>
<td>18.9861</td>
</tr>
<tr>
<td>160</td>
<td>8,192,000</td>
<td>0.2570</td>
<td>31.8758</td>
</tr>
<tr>
<td>240</td>
<td>27,648,000</td>
<td>0.5710</td>
<td>48.4204</td>
</tr>
<tr>
<td>320</td>
<td>65,536,000</td>
<td>1.0577</td>
<td>61.9629</td>
</tr>
<tr>
<td>400</td>
<td>128,000,000</td>
<td>1.7380</td>
<td>73.6499</td>
</tr>
<tr>
<td>640</td>
<td>524,288,000</td>
<td>5.0807</td>
<td>103.1941</td>
</tr>
<tr>
<td>960</td>
<td>1,769,472,000</td>
<td>13.6847</td>
<td>129.3033</td>
</tr>
<tr>
<td>1024</td>
<td>2,147,483,648</td>
<td>16.2830</td>
<td>131.8850</td>
</tr>
</tbody>
</table>

Figure 5: GPU and CPU Performance

GPU and CPU Performance (Gigaflops/s)
CHAPTER IV - TRIANGULAR MATRIX MULTIPLY

The Problem

Similar to matrix multiplication is triangular matrix multiplication. However, there are also a lot of distinctions between the two. Part of the problem with triangular matrix multiplication is that just creating two separate source matrices that are triangular in nature and running matrix multiply on them will give a correct result, but is not optimal. Because so much of each source matrix (roughly half) would contain no entries (or zero if you prefer), the blocks calculating those sub-matrices with all zeros would be computing a result that didn’t need to be done. Basically those computations add up to wasted utilization. Our goal was to eliminate this problem or at least greatly reduce it.

Since those sub-blocks with all zeros didn’t even need to be computed they were also taking up space in memory on both the host computer and graphics device. They were also compounding the total number of computations that had to be done without contributing anything to the result and thus reducing our performance potential.

The Trick

Out of this we developed a performance enhancement for triangular matrix multiply - we could use a single source matrix to pass from the host computer to the graphics card device, thus cutting the device access latency in half compared to using two separate matrices for each triangular half. However, this leaves us with one full matrix, not two triangular matrices.

The Algorithm

So, with one source matrix we needed to break it into two triangular matrices, at least conceptually. We used LU Decomposition (Lower-Upper Decomposition) to see how our single matrix could be representing two separate triangular matrices. We then developed our algorithm based on calculating a correct result. We are not actually decomposing the source matrix into two triangular matrices. Rather, we are using our analysis using the LU Decomposition to determine the indexing.

In the LU Decomposition the lower triangular matrix has a diagonal with all elements equal to 1, and the upper triangular matrix has the original source matrix diagonal. So, when we calculate any element in the lower triangle, it will have no single terms added since the last term will be a multiplication of two non-zero elements. However, for any element that resides in the upper triangle or on the diagonal, those elements will have their last non-zero multiplication done by one. So for these elements we simply add the single term that resides on the diagonal in the upper triangular matrix.
By using this approach we came up with the following sequential algorithm. Here i represents the row number (so i=0 is the first row, i=1 is the second row, etc.) and j represents the column number (j=0 is the first column, j=1 is the second column, etc.).

The first row is copied from source to result, because no multiplications or additions are necessary for this row.

\[
\text{for}(j=0 \text{ to } n-1) \\
C[i,j]=A[0,j]
\]

For the remaining rows, we need to do a different number of calculations depending on whether the result is in the upper or lower portion (triangle) of the matrix.

Start looping over each row.
\[
\text{for}(i=1 \text{ to } n-1)\{
\]
Do the calculations for those elements in the lower triangle (everything up to but before the diagonal.
\[
\text{for}(j = 0 \text{ to } i-1)\{
C[i,j] = 0; \quad \text{-- Note that no additional single term is added.} \\
\text{for}(k = 0 \text{ to } j)\{
C[i,j] += A[i,k]A[k,j];
\}
\}
\quad \text{-- end of lower triangle computations}
\]
Now do the calculations for those elements in the upper triangle (everything from the diagonal to the end of the row, inclusive).
\[
\text{for}(j = i \text{ to } n-1)\{
C[i,j] = A[i,j]; \quad \text{-- Note the diagonal term is added without being multiplied.} \\
\text{for}(k = 0 \text{ to } i-1)\{
C[i,j] += A[i,k]A[k,j];
\}
\}
\quad \text{-- end of upper triangle computations}
\}
\quad \text{-- end of algorithm}
\]

**The Results**

The following figure (Figure 6) and table (Table 2) show our results for the triangular matrix multiply program on the GPU. At this time our sequential version that is used to check the accuracy of the result matrix is not optimized, so doing a comparison would be inaccurate. We have included the CPU results to demonstrate the extreme increase in performance though. It is worth noting that the CPU at full use would be running approximately 10 Gigaflops/second, and our triangular matrix multiply that runs on the GPU is coming in higher than that once the matrix size is greater than 240 x 240
elements. Also, at the time of this paper our version is not utilizing shared memory. We are continuing to work on getting a shared memory version working, which should provide us with an additional performance boost.

![GPU Performance vs Matrix Size](image)

**Figure 6: TMM GPU Performance**

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Number of Operations</th>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time [ms]</td>
<td>Gigaflops/s</td>
</tr>
<tr>
<td>16</td>
<td>2731</td>
<td>0.000036</td>
<td>0.07585</td>
</tr>
<tr>
<td>32</td>
<td>21845</td>
<td>0.000044</td>
<td>0.49649</td>
</tr>
<tr>
<td>48</td>
<td>73728</td>
<td>0.000059</td>
<td>1.24963</td>
</tr>
<tr>
<td>64</td>
<td>178129</td>
<td>0.000080</td>
<td>2.24055</td>
</tr>
<tr>
<td>80</td>
<td>341,333</td>
<td>0.000097</td>
<td>3.51890</td>
</tr>
<tr>
<td>96</td>
<td>589,824</td>
<td>0.000123</td>
<td>4.79532</td>
</tr>
<tr>
<td>112</td>
<td>936,619</td>
<td>0.000153</td>
<td>6.12169</td>
</tr>
<tr>
<td>160</td>
<td>2,730,667</td>
<td>0.000292</td>
<td>9.35160</td>
</tr>
<tr>
<td>240</td>
<td>9,216,667</td>
<td>0.000713</td>
<td>12.95166</td>
</tr>
<tr>
<td>320</td>
<td>21,845,333</td>
<td>0.001411</td>
<td>15.48216</td>
</tr>
<tr>
<td>400</td>
<td>42,666,667</td>
<td>0.002517</td>
<td>16.95140</td>
</tr>
<tr>
<td>640</td>
<td>174,763,333</td>
<td>0.008972</td>
<td>19.47867</td>
</tr>
<tr>
<td>960</td>
<td>589,824,000</td>
<td>0.027166</td>
<td>21.71185</td>
</tr>
<tr>
<td>1024</td>
<td>715,828,000</td>
<td>0.044282</td>
<td>16.16521</td>
</tr>
</tbody>
</table>

*Table 2: TMM Average Run Times and Performance for the GPU and CPU*
So we can say even without the sequential CPU results that the CUDA version of triangular matrix multiply definitely outperforms the sequential version running on a CPU. It is our contention that the drop in performance for matrix sizes above 1024 x 1024 would be brought up and make a smooth, still increasing curve with the shared memory version of triangular matrix multiply.

Chapter V – PROJECT CONTINUATION

Rationale

High performance computing is not new, but the technologies available continue to be reinvented and reevaluated. As we have mentioned the advent of the GPU, and more specifically programming languages that allow the user to access the computing power of multiple GPUs, has created a new branch on the high-performance tree. It has been shown that programs that meet a set of criteria are ideal for this kind of parallel processing and that the performance gains are phenomenal. However, there is a lot of room for additional research and there are many software programs that meet the criteria but have not been adapted in this way.

That is to say that there are many directions this project could take in the future. Our recommendation is that the project be continued, but the students taking this on need to be aware of the challenges that they will be presented with early on. First of all, the concepts supporting parallel programming are not simple and the course offered by the Colorado State University Computer Science Department should be considered mandatory for project members. Second, team members should plan on spending a couple of weeks familiarizing themselves with the terminology in whatever code source they plan to use to work with the GPUs (CUDA isn’t the only one out there). Finally, team members should expect to spend about twice as much time on this project to be successful as they might have to spend on other projects. There is a lot of ramp up knowledge that needs to be covered before development on a completely new program can be started.

Potential Applications

One potential pleasantly parallel application is one which we have studied in CS475 (Parallel Programming in C) and parallelized using OpenMP - the calculation of a Mandelbrot set. A Mandelbrot set is the calculation of a series of points in the complex plane, using the equation $z_{k+1} = z_k^2 + c$, where $c$ is the position of a point in the complex plane (of format $a + bi$), and $z_k$ is the $k^{th}$ iteration of the function. Each point is iterated until either $|z| > 2$ (the point will go to infinity, and does not belong to the set), or a pre-determined upper limit on the number of iterations is reached. The computation and iterations of each point in the Mandelbrot set are completely independent from one another and can be calculated simultaneously.
The more difficult technique needed to parallelize certain applications, pipelining, may also be implemented in the following years. This type of program involves certain dependencies between iterations of a kernel, and thus cannot be pleasantly parallelized. A real world analogy for this would be doing more than one load of laundry at the same time.

If you needed to do four loads of laundry, and you didn’t implement any pipelining, it would literally take all night to finish. As seen in figure 4 below, four loads are done sequentially, with washing, drying and folding taking 30, 40 and 20 minutes, respectively. Each load would take 1.5 hours, start to finish, it would take 6 hours to complete all four loads. The obvious (and very natural) thing to do would be to pipeline the process.

Implementing pipelining in this case is extremely trivial. Even though only one load can be in the washer at a time, the previously washed load can be in the dryer. Similarly, the load before that can be folded. This entire process depends on a previous stage being done before the current one can be started, but multiple processes that are in different stages can all be done simultaneously.

By having more than one task being done at the same time, since each uses a different piece of equipment, the time for four loads of laundry can be reduced from 6 hours to 4 hours and 10 minutes. This saves 36% of your time, which can be spent taking care of other chores that need to be done. This fundamental time saving technique is exceptionally useful, and can be applied to hardware and many applications we are looking to parallelize.

Another application that might be interesting for future consideration is the computationally intensive task of ray tracing. This application takes a scene of objects, extends rays from a given viewpoint toward the objects and calculates what the image at every point will look like due to light sources, the object’s own reflective properties and other objects reflecting off of or casting shadows onto it. These computations are iterated.
until either a maximum number of reflections are reached or a ray travels a certain distance without intersection. This causes the ray to stop, and the pixel’s value is then updated.

A very diverse application that we also looked at for parallelization is that of processing Hidden Markov models. The likelihood of a future state in these types of systems is dependent only on the present state, and not on any past states. The parameters of the given model are first used to compute the probability of a certain output sequence, from which the probabilities of the unknown variables (usually with specific interest in the most or least probable sets) are then calculated. Hidden Markov models can be applied to a wide variety of complex applications, such as cryptanalysis, biological sequencing, gene prediction and protein folding.

![Figure 8: Laundry with pipelining – much faster.](http://www.ece.arizona.edu/~ece462/Lec03-pipe/)

**CHAPTER VI – ADDITIONAL CONCERNS**

**Manufacturability and Marketability of CUDA Projects**

The market for CUDA enabled applications, even now in CUDA’s infancy, is relatively widespread and substantially large. From our research, we have determined that almost any computational application that has a large dataset which can be broken down into a number of independent computations can benefit from a CUDA implementation. This, if employed correctly, has the potential to accelerate non-graphical applications in many different fields by an order of magnitude or more.
CUDA, despite being so new to the parallel computing market (with its SDK released only in February 2007), has already made a considerable impression on application development and parallelization. A software company called Elemental has developed a very extensive package of video encoders (RapiHD, for Adobe Premiere Pro) and converters (Badaboom, for general media conversion) that utilize CUDA-enabled graphics cards for their processing power. They have worked to make their suite as efficient as possible for encoding various file formats into the popular H.264 format, and have already seen considerable gains. Their method uses a tradeoff between CPU and GPU computation, with utilization of the CPU generally being around 30%.

For a benchmark comparison between top-of-the-line CPUs and GPUs, they’ve compared the Intel Core 2 Extreme QX9770 (~$1500 at time of writing) and the NVIDIA GTX 280 (~$300). With the same 2 minute test video file, AnandTech.com found the following results:

![Figure 9: CPU/GPU H.264 Video Encoding Comparison](http://www.anandtech.com/video/showdoc.aspx?id=3339)

As seen above, the GTX 280 performs the encoding operation in around 8 seconds, with the QX9770 coming the closest with an encoding time of 14 seconds. This shows that the GPU outperformed the CPU by 40% to 88%. Granted, for every application’s needs and particular hardware setup, this number will vary.

The reason that these advanced programs perform so well is due to the fact that they take advantage of smaller, very highly optimized programs. All of these small programs (commonly called kernels) are ones that contain independent data sets, and benefit from optimizations such as loop interchanging and loop unrolling for memory locality, and computation tiling to take advantage of on-chip caches and full utilization of memory bandwidth. These kernels that have been highly optimized, for CUDA and
previously, have been Matrix Multiplication, Discrete/Fast Fourier Transforms, dense Linear Algebra computations, Prime Number Finding, and Encryption/Decryption Algorithms.

There are a decent number of distributed computing applications, such the Berkeley Open Infrastructure for Network Computing (BOINC), which are already up and running to process computationally large problems. These efforts work on problems in a number of different areas, such as Biology, Earth Sciences, Physics, Astronomy and Mathematics. Each works by distributing computation sets to willing participants, and utilizes their idle CPU cycles for processing. Some of the applications that have been completed or are currently in progress are: studies on how diseases spread through the body, models for protein folding, climate and weather prediction, calculating the impact hazards of certain near-Earth objects, cures for AIDS, Cancer and Hepatitis C, and studying and determining the magnetization of material in nanotechnology.

The current implementations of the popular BOINC distributed computing platform utilizes idle CPU cycles from about 1.65 million participants with 3.95 million computers to achieve a computational equivalency of 1.5 PetaFLOPS. This is extremely significant to the scientific and academic communities, because it outperforms all of the current single manufacturer supercomputers today, including IBM’s Blue Gene/L (0.4782 PetaFLOPS in 2007) and the world’s first sustained 1.0 PetaFLOP computer, IBM’s Roadrunner (1.026 PetaFLOPS in May 2008).

Quite possibly the next paradigm shift to occur in computational power would be that of creating, or converting an existing distributed computing project to, a CUDA-enabled GPU system. This would take advantage of a second, fine grained level of computational parallelism (on the client side) to be able to break the data chunks down further for efficient use on the graphics card. This would also allow for the use of larger data chunks per client, and would not only be more efficient for each client involved, but for the project as a whole, and would lead to greatly increased overall processing power.

The implementation of such a system would most likely start out as a heterogeneous system of the current CPU-based and newly designed GPU-based. This would be to facilitate not only the introduction of GPUs to the system, but also to utilize both the CPU and GPU(s) of a system for the project simultaneously. An early realization of this computational power is in the conception of a rough reworking of the BOINC SETI@home platform to include support for CUDA-enabled devices. At current estimates for most systems, it is running \textbf{2-10 times faster than the CPU-only version} (http://setiathome.berkeley.edu/cuda.php) and, naturally, is strongly encouraged by the developers for CUDA-enabled participants to update and take advantage of their idle GPU cycles.

**Ethical Issues and Concerns**

Although the work we have done in this project do not present any major ethical issues, there are a few concerns that must be taken into account. The first is software
licensing issues that could be encountered if more progress is made with the project and software is released. If the project moves in the direction of parallelizing existing applications, licensing and proprietary issues must be considering, as the group would need to follow the software licenses of those applications. The other ethical question that must be addressed is the potential that the work done in this project could be used with ill intent. The furthering of knowledge in the field of programming on GPUs leads to greater accessibility to a personal, affordable “supercomputer.” The performance capabilities of GPUs can drastically aid in cryptology, and with it hacking. Having the processing power of a GPU can help malicious parties crack encryptions and hack into bank, government, and other networks.

Chapter VII – CONCLUSION

It has become clear over the past few years that an entirely new frontier in computing is opening up around us. Current theories have predicted the end of Moore’s Law, or perhaps a reinventing of Moore’s Law applied to core densities on a single chip rather than transistor densities on a single processor. The desire for continually increasing performance, the physical limitations on the sizes of transistors as well as problems encountered with the memory and power walls have led to the need for more parallelized architectures in the next and future generations of computers. Because this requirement is relatively new there is still much to be researched as different architectures are explored and new techniques created for meeting these increasing performance standards.

One potential candidate for exploitation is the GPU, which provides a ready-made parallel architecture capable of performing massive quantities of parallel computations in a fraction of the time of conventional CPUs. Our research so far has been focused on learning one of the several programming language constructs that allows for development of applications that are not graphical in nature but can take advantage of what the GPU is offering. To begin we started with a series of lecture slides posted for a class at the University of Illinois, Urbana-Champaign and also requested several academic papers to determine what research was being conducted already. In order to achieve a relatively quick learning curve we started developing the matrix multiplication program in order to allow the entire team to share results and some of the lessons we were learning. The complexity took us longer than we anticipated. We finally moved on to Triangular Matrix Multiply and achieved some success implementing a non-tiled version but have yet to get our tiled version working correctly. However, the initial results that we are demonstrating indicate that the performance gains are potentially massive for programs that can be parallelized and ported to the GPU for data manipulation.

This project can take several directions in the future, but one thing is for certain… high performance computing is a growing field and more research and continuing of this project are certainly warranted.
APPENDIX A – Abbreviations

CPU – Central Processing Unit
GPU – Graphics Processing Unit
RAM – Random Access Memory
CUDA – Compute Unified Device Architecture
GPGPU – General Purpose computing on the Graphics Processor Unit
GTX 280 – A particular NVIDIA graphics card
CS475 – Computer Science 475 – Parallel Programming Course at CSU
CSU – Colorado State University
SDK – Software Development Kit
LU Decomposition – Lower-Upper Decomposition (Triangular Matrix Decomposition)
APPENDIX B – Budget

<table>
<thead>
<tr>
<th>Date</th>
<th>Action/Fund</th>
<th>ECE Sr. Dsgn. Fund</th>
<th>Other Fund</th>
<th>Donations</th>
<th>Personal Expense</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>25-Aug-08</td>
<td>ECE Allowance</td>
<td>$150.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29-Sep-08</td>
<td>Graphics card purchase</td>
<td></td>
<td></td>
<td>($48.24)</td>
<td></td>
<td>Rob</td>
</tr>
<tr>
<td>30-Oct-08</td>
<td>Reimbursement</td>
<td></td>
<td></td>
<td>$48.24</td>
<td></td>
<td>Rob</td>
</tr>
<tr>
<td>20-Jan-09</td>
<td>ECE Allowance</td>
<td>$150.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3-May-09</td>
<td>Balance</td>
<td>$251.76</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Budget
ACKNOWLEDGEMENTS

We’d like to thank Professors Sanjay Rajopadhye and Olivera Notaros, our faculty advisors on this project, for their support and guidance from the initial inception of the project and throughout the year.

We’d also like to thank Professor Dan Connors for the use of his machines with NVIDIA GTX 280 graphics cards on the University of Colorado campus. He has provided us with free access to machines that provide a stable and consistent environment for benchmarking our results.