A Performance Comparison of a Naïve Algorithm to Solve the Party Problem using GPUs

Michael V.E. Bryant
Merrimack College
315 Turnpike Street
North Andover, MA 01845
michael.bryant@merrimack.edu

David Toth
University of Mary Washington
1301 College Avenue
Fredericksburg, VA 22401
dtoth@umw.edu

ABSTRACT

The R(m, n) instance of the party problem asks how many people must attend a party to guarantee that at the party, there is a group of *m* people who all know each other or a group of *n* people who are all complete strangers. GPUs have been shown to significantly decrease the running time of some mathematical and scientific applications that have embarrassingly parallel portions. A brute force algorithm to solve the R(5, 5) instance of the party problem can be parallelized to run on a number of processing cores many orders of magnitude greater than the number of cores in the fastest supercomputer today. Therefore, we believed that this currently unsolved problem is so computationally intensive that GPUs could significantly reduce the time needed to solve it. In this work, we compare the running time of a naïve algorithm to help make progress solving the R(5, 5) instance of the party problem on a CPU and on five different GPUs ranging from low-end consumer GPUs to a high-end GPU. Using just the GPUs computational capabilities, we observed speedups ranging from 1.9 to over 21 in comparison to our quad-core CPU system.

General Terms

Parallel Programming, GPGPU, Ramsey Theory

Keywords

Blue Waters Undergraduate Petascale Internship, CUDA, Party Problem, Performance Comparison

1. INTRODUCTION

Ramsey Theory is an area of mathematics concerned with "the mathematical study of combinatorial objects in which a certain degree of order must occur as the scale of the objects becomes large" [1]. Applications of Ramsey Theory include computational geometry, information theory, complexity, and parallelism [2]. The party problem is a problem in Ramsey Theory. The R(m, n) instance of the party problem asks how many people must be invited to a party (assuming that all the invitees will attend) to guarantee that at the party, there is a group of *m* people who all know each other or a group of *n* people who are all complete strangers. Thus, the solution to the R(5, 5) instance of the party problem indicates the fewest number of people required to attend a party to guarantee that at the party, there will be a group of 5 people who all know each other or a group of 5 people who are all complete strangers. While the party problem has been solved for some small values of *m* and *n*, it has yet to be solved for values of m and n that are both equal and at least 5 [3]. For several of these cases, the bounds on the answers to the problem have been established [3]. For example, it is known that $43 \leq R(5, 5) \leq 49$ [3].

To help visualize the party problem, 2-colored graphs are typically used to show the relationship between the people as either acquaintances or strangers. The graphs consist of vertices representing the people and edges that connect every vertex to every other vertex, forming a complete graph. A red edge connecting two vertices indicates that the vertices represent people who are strangers and a blue edge connecting two vertices indicates that the vertices represent people who are acquaintances. The graphs that are used to help visualize the problem can also be used to solve the problem. To solve R(m, n), one must find the smallest number of people *x* such that every complete graph on *x* vertices where the edges are colored blue and red contains a subgraph that is a red or blue complete graph on 5 vertices. Since the number of edges in a complete graph on *v* vertices is $(v^2-v)/2$ and each edge may be colored red or blue, there are $2^{((v^\wedge 2)-v)/2}$ complete graphs on v vertices. Therefore, to increase the lower bound on R(5, 5) from the known lower bound of 43 to 44, one would need to demonstrate that at least one of the $2^{((43^\wedge 2)-43)/2}$ or 2^{903} complete graphs on 43 vertices does not contain a subgraph that is a red or blue complete graph on 5 vertices, denoted K_5. Alternatively, one could decrease the upper bound from the known value of 49 to *u* by demonstrating that every complete graph on *u* vertices where the edges are colored blue and red contains a subgraph that is a red or blue K_5. We do note that many of the 2^{903} graphs that are isomorphic to each other, and thus testing one of those graphs eliminates the need to test the other graphs that are isomorphic to it. Unfortunately, it is often slower to test if a graph is isomorphic to another graph that has already been tested than to test if the graph contains a red or blue K_5. We also note that for every graph, another graph in the 2^{903} graphs can be obtained and if the original graph contains a red K_5, then the graph obtained by reversing the edge colors must contain a blue K_5. Therefore, only half of the 2^{903} graphs need to be tested and this can be accomplished relatively easily.

We chose to write our algorithm to attempt to increase the lower bound from the known value of 43 to 46 because of a conjecture that R(5, 5) = 46 [4]. In the computationally worst case scenario, our algorithm would test every graph and determine either that R(5, 5) > 45 if all the graphs other than the last one contained a red or blue K_5 or that R(5, 5) ≤ 46 if every complete graph on 45 vertices with red and blue edges contained a red or blue K_5. If we instead tried to decrease the upper bound on R(5, 5) by one, the worst case scenario would involve testing more graphs.

2. RELATED WORK

Over the past several years, many researchers have begun to use GPUs to do a wide variety of mathematical and scientific calculations that have traditionally been done on CPUs. GPUs have been used to run "space and time discrete simulations" called stencil codes [5]. They have been used for problems in physics such as solving Boltzmann equations for gas flow applications [6]. Scientists have used GPUs in bioinformatics for doing sequence alignment [7], for correcting errors in DNA sequencing [8], and weather forecasting [9]. GPUs are also being used for colorectal cancer research [10], cheminformatics [11], finite element numerical integration [12], n-body problems [13], dynamic programming applications [14], and many other scientific and mathematical applications. GPUs are being used as components in some of the newest and fastest supercomputers because of their ability to provide significant computational capability, while using less power than some CPU alternatives. The success that researchers have had with a wide variety of applications motivated us to evaluate whether GPUs would decrease the amount of time to solve the party problem.

3. DATA STRUCTURES AND ALGORITHM

As we designed our application, we were mindful of the idea that it would need to run efficiently in parallel on both multicore systems using OpenMP and on GPUs using CUDA with minimal revision to the code to port it from the CPU to the GPUs. The key implication of this is that the code had to run with a minimal amount of communication between processing cores when running on the CPU only and when running on the GPUs only. Because our application just requires us to test a large number of graphs, it is really just an embarrassingly parallel application. Therefore, we realized that we could simply divide the number of graphs evenly by the number of cores (CPU or GPU as appropriate), have each core report the results of testing its share of graphs to a master processor, and have that processor output the results. As long as each core would be able to efficiently generate the next graph that the core was supposed to test, this design would work well. This requirement affected the data structure we used to represent the graphs.

3.1 Data Structures

As discussed in section 1, the party problem can be solved using graphs. In computer programs, a graph is typically represented using adjacency lists or an adjacency matrix. For our purposes, an adjacency matrix is preferable to adjacency lists for both performance and ease of implementation reasons. While a natural representation of a graph using an adjacency matrix is a two-dimensional array, for our application, a single-dimensional array was more useful. To help illustrate this, we show a simple graph with five vertices and the corresponding two-dimensional adjacency matrix in Figure 1 where a red edge is represented as a zero, a blue edge is represented as a one, and no edge is represented as -9. The adjacency matrix in Figure 1 indicates there are no edges from a vertex to itself, because in the party problem, a person is assumed to know himself/herself and thus there is no need for an edge from a vertex to itself. Therefore, the information on the diagonal of the matrix from the upper left entry to the lower right entry does not need to be stored. Because the relationships between two people are symmetric (if person 1 knows person 2, then person 2 knows person 1), the information above the diagonal containing -9s is the same as the information below the diagonal and is therefore redundant. By only storing the information above the diagonal, we can save a little more than half the memory that would be required to store the information in the graph. Because many GPUs have large numbers of processing cores, each with fast access to a limited amount of the typically small quantity of memory on the GPU, this optimization was important to allow all the cores to evaluate different graphs in a single instant. In order to save memory, we "flatten" the matrix and store it as a one-dimensional array as shown in Figure 2. While this introduced some complexity during development, the advantages it provided far outweighed the added complexity. Since the flattened array contains only zeros and ones, the digits it contains may be thought of as a binary number representing the graph. By adding one to the rightmost element of the array and making any carries if appropriate to the elements to the left, one has adjusted the array to contain the binary representation of the next graph. By continuing to add one to the digit in the rightmost array slot and making the appropriate carries, one can generate all of the possible graphs, one at a time. This process is a very efficient way for a processing core to generate the next graph to test, and the primary advantage of using the one- dimensional array representation of the graph. By assigning equally sized sets of graphs to each core where the graphs in a set can be represented as consecutive binary numbers, the cores were able to efficiently generate each subsequent graph that they needed to test, making the one-dimensional array an excellent data structure for storing the graphs.

3.2 Algorithm

The algorithm we used was a naïve algorithm. The algorithm tested the first 335,544,320,000 (which is between 2^{38} and 2^{39}) graphs of the 2^{990} graphs that need to be tested to determine if R(5, 5) ≥ 46. We refer to the algorithm as naïve for two reasons. The first reason is that the algorithm is a brute force approach to tightening the lower bound on the value of R(5, 5). Even more importantly, the algorithm was not optimized for running on the GPUs. For many applications, it is possible to get significantly better performance on algorithms that are optimized to run on GPUs. However, we were more concerned with the speedup that could be obtained with minimal effort.

In the version of the code that the CPU ran and the version that the GPUs ran, each of the n processing cores was given the value of n and assigned $1/n^{th}$ of the graphs to test by being given a

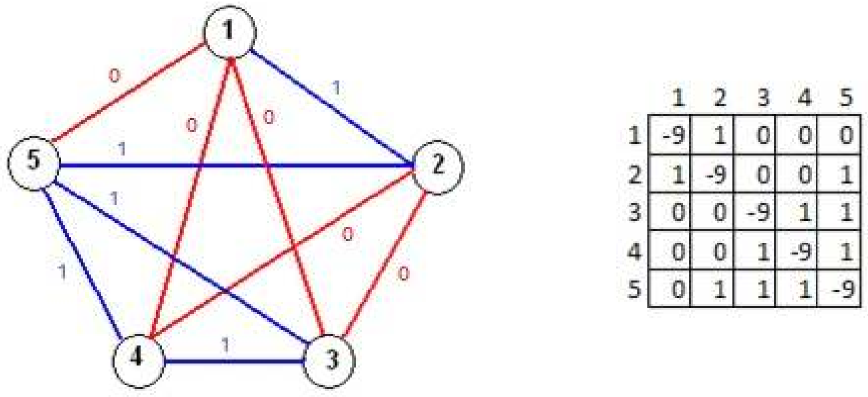

Figure 1. A Five Vertex Graph and Its Adjacency Matrix

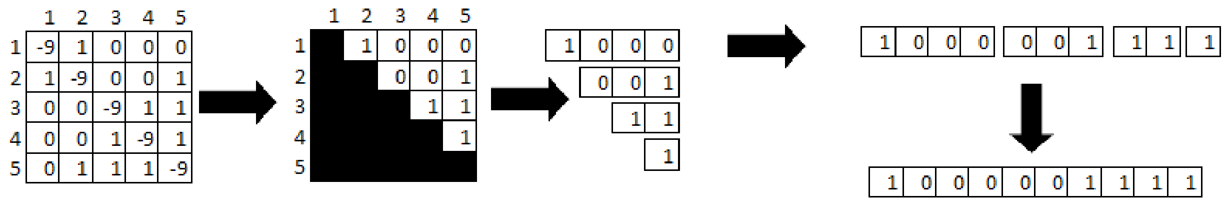

Figure 2. Flattening the Two-Dimensional Array to a One-Dimensional Array Keeping Only the Necessary Information

decimal value that the core converted into a binary number, corresponding to the first graph in the range of graphs to test. The cores were then able to calculate the number of graphs they needed to test. The algorithm we used to test a graph to see if it contains a red or blue K_5 cycles through all of the possible sets of five vertices until it finds a set that forms a red or blue K_5 or until it has determined that no set of five vertices can contain a red or blue K_5. Once a K_5 is found, the graph is discarded and the next graph is generated by adding 1 to the binary representation of the discarded graph to obtain the next graph to test. In the OpenMP version of the code for the CPU, whenever a core found a graph with neither a red or blue K_5, it was supposed to print out the graph, but this situation never arose. In the CUDA version of the code, whenever a core found a graph with neither a red or blue K_5, it was supposed to write the graph to a shared memory location and set a flag in shared memory. When the cores were all finished, the flag was copied to the CPU and if a graph with no K_5 had been found, then the shared memory with the graph's information would be copied to the host system and printed out.

4. METHODOLOGY

We did our testing on an upgraded Gateway GT5674 computer with an AMD Phenom 9500 2.2 GHz quad-core CPU, 4 GB of RAM, and a 650 watt power supply. The computer ran the Windows Vista operating system. The GPUs we used ranged from a low end home user card (GeForce 9500 GT) to a high end home user card (GeForce GTX 480) to a high end workstation card (Quadro FX 5800). The GPUs had varying numbers of CUDA cores, multiprocessors, memory, and different compute capabilities, as shown in Table 1.

In order to determine the performance improvement we could obtain by using the GPUs, we first ran a version of the code using OpenMP on the system using 1, 2, and 4 cores. We then ported the code to CUDA such that it would not use the CPU cores for testing any graphs, thus using the CPU minimally, and ran it on each GPU shown in Table 1.

For the performance testing, we chose to have the program test 335,544,320,000 graphs. That number was chosen because it was a multiple of the number of cores in every one of the GPUs we used and the algorithm took about an hour to test that many graphs using all 4 CPU cores. By picking a value that took the CPU a long time, we expected to see what, if any, performance gains we could reasonably expect under normal conditions of the cores having a huge number of graphs to test, which is what we would encounter if we tried to solve the problem with GPUs or CPUs.

Our only effort to tune the CUDA version of the application in an attempt to achieve better performance was to run the tests on each card using several different numbers of threads per block. In order to achieve high performance and scalability, NVIDIA GPUs divide the threads for a multithreaded CUDA program into groups of threads called blocks [16]. Each block runs on a streaming multiprocessor, so the number of blocks that can run simultaneously and thus the performance of an NVIDIA GPU

Table 1. Specifications for GPUs Used

GPU	CUDA Cores	Streaming Multiprocessors	Memory (GB)	Compute Capability
GeForce 9500 GT	32	4	1	1.1 [15]
GeForce GT 240	96	12	0.5	1.2 [15]
GeForce GTS 450	192	4	1	2.1 [15]
Quadro FX 5800	240	30	4	1.3 [15]
GeForce GTX 480	480	15	1.5	2.0 [15]

scales with the number of streaming multiprocessors [16]. One can adjust the number of threads that run in each block to try to obtain better performance by using more blocks, to a maximum of 512 threads per block on GPUs with compute capability 1.x and 1024 threads per block on GPUs with compute capability 2.x [16]. Since we had GPUs with both 1.x and 2.x compute capabilities, we chose to limit the threads per block to 512 for all of our testing for consistency. Because some people have suggested that the number of threads per block should be at least 64 as well as being a multiple of 64 [17] and others have found that 128 to 256 threads per block resulted in the best performance for their applications [18], we also ran tests with 64, 128, and 256 threads per block in addition to 512 threads per block. The GeForce 9500 GT which was unable to run the code using 512 threads per block due to hardware limitations, so we were only able to test it with 64, 128, and 256 threads per block on that GPU. We discuss the results of these tests in Section 5.

In order to try to take full advantage of the GPU for computing rather than also for updating the computer's display, we explored the possibility of using a second graphics card dedicated to the display in our test system. The GPUs with compute capability 1.x were compatible with a second NVIDIA graphics card, an NVIDIA GeForce 6200 that is not CUDA-enabled. This allowed us to use the GeForce 6200 for the display, freeing up the CUDA cards to do only the computations. The cards with compute capability 2.x did not work with the second graphics card. Therefore, we tested all the GPUs without the GeForce 6200. We also tested cards with compute capability 1.x while using the GeForce 6200 for the display. This allowed us to determine if using a GPU for both computations and to run the display had a significant impact on the performance of the cards. We discuss the results of these tests in Section 5.

5. RESULTS

We conducted a test consisting of 10 trials where each of the devices (CPU-only using 1, 2, and 4 cores and GPU-only with each GPU) tested 335,544,320,000 graphs. Each test was repeated for each GPU using each number of threads per block except 512 threads per block for the GeForce 9500 GT. For the GPUs that were compatible with the second graphics card, we conducted 10 trials using the second graphics card to drive the display with each number of threads per block other than 512 blocks for the GeForce 9500 GT. After analyzing the results of 10 trials for each test, we concluded that the range of the results for each GPU and for each number of CPU cores used was sufficiently small to conclude that 10 trials was sufficient to be confident that we were getting accurate results. In 24 of the 30 GPU tests, the range between the values obtained from the 10 trials was 2 seconds or less. In the remaining 6 tests, the ranges were one of 3 seconds (test average 603 seconds), one of 5 seconds (test average 2241 seconds), two of 6 seconds (test averages 634 seconds and 3167 seconds), one of 10 seconds (test average 2258 seconds), and one of 13 seconds (test average 637 seconds). The only GPUs that had a range of more than 1 second for the 10 trials in any given test were the GeForce 9500 GT and the GeForce GT 240, which are the lower end home GPUs. For the CPU core tests, the ranges for the tests were 36 seconds (test average 3254 seconds) for 4 cores, 88 seconds (test average 6846 seconds) for 2 cores, and 130 seconds (test average 13,226 seconds) for 1 core.

5.1 Effects of Using the GPU to Drive Display

We observed that using the GPU to drive the system's display in addition to testing the graphs did increase the amount of time taken to finish testing the graphs, as shown in Table 2. The least powerful GPU (the GeForce 9500 GT) consistently suffered the biggest performance loss in terms of additional time to finish testing the graphs. However, when taking the additional time as a percentage of the time to test the graphs when not using the GPU, the most powerful GPU (the Quadro FX 5800) suffered the biggest performance loss. The increase in time required to test the graphs was only an additional 2.5% or less of the time taken to complete the computations without using the GPU for the system's video output. Therefore, we do not believe using the GPUs to run the systems video is a significant detriment to the performance of the GPU on the computations.

5.2 Effect of Using Different Numbers of Threads Per Block

As discussed in the Section 4, the number of threads per block can have an impact on the performance of the GPUs, so we ran the tests using 64, 128, 256, and 512 threads per block for each card except 512 for the GeForce 9500 GT. We found that there was a significant variation on the running time of the algorithm using different numbers of threads per block, as shown in Figure 3. The data shown in Figure 3 only shows the variation based on threads per block for the GPUs when the second graphics card was not in the system for consistency, but we observed that the data from the tests where the second graphics card was used demonstrated analogous results. In Table 3, we show the percent increase in time required to test the graphs from the tests using the number of threads per block that produced the fastest time to the tests using the number of threads per block that produced the slowest time. The increase in the time required to test all the graphs ranged from 8% to 54% based on the GPU. Although we observed the least impact on the GPU with the most memory, the GPU with the least memory had the second smallest impact which was significantly less than the other GPUs, so the magnitude of the impact was not simply based on the amount of memory of the GPU. 128 threads per block produced the best results with the GeForce GT 240, Quadro FX 5800, and GeForce GTX 480. The GeForce 9500 GT performed best with 64 threads per block, but was only 4 seconds faster than it was with 128 threads per block. The GeForce GTS 450 performed best with 256 threads per block, but only 10 seconds better than with 128 threads per block. Therefore, when comparing the performance of the GPUs, we use the performance from the test with the number of threads per block that was most commonly the best for the GPUs. We note the difference between the speedup obtained by using the performance when the GPUs used 128 threads per block and the performance from the best number of threads per block would be negligible.

Table 2. Performance Comparison of GPUs When Used for Video vs. When Not Used for Video

GPU	Threads Used Per Block	Average Time (seconds) When Using GPU for Video	Average Time (seconds) When Not Using GPU for Video	Additional Time Taken (seconds)	Additional Time as Percent Difference from Average Time When Not Using GPU for Video
GeForce 9500 GT	512	------	------	------	------
GeForce 9500 GT	256	3167.4	3156.3	11.1	0.4
GeForce 9500 GT	128	2257.8	2242.4	15.4	0.7
GeForce 9500 GT	64	2253.9	2241.3	12.6	0.6
GeForce GT 240	512	636.9	634.0	2.9	0.5
GeForce GT 240	256	538.0	531.2	6.8	1.3
GeForce GT 240	128	524.3	521.4	2.9	0.6
GeForce GT 240	64	604.7	604.1	0.6	0.1
Quadro FX 5800	512	297.6	292.4	5.2	1.8
Quadro FX 5800	256	289.0	283.8	5.2	1.8
Quadro FX 5800	128	275.8	269.1	6.7	2.5
Quadro FX 5800	64	277.1	272.3	4.8	1.7

5.3 Speedups

The speedups attained by using the GPUs over using one, two, and four cores of the CPU in our quad core system are shown in Table 4. We note that it is strange that slightly greater than linear speedup was observed when running the program on the CPU only with all 4 cores. One would expect that there would be a small performance loss, rather than performance gain in that situation. We suspect that there might have been some process running on the computer such as antivirus software or a regularly scheduled task from the operating system that did not occur during the time the program was run with 4 cores, but did occur when the program was run with one core and two cores. The GPU with the fewest cores attained a small speedup of 1.44 over using all of the cores of the CPU, meaning it would take about 1.44 identical quad core systems to test the graphs in the same amount of time that GPU did. In contrast, a high-end GPU (GeForce GTX 480) attained a speedup of greater than 21 over the quad-core system using all of the cores of the CPU, meaning it would take over 21 identical quad core systems to test the graphs in the same amount of time that GPU did.

6. CONCLUSIONS

Our results have shown that a brute force solution to the party problem would be greatly speeded up running on GPUs, as our fastest GPU did the same amount of work as our host system in a fraction of the time. It would take more than 21 of our quad core systems or about 88 of the CPU cores in our host computer to do the same amount of work as the GPU in the same amount of time. We note that although faster CPUs are now available, they should still be significantly outperformed by the single GPU. Also, with systems available that can run up to 8 GPUs and with faster GPUs available, it appears that using GPUs would be a good way to make progress on the party problem. However, while we have seen the great potential of GPUs to speed up the process of making progress on the party problem, we have two concerns. Our program only tested graphs that allowed the cores to run continuously in parallel. When certain graphs much further along in the set of graphs to test begin to be tested, due to the GPU architecture, the code of our algorithm will have to branch in various places, which will force it to run sequentially at times, decreasing the performance gains we got from the GPUs. An issue that is much more problematic than the branching is that the brute force solution requires us to test 2^{990} graphs to raise the lower bound on R(5, 5) to 46 or to move the upper bound to 46. Our idea of using GPUs does not scale sufficiently by itself to accomplish this in a reasonable amount of time. While we were able to test more than 2^{38} graphs in 2.5 minutes using a small quantity of hardware available to any consumer, a back of the envelope calculation suggests that it would take more than a year on the Titan supercomputer which has 18,688 GPUs to test just the first 2^{66} graphs, leaving us too far from completing the tests we need to solve the problem [19]. Even a new generation of supercomputers with many times the number of GPUs as current supercomputers where the GPUs were also many times faster and had many times the number of cores as the current GPUs would still not let us solve the problem. Therefore, to devise a workable method to solve the party problem in a

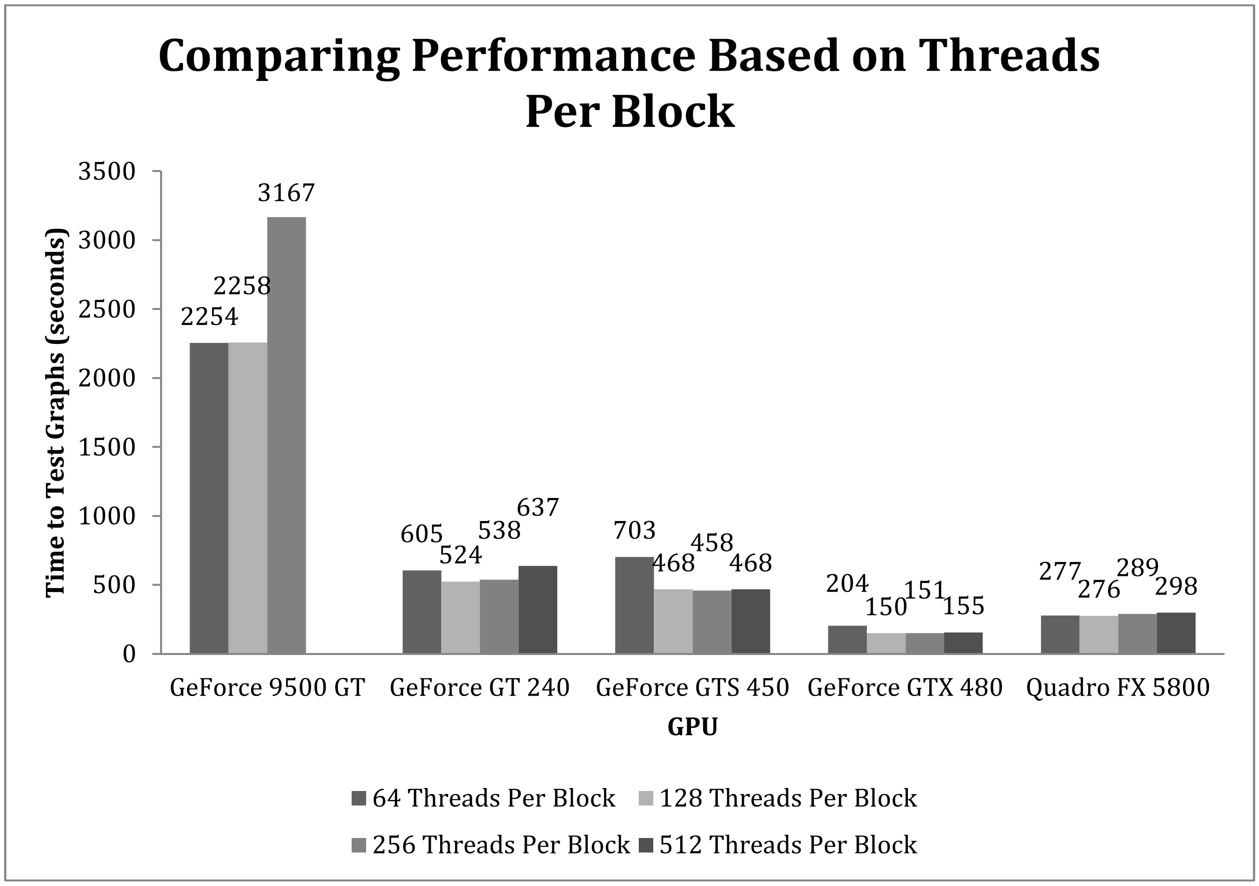

Figure 3. Comparison of Running Time of GPUs Using Various Numbers of Threads per Block

Table 3. Comparison of Percent Increased Time Required by GPUs to Test Graphs with Best and Worst Numbers of Threads Per Block

GPU	Percent Time Increase
Quadro FX 5800	8
GeForce GT 240	21
GeForce GTX 480	36
GeForce GTS 450	41
GeForce 9500 GT	54

reasonable amount of time, we will need to devise a better algorithm than just the naïve brute force algorithm. If the algorithm is amenable to running on GPUs, perhaps using GPUs will help us solve the problem in the future.

7. FUTURE WORK

There are several possible opportunities for future work related to this research. The first strategy we could try is to use a more intelligent algorithm than the brute force algorithm our program employed. Using mathematics, we could decrease the set of graphs to test by eliminating classes of graphs that can be shown will contain a red K_5 or blue K_5 subgraph. This may require us to devise a significantly more complicated algorithm to run on the GPUs, as the GPU cores may no longer be able to generate the next graph to test using the efficient method our current algorithm employed. We could also expand upon work that for a CPU-based system, used a breadth first search technique that employed pruning to try to search the tree of all possible graphs to generate only graphs with at least 43 vertices that contain neither a red or blue K_5 [20].

Table 4. Speedup of Devices

Device	Average Time (seconds)	Speedup Over Single Core	Speedup Over Quad Core System
CPU using 1 core	13226	--------	--------
CPU using 2 cores	6846	1.93	--------
CPU using 4 cores	3254	4.06	--------
GeForce 9500 GT	2258	5.86	1.44
GeForce GT 240	524	25.22	6.20
GeForce GTS 450	468	28.26	6.95
Quadro FX 5800	276	47.92	11.79
GeForce GTX 480	150	88.18	21.69

Another opportunity to explore this problem in more depth is to determine the binary representation of the first graph that causes the CUDA code we wrote to branch and thus run sequentially. We also would like to measure the performance of the cards during the period when the graphs cause the code to branch and compare it to the performance of the OpenMP version of the code running on just the CPU cores for the same set of graphs.

8. REFLECTIONS

This research project was a result of The Blue Waters Undergraduate Petascale Education Program with which I participated in as a student intern. The BWUPEP is an educational program that gives students from colleges and universities around the country the opportunity to learn about the high performance computing discipline and apply what they learn in specific research projects at their home institutions. Previous to my experience with the Blue Waters Undergraduate Petascale Education Program internship, I had limited knowledge about the field of computer science and parallel computing. I knew this internship would be the opportunity of a lifetime for me to get my foot into the door of the computer science world. Over the course of the internship, I learned more than I ever imagined and have gained experience that will be valuable to me for years to come. I am now confident in my abilities to handle issues related to HPC and computationally intense problems.

My Blue Waters Undergraduate Petascale Education Program internship began with a two-week trip to attend Blue Waters Undergraduate Petascale Institute at NCSA at the University of Illinois Urbana-Champaign to learn about High Performance and Parallel Computing. There I met new students from around the country who were also awarded the internship to conduct research at their respective universities and colleges. During those two weeks we underwent intense educational sessions everyday learning the methodologies and ideologies of High Performance Computing and methods used to develop and debug code on parallel computers and clusters. From this boot camp experience, I took back with me the knowledge and ability to conduct my research for this project and future work, as well as new friendships I had formed.

The bonus of my BWUPEP internship and the culmination of all my hard work involved a free trip to SC11 in Seattle, Washington to work as a student volunteer and participant in the education program with the other student interns and volunteers. I created a poster to be presented at the resource fair about my research experience as well as the research I conducted over the summer for this project. Going to SC11 was an eye-opening experience and I hope to attend more of the SC conferences in the future. I got the chance to meet many students in multiple disciplines of computation as well as network with professors and major companies who are involved in supercomputing.

This internship has provided me with the tools and knowledge I need to move forward and continue work and research in the field of high performance and parallel computing. Over the course of the internship, I have learned the basics of a computer's hardware and architecture and how the hardware works along with the operating system to conduct multiprocessing and parallel computing. I have been exposed to multiple ways to parallelize code using OpenMP, MPI, and CUDA. The mentors at the boot camp in Illinois instilled in me the ideologies of high performance computing and taught me the systematic methods to successfully debug parallelized code. I experienced hands on learning by creating several small-scale local clusters on my own with the resources I had available at my home institution. I also learned about Graph Theory, Ramsey Theory, and the mathematics they involve, catching a glimpse of the numerous computationally intense problems of today.

This internship has given me the opportunity to experience first hand the process of research in an academic environment, which I will continue to use in my future work. This experience as a whole has provided me with the knowledge, tools, and experience to be successful in future endeavors. My research work with my mentor, Dr. David Toth, has opened doors for me that I never knew existed. The work I have done for this project has sparked my interest for more computationally intense problems that might hold some potential for me to work on using HPC in the future.

9. ACKNOWLEDGEMENTS

This work was supported by the Blue Waters Undergraduate Petascale Education Program funded by the National Science Foundation's Office of CyberInfrastructure, which provided support for one of the authors in the form of a Blue Waters Undergraduate Petascale Research Internship. This work was also supported by NVIDIA through their Academic Partnership

Program, which provided the GeForce GTX 480 and Quadro FX 5800 GPUs we used.

10. REFERENCES

[1] ramsey theory - Wolfram|Alpha. (2012). http://www.wolframalpha.com/input/?i=ramsey+theory.

[2] Vera Rosta, Ramsey Theory Applications, The Electronic Journal of Combinatorics. December 7, 2004, http://www.combinatorics.org/ojs/index.php/eljc/article/view/ds13/pdf.

[3] S. P. Radziszowski, Small Ramsey Numbers, The Electronic Journal of Combinatorics. DS1.10. (originally published July 3, 1994, last updated August 4, 2009), http://www.combinatorics.org/Surveys/ds1/sur.pdf.

[4] Personal communication with Peter Christopher, Ph.D., January 2005.

[5] A. Schafer, D. Fey, High Performance Stencil Code Algorithms for GPGPUs, Procedia Computer Science 4 (2011) 2027-2036.

[6] Y. Y. Kloss, P. V. Shuvalov, F. G. Tcheremissine, Solving Boltzmann equation on GPU, Procedia Computer Science 1 (2010), 1083-1091.

[7] C. Ling, K. Benkrid, Design and Implementation of a CUDA-Compatible GPU-based Core for Gapped BLAST Algorithm, Procedia Computer Science 1 (2010) 495-504.

[8] H. Shi, B. Schmidt, W. Liu, W. Muller-Wittig, Quality-score guided error correction for short-read sequencing data using CUDA, Procedia Computer Science 1 (2010) 1129-1138.

[9] T. Shimokawabe, T. Aoki, J. Ishida, K. Kawano, C. Muroi, 145 TFlops Performance on 3990 GPUs of TSUBAME 2.0 Supercomputer for an Operational Weather Prediction, Procedia Computer Science 4 (2011) 1535-1544.

[10] GPGPUGRID.net, News archive (2011), http://www.gpugrid.net/old_news.php.

[11] M. Maggioni, M. D. Santambrogio, J. Liang, GPU-accelerated Chemical Similarity Assessment for Large Scale Databases, Procedia Computer Science 4 (2011), 2007-2016.

[12] P. Maciol, P. Plaszewski, K. Banas, 3D finite element numerical integration on GPUs, Procedia Computer Science 1 (2010), 1093-1100.

[13] L. Nyland, M. Harris, J. Prins, Fast N-Body simulation with CUDA, in: GPU Gems, vol. 3, Addison Wesley, 2007, 677-795.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Scheaffer, K. Skadron, A performance study of general-purpose applications on graphics processors using CUDA, J. Parallel Distrib. Comput. 68 (2008), 1370-1380.

[15] NVIDIA, CUDA GPUs | NVIDIA Developer Zone (August 2011) - http://developer.nvidia.com/cuda-gpus.

[16] NVIDIA CUDA C Programming Guide 3.2 (November 2010) – http://developer.nvidia.com/object/cuda_3_2_downloads.html.

[17] S. Walkowiak, K. Wawruch, M. Nowotka, L. Ligowski, W. Rudnicki, Exploring utilization of GPU for database applications, Procedia Computer Science 1(2010) 505-513.

[18] V. W. Lee, C. Kim, J. Chhugani, M. Desiher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, P. Dubey, Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU, Proceedings of the 37th annual international symposium on Computer architecture (2010) 451-454.

[19] Introducing Titan | The World's #1 Open Science Supercomputer (2012), http://www.olcf.ornl.gov/titan/.

[20] S. Krach, A High Performance Computing Approach to Ramsey Theory, undergraduate thesis, Department of Computer Science, Merrimack College, 2011.