12-2011

Analysis and Implementation of Room Assignment Problem and Cannon's Algorithm on General Purpose Programmable Graphical Processing Units with CUDA

Harsh vardhan Dwivedi
Clemson University, hvdwivedi@hotmail.com

ANALYSIS AND IMPLEMENTATION OF ROOM ASSIGNMENT PROBLEM AND CANNON'S ALGORITHM ON GENERAL PURPOSE PROGRAMMABLE GRAPHICAL PROCESSING UNITS WITH CUDA

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
COMPUTER ENGINEERING

---

by
Harsh Vardhan Dwivedi
December 2011

Accepted by:
Dr. Melissa C. Smith, Committee Chair
Dr. Walter B. Ligon
Dr. J. Barr von Oehsen

ABSTRACT

General-purpose Graphics Processing Units (GP-GPU) has emerged as a popular computing paradigm for high-performance computing over the last few years. The increased interest in GP-GPUs for parallel computing mirrors the trend in general computing with the rise of multi-core processors as an alternative approach to increase processor performance. Many applications that were previously accelerated on distributed processing platforms with MPI or multithreaded techniques such as OpenMP are now being investigated to assess their performance on GP-GPU platforms. Since the GP-GPU platform is designed to give higher performance for parallel problems, applications on other parallel architectures are good candidates for performance studies on GP-GPUs.

The first case study in this research is a GP-GPU implementation of a Simulated Annealing-based solution of the Room Assignment problem using CUDA. The Room Assignment problem attempts to arrange *N* people in *N/2* rooms, taking into consideration each person's preference for a roommate. To evaluate the implementation, it was compared against the serial implementation for problem sizes 5000, 10000, 15000 and 20000 people. The GP-GPU implementation achieved as much as 78% higher improvement ratio than the serial version in comparable execution time. The second case study is a GP-GPU implementation of Cannon's Algorithm using CUDA. The GP-GPU implementation is compared with a serial implementation of a conventional matrix multiplication O(n^3). The GP-GPU implementation achieved upto 6.2x speedup over the conventional serial multiplication. The results for both applications with varying problem sizes are presented and discussed.

DEDICATION

I dedicate this thesis to my parents and my academic advisor, Dr. Melissa C. Smith. I also dedicate this work to all members of the scientific community, toiling in laboratories for the advancement of our knowledge and the betterment in the quality of life for people everywhere.

ACKNOWLEDGMENTS

This work acknowledges the guidance and support from my advisor Dr. Melissa C. Smith. I am profoundly indebted to her for her guiding advice and direction in this research. Her guidance helped inculcate a strong motivation for scientific research.

TABLE OF CONTENTS

Table of Contents (Continued)

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

General-purpose Graphics Processing Units (GP-GPUs) have emerged as a leading architecture for parallel application development and acceleration in recent years. Owing to their very high core count and ample on-device memory, they offer high performance potential for suitable applications. This research presents two very different case studies of algorithm implementation and performance analysis on a GP-GPU architecture: the Room Assignment Problem [1] and Cannon's Algorithm [1] for matrix-matrix multiplication.

Monte Carlo methods, which are used in the Room Assignment Problem, have been a popular choice for finding optimal solutions in computational problems. They are particularly useful for multi-dimensional problems where the cost function is optimized by repeatedly making and evaluating moves in a random order. One Monte Carlo technique for optimization is Simulated Annealing. Simulated Annealing techniques have been proposed for many algorithms including the Travelling Salesman problem, the Room Assignment algorithm, and those related to the physical design of computers such as the place and route of circuits [2]. The appeal of Simulated Annealing-based solutions is their simplicity. Once a cost function for a problem is formulated, Simulated Annealing can be applied to optimize the problem. The downside of these Monte Carlo based techniques is that execution times for serial implementations are long since many random steps are necessary to find a solution and serial implementations can only evaluate one move at a time. However, these techniques have a high potential for optimization through parallel implementations, such as GP-GPUs, since each parallel branch of the code can simultaneously generate and process random moves. The inherent parallelism of the GP-GPU architecture is amenable to the implementation of Monte-Carlo based solutions due to the large number of independent processing units, each of which can run parallel branches of the code.

The first case study for GP-GPU implementation is the Room Assignment problem where *N* peopled are assigned to *N/2* rooms while maximizing mutual compatibility between roommates in each room. A solution to this problem using Simulated Annealing has been implemented on shared memory architectures using OpenMP and distributed memory architectures using MPI [3]. This research will present an implementation of the Simulated Annealing solution to the Room Assignment problem on NVIDIA GTX-580 GP-GPU architecture using CUDA and study the performance response for this architecture relative to the algorithm characteristics.

The second case study in this research is an implementation of Cannon’s Algorithm for matrix-matrix multiplication on the same GP-GPU. Matrix-matrix multiplication is found in many applications in high-performance computing such as digital image and signal processing, graph theory and linear algebra subroutines. The performance of the GP-GPU implementation of Cannon’s Algorithm is compared with a common MPI-based implementation. Differences between the two implementations will be discussed and compared with the performance of a traditional $O(n^3)$ implementation. The analysis will provide insight regarding how similar/dissimilar the MPI implementations for homogenous clusters are with the GP-GPU implementations in terms of performance.

The remainder of this thesis is organized as follows: Chapter 2 presents related work, Chapter 3 details the CUDA programming model and the hardware and software used, Chapter 4 presents the Room Assignment and Cannon’s Algorithm case studies with implementation details for both algorithms. Finally Chapter 5 presents results and analysis from both case studies and Chapter 6 offers conclusion and suggestions for future work.

CHAPTER TWO

RELATED WORK

In this chapter we discuss work found in the literature related to the two case studies: the Room Assignment Problem and Cannon's Algorithm. Section 2.1 discusses two previous implementation efforts for the Room Assignment Problem, including a previous shared and distributed memory implementation and a Simulated Annealing-based solution. Section 2.2 covers prior implementation and analysis efforts for Cannon's Algorithm and other parallel matrix multiplication algorithms.

2.1 Room Assignment Problem

In [3], Lazarova presents an MPI and OpenMP implementation of the Room Assignment problem. The OpenMP version is reported to achieve a speedup of 1.7x over a sequential implementation for a problem size of 20,000 people. The MPI implementation uses two strategies: synchronous and asynchronous move generation. Both strategies use a master process that broadcasts the global configuration to the slave processes, which independently generate moves and send them back to the master process, which, in turn, applies them to the global configuration. The differentiating factor between the synchronous and asynchronous strategies is how the slave processes continue to generate and evaluate local moves while the master process updates and broadcasts the global configuration. To prevent clashes between moves generated by any two processes, each process in the asynchronous strategy is allowed to make perturbations only to certain rooms. Such a hazard does not exist in the synchronous strategy since synchronization definitively takes place through the master process after a set number of moves in the slave processes. The 8-processor MPI implementation for 20,000 people achieved a speedup of 4.6x for the synchronous strategy and 3.8x for the asynchronous strategy.

In [4], Martinez-Alfaro et. al., present a Simulated Annealing-based solution for the Classroom Assignment problem, which is analogous to the Room Assignment problem. In this case, classrooms in a large institution are assigned to classes subject to classroom and instructor availability and classroom special resource availability, among other factors. A modular cost function composed of terms related to these factors is used in their proposed optimization approach. The advantage with a modular cost function is that it can be simplified to fit the data available for the problem at hand. The authors explain that they could use this cost function, for instance, with just the meeting times data and without any regard for special resource requirements for a classroom by attaching zero weight to the term(s) related to that data.

Both [3] and [4] describe the use of simulated annealing to provide optimal solutions for a given problem and present their implementations. While [4] does not use a parallelized approach, [3] presents parallelized strategies for simulated annealing methods. [5] and [6] present solutions for different problems (Protein Substructure searching and IC Floorplanning) on GP-GPU architectures with Simulated Annealing techniques. The work in this thesis presents a solution to the Room Assignment problem based on Simulated Annealing techniques on a GP-GPU architecture. A review of the literature did not reveal any previously published results for the Room Assignment problem on the GP-GPU and to the best of our knowledge, this is the first set of results on this architecture.

2.2 Cannon's Algorithm

In [7], Lee et. al, present an implementation of the Generalized Cannon's Algorithm (GCA) that runs on an arbitrary number of processors with toroidal mesh interconnections. In their work, two layers of processor arrays are considered, a virtual array of processors and the actual array of physical processors available. The number of processors in the virtual array is equal to the number of sub-matrices the product matrix is divided into. The virtual array of processors need not be square, which allows the product matrix to be decomposed into a non-square array. Since in general the number of physical processors available is less than the processors in virtual array, the virtual array must be partitioned to fit into the physical array of processors. The virtual array is partitioned differently for each matrix to be multiplied, depending on the number of decompositions of the matrix. The decomposed matrices are then distributed in the virtual array. Attempts are made to map any virtual processors receiving the same sub-matrix block from the matrices to be multiplied to the same physical processor to reduce the communication time during multiplication.

They also propose a partitioning scheme that reduces the number of page faults. If the sub-matrices distributed among the processors are too large, then after each computation and shift step, page faults will occur. With their proposed partitioning scheme, the sub-matrix blocks on each processor are further decomposed into smaller sub-matrices such that each sub-matrix fits into the main memory. After the second step of decomposition, once a processor finishes computation of a sub-sub-matrix, it is shifted to the left-neighbor processor, which performs computation on this sub-sub-matrix. Finally, the shifting procedure stops when the sub-sub-matrix moves back to the original processor. This scheme allows a sub-matrix to be used by all the processors while it is still in the main memory, eliminating many disk-reads. The authors compared the performance of the GCA against the Scalable Universal Matrix Multiplication Algorithm (SUMMA) [8], another parallel matrix multiplication algorithm, and reported that GCA performs better than SUMMA for all matrix sizes tested..

In [9], Alqadi et. al, present various parallel matrix multiplication algorithms, a theoretical analysis of their performance, and a comparison with measured performance. To evaluate the speedup and efficiency of the algorithm, they use the metric: average number of flops per communication access. They analyzed six algorithms: Systolic Algorithm [10], Cannon's Algorithm [11], Fox's Algorithm with squared and scattered decomposition [12], Parallel Universal Matrix Multiplication (PUMMA) [13], SUMMA and Distribution Independent Matrix Multiplication (DIMMA) [14], [15]. They report that the systolic algorithm gave the maximum efficiency followed by PUMA, DIMMA and SUMMA.

In [16], Ismail et. al, present an implementation of the conventional nested 'for-loop' matrix multiplication algorithm with a new programming model called *Serial, Parallel and Concurrent Core-to-Core programming model (SPC^{3}M)* which is targeted toward multi-core processors. They provide the basic guidelines to program with this model and then provide a performance comparison against the same algorithm implemented with OpenMP. Using 24 concurrent threads, they report a speedup of up to 23.7x over the OpenMP implementation.

While [7] and [9] present implementations of Cannon's Algorithm and performance results, [16] presents a multi-threaded accelerated implementation for matrix-multiplication. The implementation in this thesis is similar to works in [7] and [9] in that it is a Cannon's algorithm implementation and similar to [16] in that it uses a multi-threaded approach to implement Cannon's algorithm and completely different from all the previously explored approaches in that this implementation is for a GP-GPU architecture.

2.3 Summary

This chapter covered work related to the two case studies used in this research. The next chapter discusses the CUDA programming model, introduces key CUDA programming concepts and terms, and specifies the hardware and software used.

CHAPTER THREE

CUDA PROGRAMMING MODEL AND HARDWARE, SOFTWARE USED

To obtain optimum performance from a GP-GPU platform using CUDA, it is essential that the underlying architecture of the GP-GPU and the programming model be investigated and understood. This chapter focuses on the CUDA programming model, the software installations, and the specifications of the machine used in this research.

3.1 GP-GPUs and the CUDA Programming Model

The CUDA programming model introduces extensions to traditional C programming. The portion of the code that is identified for parallel implementation on the GP-GPU hardware is encapsulated in a CUDA *kernel*. A kernel contains the code definition for threads that will run in parallel on the GP-GPU hardware. These threads are organized into blocks and multiple blocks are scheduled to run on the GP-GPU hardware at the launch of any kernel. The number of threads per block and the number of blocks scheduled to run at the launch of the kernel are collectively known as the *execution configuration* for that kernel. Figure 3.1, shows a hierarchical view of the blocks and threads while executing a kernel [17]. The CUDA programming model dictates that all thread blocks execute in parallel. CUDA offers built-in device related variables like `threadIdx`, `blockIdx`, and `blockDim` that allow the user to access a particular thread within a block.

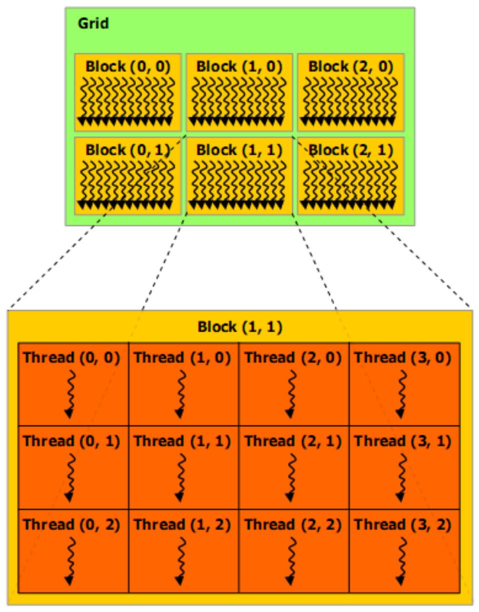


Figure 2-1. Grid of Thread Blocks

Figure 3.1: Hierarchy of threads and blocks during execution

A CUDA-supported GP-GPU consists of multiple streaming multiprocessors (SMs). Each SM is composed of many Single Instruction Multiple Thread (SIMT) processing units called *CUDA cores*, which execute threads from a kernel. The lowest level of execution on the GP-GPU is a warp, which is a group of 32 threads. A group of warps represents all threads in a block and finally the blocks are the top of the execution hierarchy. Normally, each block is tied to a specific SM, but each SM may execute multiple blocks at any given time. With the Fermi Architecture, each SM can execute a maximum of eight blocks concurrently. Because of the SIMT architecture in the GP-GPU, all threads in a warp execute the same instruction and if any thread in a warp deviates from the common stream of instructions, execution in that warp is serialized. Thus branching instructions potentially hurt the overall performance in GP-GPU codes.

The maximum block size, i.e. the maximum number of threads a block can have, is 1024 on the Fermi architecture. The maximum number of threads that a SM can execute concurrently is 1536: a maximum of 48 concurrent kernels on a SM multiplied by the warp size of 32 threads yields 48x32=1536 threads.

For the maximum performance, the number of threads in a block (block size) should always be a multiple of 32, which ensures that none of the warps from that block will be under-populated. Consider the following scenario of 193 threads per block. This block will be split into 7 (ceil(193/32) = 7) warps. The 7th warp, however will only contain 1 thread and would thus be under-populated from its ideal size of 32 threads and does not provide a reduction in processing time. The execution time would be the same even if the 7th warp contained 32 threads, hence a loss of potential performance or underutilization.

3.2 CUDA Memories and Occupancy

The memory hierarchy in CUDA is comprised of the *global*, *shared*, and *local* memory. The global memory can be accessed (Read/Write) by any thread from any block. The shared memory is local to a block and can only be accessed by the threads from that block. Local memory is only accessible from that thread. When no shared memory is used for a block and all the threads are accessing only the global memory, it is recommended to keep the block size lower so that the scheduling engine has maximum freedom to schedule the blocks on any free SM.

Another important term that should be considered when tuning the application for performance is *Occupancy*. Occupancy refers to the ratio of the number of warps running on an SM processing unit to the theoretical maximum number of warps that can run on the SM. Occupancy is important for kernels that are *bandwidth bound*. Note that occupancy does not directly imply performance, however a higher value of occupancy can assist in hiding latency in *memory bound* kernels. To calculate the occupancy, NVIDIA provides an Excel workbook called “CUDA Occupancy Calculator” that calculates a kernel’s occupancy based on the number of threads in a block, number of registers per thread, shared memory per block, and the compute capability of the GP-GPU device. Version 2.4 of this workbook was used in this research.

3.3 Hardware Used

The experimental set-up consisted of two systems. The Room Assignment problem was implemented on the first system, which consisted of a single Intel Xeon 2.66 GHz Quad Core processor with 4 GB system memory running the Ubuntu 11.04 64-bit operating system and a NVIDIA GTX-580 GP-GPU with 3 GB of device memory. The Cannon’s Algorithm was implemented on the second system, which consisted of an Intel Core 2 Quad Core 2.66 GHz with Windows 7 64-bit operating system and a NVIDIA GTX-580 with 1.5 GB device memory. Both GP-GPUs used in this research were from the Fermi architecture family with compute capability 2.0. Each has 16 SMs with 32 CUDA cores each for a total of 512 CUDA cores on these devices. Tables 3.1 and 3.2 provide key details about the GTX-580 devices used in this research.

Table 3.1: System #1 GTX-580 Important Device Characteristics

GPU Clock Speed	1.57 GHz
Total Global Memory	3072 MB
Total CUDA Cores	512 (16 Multiprocessors x 32 Cores/MP)
L2 Cache Size	786432 Bytes
Total Registers per block	32768
Warp Size	32
Max Threads per block	1024

Table 3.2: System #2 GTX-580 Important Device Characteristics

GPU Clock Speed	1.59 GHz
Total Global Memory	1504 MB
Total CUDA Cores	512 (16 Multiprocessors x 32 Cores/MP)
L2 Cache Size	786432 Bytes
Total Registers per block	32768
Warp Size	32
Max Threads per block	1024

3.4 Software Used

Both GP-GPU implementations were developed with CUDA version 4.0 SDK and associated toolkits. The serial version of the Room Assignment problem was compiled with g++ version 4.4.5 and the serial O(n^3) multiplication for comparison with

Cannon’s Algorithm was developed with Microsoft Visual Studio 2010. The GP-GPU implementation of the Room Assignment algorithm also uses the CUDA Thrust library [18]. Thrust is a C++ template library providing access to multiple parallel algorithms for rapid development of applications. The parallel algorithms implemented with Thrust are themselves accelerated with the GP-GPU.

The biggest benefit from using the Thrust library for the Room Assignment problem was the convenient allocation of memory on the host and GP-GPU device. The host and device memory could be allocated by simply using the “`vector`” construct from the Thrust library. The vector can be of type “`host`” or “`device`”, the former referring to allocation on the host and the latter referring to allocation on the GP-GPU device. It was also useful for debugging purposes. At any point, any device variable can be printed with the “`cout`” operator. If the CUDA API is used directly, the variable must first be copied to a host variable and then print that host variable. The use of `Thrust::min_element()` function allowed accelerated parallelized search through an array of values to provide the search result and location of the result. This function is further detailed in the next chapter.

3.5 Summary

This chapter discussed the CUDA programming model, the software environment used and key hardware specifications and details. The next chapter describes the GP-GPU implementations of the Room Assignment problem and Cannon’s algorithm.

CHAPTER FOUR

CASE STUDIES

This chapter discusses the two case studies: the Room Assignment problem and Cannon's Algorithm for matrix multiplication. Each case study is presented with an introduction to the problem followed by a discussion of the GP-GPU implementation with a focus on the program architecture. Sections 4.1 and 4.2 discuss the Room Assignment Problem and its solution implementation on the GP-GPU while sections 4.3 and 4.4 discuss Cannon's Algorithm with its implementation on the GP-GPU.

4.1 Room Assignment Problem

The Room Assignment problem is concerned with distributing *N* people into *N/2* rooms where each individual has a roommate preference list. The Room Assignment problem was suggested as a research problem by Gale and Shapley in [19] which described the stable marriage problem. The stable marriage problem involves a set of *N* men and *N* women, each with their preference list for individuals of the opposite sex. A stable matching in this problem would be a configuration that pairs a man and a woman such that no two individuals prefer each other over their current partners. While Gale and Shapley showed that for every stable marriage problem instance there would exist at least one stable matching, this is not the case for the Room Assignment problem [20]. For any given problem instance of Room Assignment, a stable matching may or may not exist. In the context of the Room Assignment problem, a stable matching would be one where no two individuals prefer to be matched with each other over their current roommates [20].

In [20], Irving presents an $O(n^2)$ complexity algorithm to formulate a stable solution for any instance of a Room Assignment problem and find that solution if one exists. This thesis explores another approach towards the solution of the Room Assignment problem using a Monte Carlo-based approach with Simulated Annealing. Simulated Annealing is a combinatorial optimization method used for many complex systems. This process is generally done through the use of a cost function whose minimum value is reached through the process of Simulated Annealing as described below. The cost function at any state of the system is associated with the current configuration of the system. Thus, slightly varying the configuration of the system and then evaluating this cost function moves the system in the intended direction.

Simulated Annealing is a Monte-Carlo-based technique to obtain optimized solutions to a variety of problems. In [2], Kirkpatrick et. al., adapted the Metropolis Algorithm [21], for use as a general-purpose solution optimization technique for various problems. They adapted the Metropolis Algorithm by replacing the *total energy of system* with a *cost function*, using a set of parameters $\{x_i\}$ to represent configurations and using temperature as a controlling parameter. Simulated Annealing can be contrasted to an iterative improvement based solution approach. The main advantage of using Simulated Annealing is that the solution can transition out of local optimums at non-zero temperatures. Another important feature of Simulated Annealing in [2] is that "the gross features of the eventual state of the system appear at higher temperatures whereas fine details develop at lower temperatures". The general process of adapting a general problem to Simulated Annealing is presented in the following points from [1]:

- "Decide how to represent Solutions
- Define a cost function for the problem
- Define how to generate a new, neighboring solution from an existing solution
- Design a cooling function"

The Room Assignment problem, which aims to obtain the best possible arrangement of *N* people in *N/2* rooms, was adapted for a Simulated Annealing-based solution. First, individual *i* indicates a preference for individual *j* as a roommate with a *dislike coefficient*, d_{ij}, defined between the two individuals. A higher dislike coefficient value between two individuals indicates a lower preference for them as roommates. For a given configuration of roommates, the cost function is computed by summing the dislike coefficients between individuals in a given room and then summing over all rooms. The total *sum* obtained quantifies the quality of the roommate configuration. The initial sum is equivalent to initial energy of the system and the change in cost function is equivalent to change in energy of the system in the Simulated Annealing approach. Starting with an initial configuration of roommates and obtaining an initial sum for this configuration, a random perturbation is made to this configuration by randomly choosing two people and swapping their roommates. The cost function is then re-evaluated and if the new value of cost function is found to be lower than the previous value, the current configuration is accepted. If the new value is larger, then a comparison is made between $e^{((sum-new\_sum)/t)}$ and a random number generated between [0,1] [1]. If the random number is less than or equal to the exponential term, then the new configuration is accepted despite having a larger sum than the previous configuration. If the new configuration fails both testing conditions (lower sum or comparison with the exponential term) then it is discarded. Next, the temperature is lowered according to the cooling schedule ($T=\alpha T$) and the algorithm proceeds until the temperature reaches zero at which point, the final configuration of roommates is accepted as the optimized solution. In this way the Room Assignment problem is adapted to variables and constructs from Simulated Annealing to arrive at a solution for the Room Assignment problem. Figure 4.1 gives the pseudo-code for a sequential Simulated Annealing-based solution for the Room Assignment problem.

```
Create an initial configuration of roommates
Calculate initial sum
while(T>0)
{
    Pick two people randomly and swap their roommates
    Calculate the new sum with this configuration
    Generate random number (x) in the interval [0,1]
    If(new_sum<old_sum || x<=$e^{((sum-new\_sum)/T)}$)
    {
        Accept new_configuration as current configuration
    }
    Reduce temperature according to cooling schedule (T=αT)
}
```

Figure 4.1: Pseudo-code for Simulated Annealing-based solution for the Room Assignment Problem

4.2 Room Assignment Problem Implementation

Two versions of the Room Assignment Solution problem were written, one for serial execution and another for parallel execution on the GP-GPU. The inputs required to run both versions include the *dislike coefficient file* describing the dislike coefficients between any two individuals, *initial temperature* for simulated annealing, and the *target improvement ratio* (the ratio of original sum to the minimized sum). A higher value of improvement ratio indicates a higher quality of solution. Upon completion of the algorithm, both versions of the program display the minimized sum, the initial sum, the number of iterations, the final temperature, execution time and final solution verification results.

The total execution time is recorded for both versions by measuring from the `while (T>0)` line in the pseudo-code. The parallel implementation on the GP-GPU includes the time required to transfer the dislike coefficient matrix and the initial configuration from the host to the GP-GPU and the final transfer of the solution configuration back to the host from the GP-GPU.

The input files for the dislike coefficients are generated using a Mersenne Twister random number generator [22]. An upper bound of 30,000 for a dislike coefficient value between two people was selected to eliminate the problem of overflow. If the dislike coefficients are very large and the problem size consists of a large number of people, then it is possible that the sum of dislike coefficients can overflow the max bound of the variable holding the sum value, which would give an invalid value for the minimized sum.

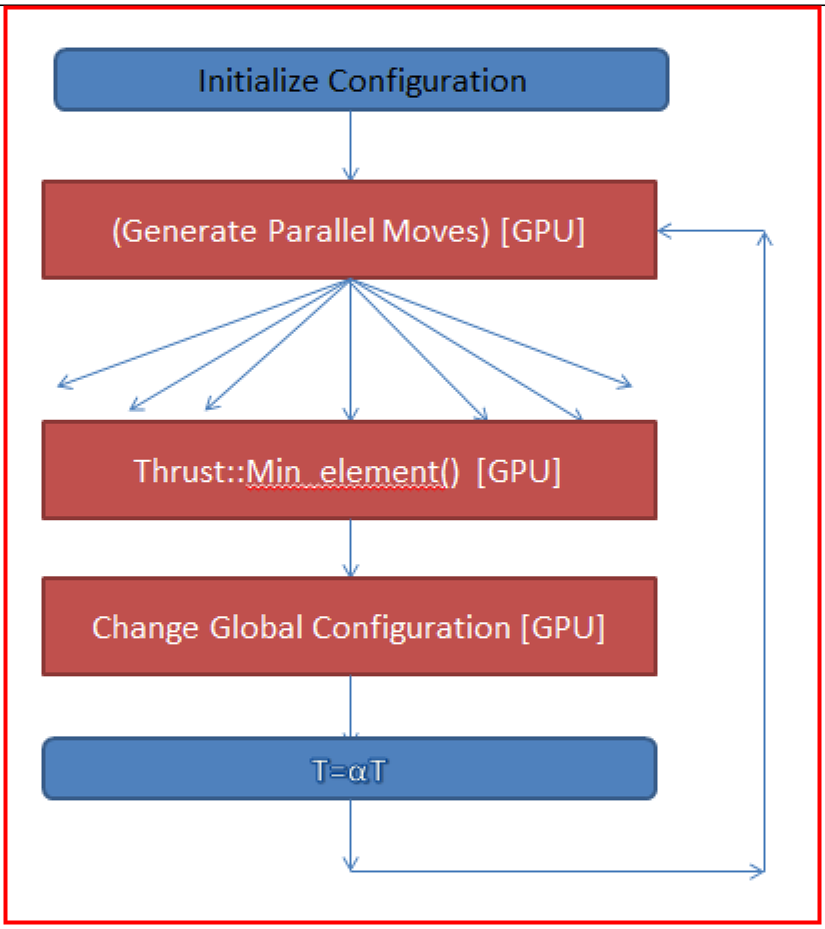

Initialize Configuration
(Generate Parallel Moves) [GPU]
Thrust::Min_element() [GPU]
Change Global Configuration [GPU]
T=αT

Figure: 4.2 Room Assignment GP-GPU Implementation Architecture

Figure 4.2 depicts the architecture of the GP-GPU implementation of the Room Assignment problem. The blue boxes indicate execution stages running on the host and the red boxes show code execution on the GP-GPU. The initial configuration phase (first blue box) of the code runs on the host processor (x86) in both the GP-GPU and serial versions. In this part, an initial room assignment is made and an initial sum is calculated. The initial room assignment assigns the following pairs as roommates: (0,1), (2,3), (4,5) and so on. The quality of any solution produced is measured by the improvement ratio of the solution. Both versions of the code use a dislike coefficient file to determine the dislike between any two people. The dislike coefficients for the pairing (*i*,*j*) and (*j*,*i*) are equal: $d_{ij}=d_{ji}$. As previously mentioned, the dislike coefficients between the two roommates are summed together and this summation is carried out for all rooms giving the total sum for the configuration. The dislike coefficients, which are normally stored in a 2D array when used on the host, are stored in row-major order in a linear array on the GP-GPU.

The GP-GPU code consists of three explicit CUDA kernels: `seedInitRNG`, `k1` and `k2`.

1. `seedInitRNG` is used to initialize the random number generator for each thread.
2. `k1` is the main kernel where every thread generates a parallel move and performs the simulated annealing exponential function calculation, comparing the result to the global sum represented by the first red block in Figure 4.2.
3. `k2` is the kernel used to update the global sum value. `k2` was written to obviate the need to transfer the global configuration back to the host for updating. With the `k2` kernel, the global configuration is updated on the GP-GPU. Updating the configuration consists of updating the global sum value and the current arrangement of roommates. In Figure 4.2, the third red block represents this stage of execution.

The following sub-sections, 4.2.1 to 4.2.5, provide the implementation details including random number generation, solution verification, move selection, etc. for the Room Assignment problem.

4.2.1 Random Number Generation

Kernel `k1` generates parallel moves for simulated annealing using random numbers generated inside the GP-GPU kernel with the *CURAND* library. In contrast, the serial version of the code uses the Mersenne Twister random number algorithm [22] due to the quality of random numbers generated by that algorithm. The MPI implementation in [3] also uses the Mersenne Twister algorithm for random number generation.

The *CURAND* library API provides functions that are used both in the GP-GPU kernel and the host code to generate random numbers: `curand()` and `curand_uniform()`. The function `curand()` was used to generate the two random people who swap roommates and the function `curand_uniform()` was used to generate uniform random numbers between 0 and 1. Although the specification of the Metropolis Algorithm in [23] mentions generation of uniform random numbers in the range [0,1], the *CURAND* library guide [24] specifies that `curand_uniform()` generates numbers in the range (0,1]. While this is different from the exact algorithm specifications, we did not observe any significant difference in performance.

The *CURAND* random number generator (RNG) has a period greater than 2^{190} [24]. Since each thread has its own RNG, each thread must have its RNG initialized. For initialization, each thread receives the same seed, offset as 0, and a different sequence number. The `threadnumber` or the `threadID` is used as the sequence number when initializing each RNG in the thread. The initialization is done through the `seedInitRNG` kernel.

4.2.2 Cooling Schedule

To compare the performances of the serial and the GP-GPU version of the code, it was determined that a better metric would compare the quality of the solution produced by both the versions in an equivalent amount of execution time. Therefore, attempts were made to equalize the execution times of the serial and GP-GPU implementations. With a cooling constant of 0.999999 for the serial version and 0.999 for the GP-GPU version and sufficiently varying the number of blocks for parallel move generation on the GP-GPU, the execution time of both implementations were approximately equal for the problem size of 5000 people. Subsequently, the same values for cooling constant and number of blocks are used for the GP-GPU version for problem sizes 10000, 15000 and 20000 people. With this approach we were able to study the performance characteristics of the GP-GPU version as the problem size varied with a fixed cooling constant. We were able to observe that the GP-GPU version naturally moved toward a speedup against the serial version even though both the serial and the GP-GPU version started with equalized runtimes. These results, as discussed in the next chapter, revealed the performance characteristics of the GP-GPU architecture with a fixed cooling constant. As discussed in Chapter 6, future work will include equalizing the runtime for every problem size and then studying the resulting quality of solutions achieved.

4.2.3 Solution Verification

Two verification checks are applied to the solutions produced by both implementations. The first verification checks the validity of the solution configuration as shown in Figure 4.3. The verification is performed by visiting each person and verifying that the listed roommate also has this person listed as their roommate. If both people list each other as the roommate, the verification is confirmed. The second verification recalculates the configuration sum on the host side and compares it to the minimized sum received from the GP-GPU.

```
for(i=0; i<NUM_PEOPLE; i++)

{

        if(a[a[i]]!=i)

        {

                valid_soln='E';

                printf("\nWRONG SOLUTION!");

        }

}
```

Figure 4.3: Verification To Check Final Solution Validity

4.2.4 Move Selection

As seen in Figure 4.2, for every temperature step, the GP-GPU implementation evaluates multiple moves while the serial version evaluates only a single move. The GP-GPU implementation goes through total of 1417415 (~1.42x10^6) time steps. With evaluation of 19200 (100 blocks x 192 threads per block) moves per time step, the GP-GPU implementation is evaluating 2.72×10^{10} moves. The serial version goes through 1418122547 (~1.42×10^{9}) time steps and effectively the same number of moves since it evaluates only one move per time step. Thus the GP-GPU version is able to go through close to 19 times as many moves as the serial version in the same execution time.

Once all of the threads finish generating and storing moves and calculating the new sum values, the least sum value must be selected. There are three potential ways to implement this selection. The simplest option from a coding perspective would be to copy the sum values back to the host, search for the least value among them and then send the thread index responsible for this value back to the GP-GPU. This approach involves extra overhead to transfer the data between the host and GP-GPU frequently. A second option is to write a kernel that performs this operation, launch a single thread to run this kernel that searches for the least sum value. This approach avoids the time required to copy data to the host by running the search part on the GP-GPU, but still has the disadvantage of an inherently sequential search. The third option is to use a GP-GPU accelerated parallel search for selecting the thread with the minimum sum. This approach has the advantage of no data transfer back to the host as well as utilizing the available parallelism on the GP-GPU. Therefore, to search for the best move among all the parallel generated moves, the reduce algorithm `thrust::min_element()` from the GP-GPU accelerated Thrust library was used. The function `min_element()` is a reduce algorithm implemented using a tree based reduction strategy. Multiple parallel executing units on the GP-GPU compare two items from the array and the array values are reduced to an intermediate smaller set of values. The comparison continues until a single value emerges. In this way, the search for smallest value is accelerated with parallelization. Figure 4.4 provides an illustration of a tree based reduction strategy.

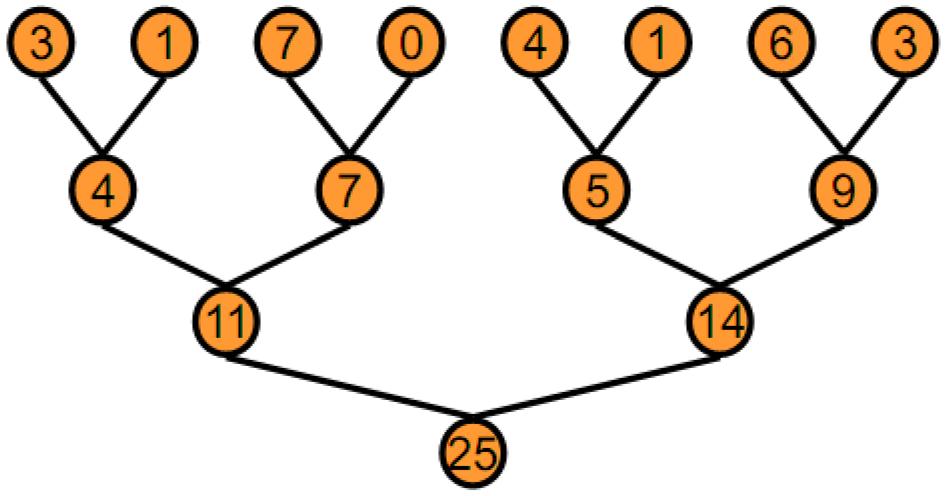


Figure: 4.4 Tree Based Parallel Reduction Strategy [25]

Another benefit from using the `thrust::min_element()` function is that the user does not have to optimize the number of threads and the number of blocks to be launched. The Thrust library automatically optimizes these variables for maximum performance. Currently, the Thrust library optimizes the maximum occupancy performance metric for the kernel.

4.2.5 Execution Configuration Selection For Parallel Move Generation

For GP-GPU implementations, it is desirable to have the maximum possible number of simultaneously running threads. This criterion will maximize the number of parallel moves generated, which increases the probability of reaching better solutions. The Roommate Assignment Problem implementation does not require the threads to use shared block memory and each thread in any thread block is completely independent of any other thread whether in the same block or in another thread block. The CUDA occupancy calculator was used to calculate the optimum number of threads in a thread block for the highest occupancy. The CUDA Occupancy calculator is an Excel workbook where the user enters the Compute Capability of the device, threads per block, registers per thread, and shared memory per block.

The number of registers per thread for a given kernel is obtained by compiling the CUDA code with the option `"--ptxas-options=-v"`. The compiler provides a verbose output with this option and the number of registers for the kernel is provided in this output. With this option it was determined that kernel `k1` uses 25 registers per thread. Figure 4.5 shows the graph generated by the CUDA Occupancy calculator. The multiprocessor warp occupancy is highest for 192, 384 and 576 threads per block. Out of these, the lowest block size of 192 was chosen so that the CUDA scheduling engine would have the maximum flexibility to schedule the blocks on any SM.

After setting the thread block size to 192, the number of blocks was varied such that the execution time for the serial and the GP-GPU code was approximately equal, which allows for comparing the performance in terms of the quality of solution for both the serial and parallel code.

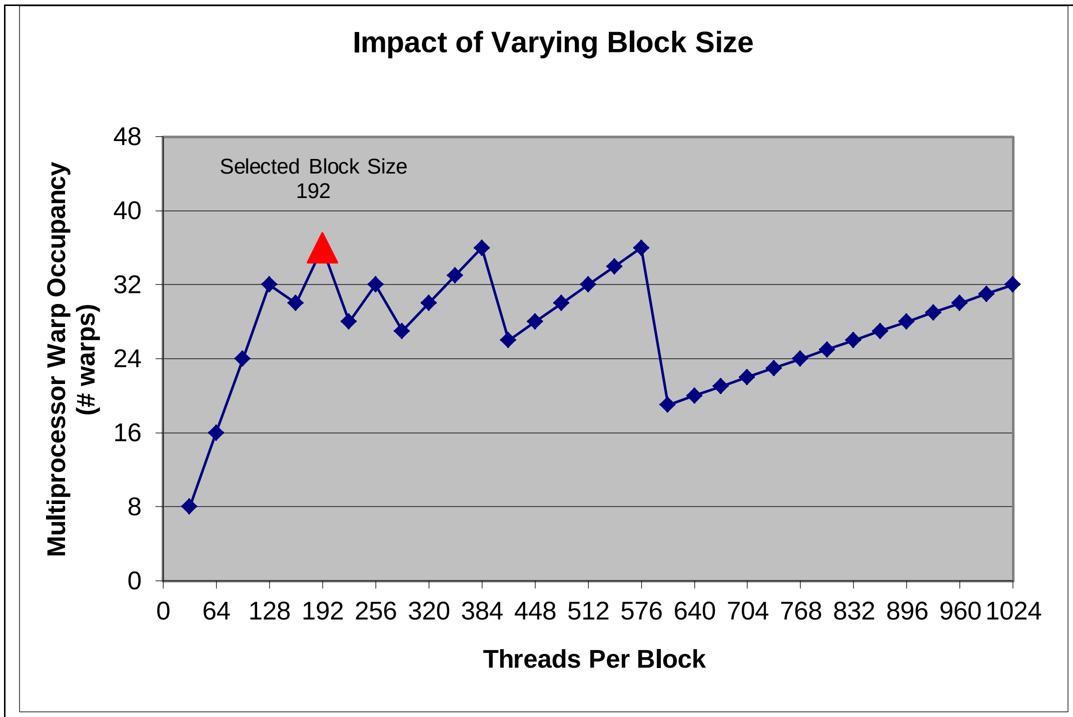


Figure 4.5: Multithreaded Warp Occupancy vs Threads Per Block from the CUDA Occupancy Calculator

4.3 Cannon's Algorithm

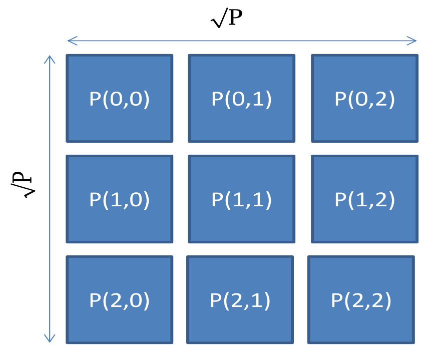
Traditional implementations of Cannon's Algorithm have used MPI-based clusters where each node computes a single block of the product matrix. In this regard, this work investigates the performance of Cannon's algorithm on a different parallel architecture, a GP-GPU. While other fast and highly optimized libraries for matrix multiplication such as CUBLAS [26] and MAGMA [27] exist for GP-GPUs, this work is an investigation of the architectural portability of MPI-like implementations of Cannon's algorithm on the GP-GPU.

In Cannon's Algorithm [11], two matrices, each of size *(NxN)* are multiplied. The algorithm uses *P* processors for computing the product matrix. Each processor computes an *((N/√P)x(N/√P))* block of the product matrix. Traditionally, Cannon's Algorithm is implemented on a two-dimensional mesh *(√Px√P)* of *P* processors. Two matrices, each of size *(NxN)*, are multiplied where each processor multiplies a sub-matrix of size *((N/√P)x(N/√P))* effectively dividing each matrix into *P* sub-matrices. This implementation assumes that *P* is a perfect square and that *N* is a multiple of √*P*.

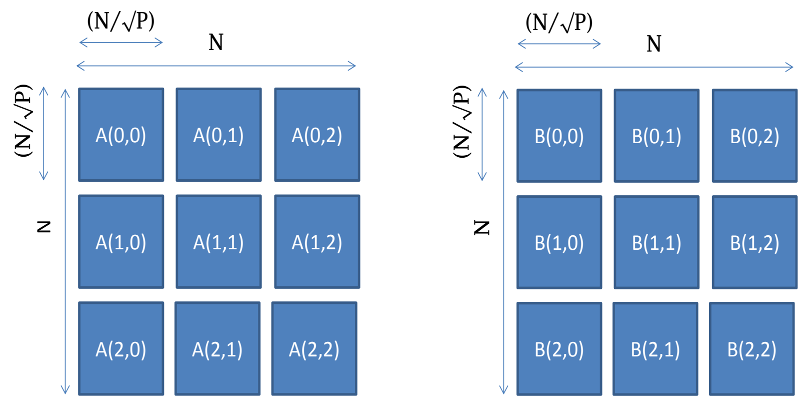
The first step in Cannon's algorithm as shown in Figure 4.6, is dividing both matrices into sub-matrices and then distributing them to the processors in the two-dimensional mesh. Each of the matrices, *A* and *B*, is divided into a grid of sub-matrices of the size, *((N/√P)x(N/√P))*. In the next step, called skewing, each row *i* of this grid from matrix *A*, is shifted left *i* times, with wraparound and each column *i* from the grid of matrix *B*, is shifted *i* times upwards with wraparound. After the skewing phase, each processor multiplies its two sub-matrices and accumulates the result in its product sub-matrix. After each multiplication, each row from the sub-matrix grid of matrix *A* is shifted left once (with wraparound) and each column from the sub-matrix grid of matrix *B* is shifted up once (with wraparound). This process continues until all sub-matrices have reached their starting locations at the end of skewing. Figures 4.6 and 4.7 together depict a complete flowchart describing Cannon's Algorithm.

Consider two matrices A and B (each of size (NxN)) to be multiplied and the product matrix be C (NxN).

1. Consider a grid of P (√Px√P) processors.



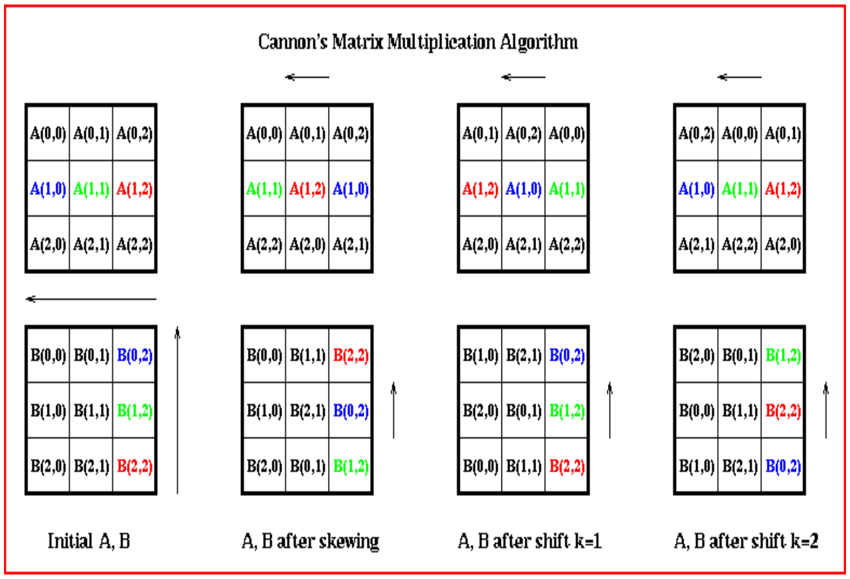
2. Matrix A and B are divided into P sub-matrices each. Each sub-matrix is of the size ((N/√P)x(N/√P)).



It must be noted here that each tile in the above grids of Matrix A and B, represents a sub-matrix. Example: A(0,1) represents a sub-matrix of size ((N/√P)x(N/√P)) from main matrix A which was of size (NxN).

Figure 4.6: Flowchart Describing Steps Involved In Cannon's Algorithm

3. Each matrix is skewed. Each row i of grid of sub-matrices from A is shifted left i times with wraparound and each column j of grid of sub-matrices from B is shifted up j times. The following figure [29] shows the process of skewing for sub-matrices from A and B.



4. After skewing, each processor in the processor grid has the right sub-matrix from A and B. Each processor multiplies the two sub-matrices available to it. After this, each sub-matrix from A is shifted left once and each sub-matrix from B is shifted up once and the processors multiply the two new sub-matrices available to them. Each processor accumulates the result of sub-matrix multiplication in the product sub-matrix it is responsible for computing. The shifting of sub-matrices and consequent multiplication and accumulation of new sub-matrices at each processor continues until all the sub-matrices have reached the original position they had after skewing.

Figure 4.7: Continued Flowchart Describing Cannon's Algorithm

4.4 Cannon's Algorithm Implementation

The GP-GPU implementation differs from the previous MPI implementation [28] in one major aspect, instead of sending and receiving sub-matrix blocks between processors, no such communication is required on the GP-GPU between the processing elements. Instead both matrices are copied to the GP-GPU once at the start of calculations and then copied back at the end of computation. A comparison of the GP-GPU performance versus the serial implementation of the conventional matrix multiplication O(n^3) follows.

The GP-GPU implementation of Cannon’s Algorithm transfers both of the matrices to the global memory of the GP-GPU in one operation. Individual threads on the GP-GPU mimic the role of processors in the MPI implementation by multiplying sub-matrices. Each thread requires the pointers to the sub-matrices as an input argument. The use of pointers reduces the amount of communication required because after each multiplication, only the pointers for an individual thread would change and no other data communication is necessary.

The GP-GPU implementation uses flattened data structures, i.e. both matrices of size (*NxN*) are converted into 1D arrays and then stored in row-major order in the GP-GPU global memory. A sub-matrix is accessed through a pointer to the first element of the sub-matrix. Therefore, to access a grid of sub-matrices, an array of pointers is used. Each pointer in this array points to the first element of a specific sub-matrix. Hence, the grid of sub-matrices is accessed using a 2D array of pointers. This 2D array itself is flattened and is stored in a row-major format in the GP-GPU memory. Figure 4.8 shows the architecture of the GP-GPU implementation of Cannon’s algorithm. The red blocks refer to GP-GPU execution through a kernel and the blue boxes are implemented on the host processor.

The GPU implementation of the algorithm consists of the following kernels:

1. `Pointer_assignX()`: This kernel populates the pointers in the pointer array with addresses to the starting elements of the sub-matrices. It takes as input the dimensions of the processor array, which is (√Total number of threads) in the GP-GPU implementation, pointer to the pointer array and the size of the main matrices (*N*).
2. `Row_shift_gpu()`: Used to shift the rows of a matrix. This kernel shifts the rows of the pointer array, which in turn shifts sub-matrices from one thread to another. This kernel avoids unnecessary data transfers to the host and the associated communication costs.
3. `Col_shift_gpu()`: Used to shift the columns of a matrix. Same as the above kernel except this kernel shifts columns of the pointer array to shift the corresponding sub-matrices.
4. `Kernel_multiply()`: Requires pointers to two sub-matrices as input arguments. This kernel multiplies the two-submatrices and accumulates the product sub-matrix for the given thread. This kernel performs the main computation in the implementation. The multiplication of the sub-matrices is performed just as in a traditional Cannon's Algorithm using the $O(n^3)$ multiplication for each sub-matrix multiplication.

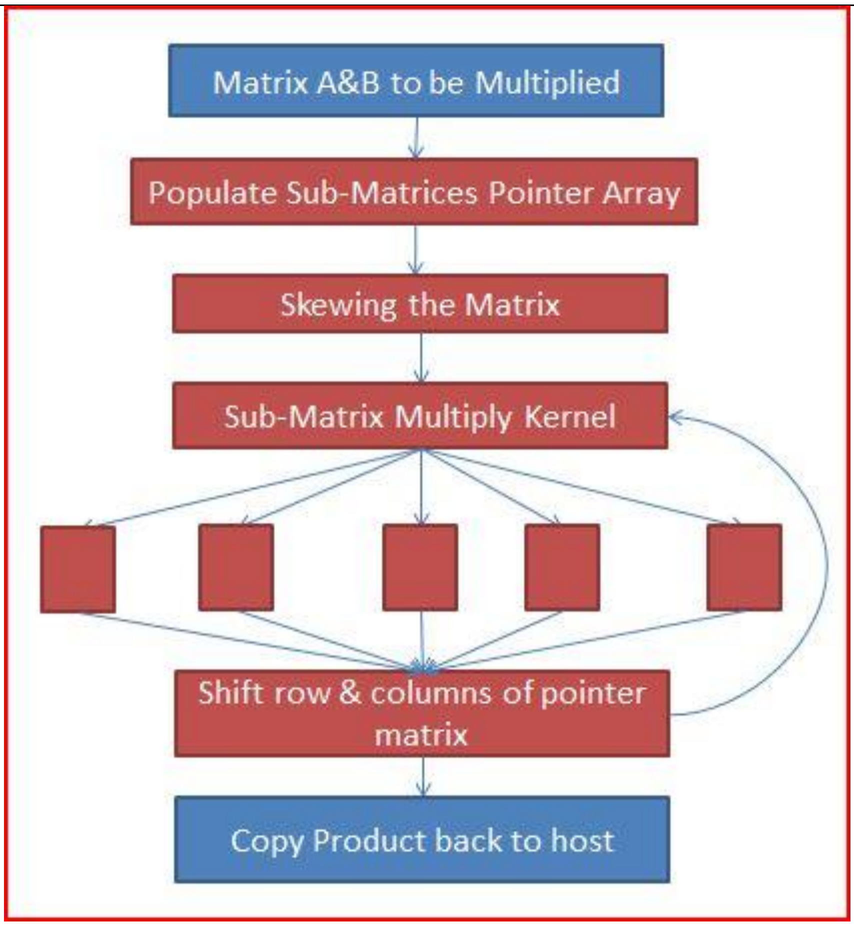

Matrix A&B to be Multiplied
Populate Sub-Matrices Pointer Array
Skewing the Matrix
Sub-Matrix Multiply Kernel
Shift row & columns of pointer matrix
Copy Product back to host

Figure 4.8: GP-GPU Implementation Of Cannon's Algorithm

4.5 Summary

In this chapter we introduced both case studies and discussed the parallel implementations for the GP-GPU platform including the program architecture used in both applications. The Room Assignment problem's GP-GPU implementation used simultaneous threads for multiple move evaluation in each temperature iteration and the GP-GPU parallelized search to select the best move. Our approach in the Room Assignment problem shows that the GP-GPU solution naturally moves towards a faster solution starting with equalized runtimes as compared to the serial version. Cannon’s Algorithm implementation used multiple threads as independent processing units to compute product-submatrices and uses a shifting pointer array to implement movement of sub-matrices from one thread in the processor grid to another. The next chapter will discuss the results obtained from both case studies.

CHAPTER FIVE

RESULTS AND ANALYSIS

In this chapter we discuss the results obtained for both case studies and provide an analysis of these results. Section 5.1 discusses the results from the Room Assignment problem including the basis for selecting a different cooling constant for the serial version. Section 5.2 presents the performance results for the GP-GPU implementation of Cannon’s Algorithm and contrasts it with the $O(n^3)$ x86 and MPI-based traditional implementations.

5.1 Room Assignment

The performance of the GP-GPU implementation is evaluated by comparing the improvement over the initial room assignments that was achieved by both versions (improvement ratio). As mentioned in section 4.2.2, the constants related to the cooling schedule were adjusted to make the runtime of the GP-GPU and the serial implementations comparable, 0.999 and 0.999999 respectively. Table 5.1 shows the runtimes of the serial version by increasing the cooling constant value (from 0.999 to 0.999999) for a problem size of 5000 people. From the table, the runtime with a cooling constant of 0.999999 is closest to the runtime of the GP-GPU implementation using a constant cooling constant of 0.999. With a cooling constant of 0.999, the GP-GPU runtime for problem size 5000is 273.27 seconds and it produces an improvement ratio of 171.77, 75% better than the serial version.

Table 5.1: Varying Cooling Constant for Serial Implementation of the Room Assignment Problem

Value of Cooling Constant for Serial Version	Runtime Duration for Serial Version (seconds)	Improvement Ratio achieved by Serial Version
0.999	0.27	20.67
0.9999	2.74	58.30
0.99999	27.37	96.77
0.999999	274.89	97.88

From Table 5.1 we see that if the cooling constant for the serial version was kept the same as the GP-GPU version (0.999) a performance comparison with the GP-GPU implementation would have been misleading because the GP-GPU version produces higher improvement ratio but also a longer runtime while the serial version produces a shorter runtime but also a lower improvement ratio. Further, even though there is not a significant increase in the improvement ratio between 0.99999 and 0.999999, this is a problem size specific issue. The same serial implementation for a problem size of 15000 people produces a higher improvement ratio (108.68 compared to 173.74) for 0.99999 versus 0.999999. Thus to increase the range of problem sizes comparable with this adjustment of the cooling constant, 0.999999 was selected for the serial implementation.

As discussed, the problem size of 5,000 people was used when equalizing the runtime of the two implementations; additional results were collected for problem sizes of 10,000, 15,000 and 20,000 people using the same cooling schedule constants. Both implementations use the `gettimeofday()` function to collect the runtime of the algorithm. For each problem size, ten dislike-coefficient files were generated and each file was run ten times resulting in 400 executions of both the GP-GPU and serial implementations.

To achieve the best improvement ratio for the room assignments, a very high initial temperature, 1.7x10^{308}, was given to both implementations of the algorithm. The program stops when the temperature variable (type: `double`) reaches the smallest positive value possible for that data type. Figures 5.1, 5.2, 5.3 and 5.4 show the average improvement ratios achieved by the parallel and the serial versions for each of the problem sizes.

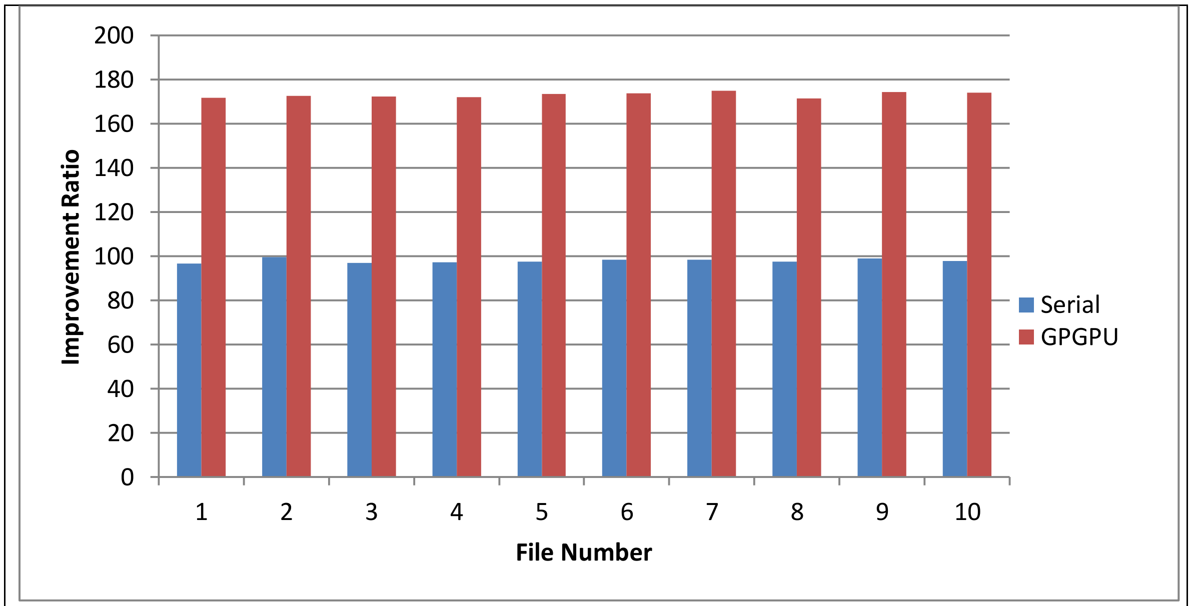


Figure 5.1: Average Improvement Ratios Achieved For 5000 people (Solution Quality)

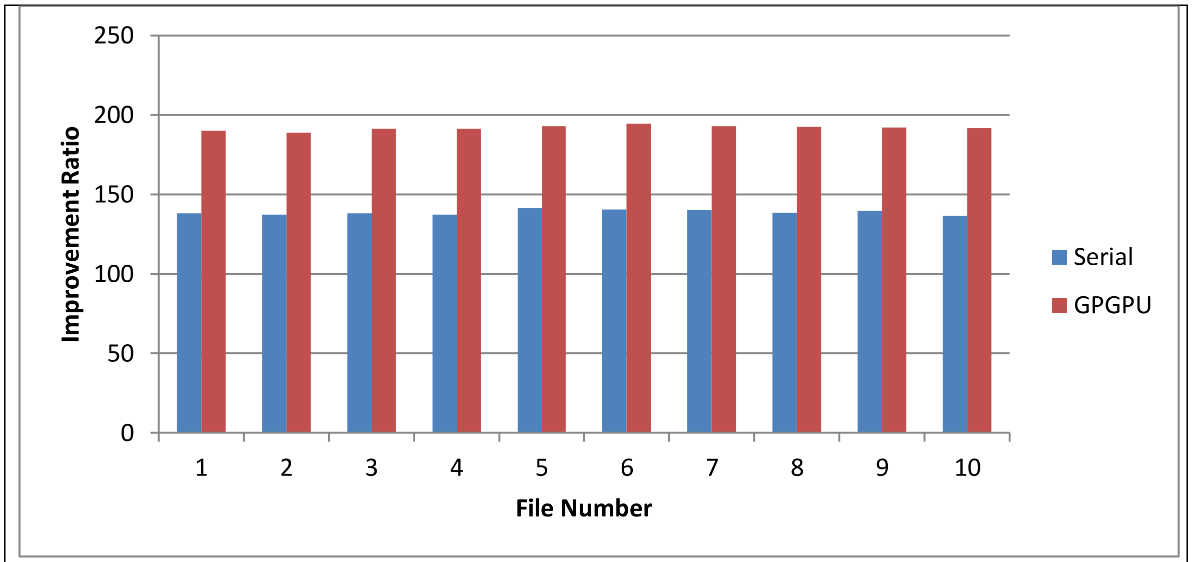


Figure 5.2: Average Improvement Ratios Achieved For 10000 People (Solution Quality)

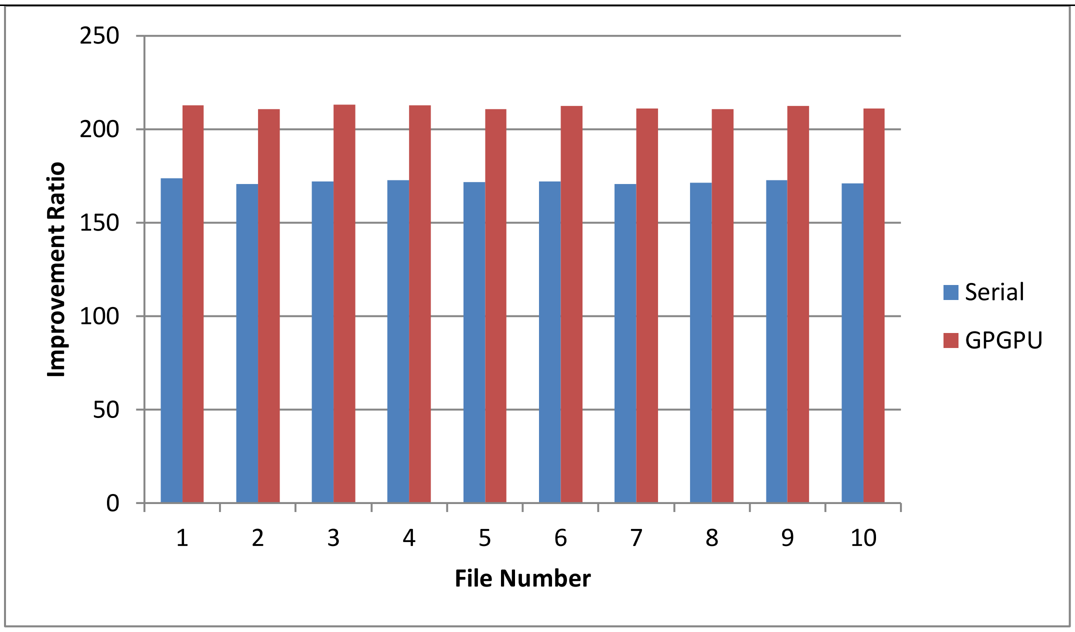


Figure 5.3: Average Improvement Ratios Achieved For 15000 people (Solution Quality)

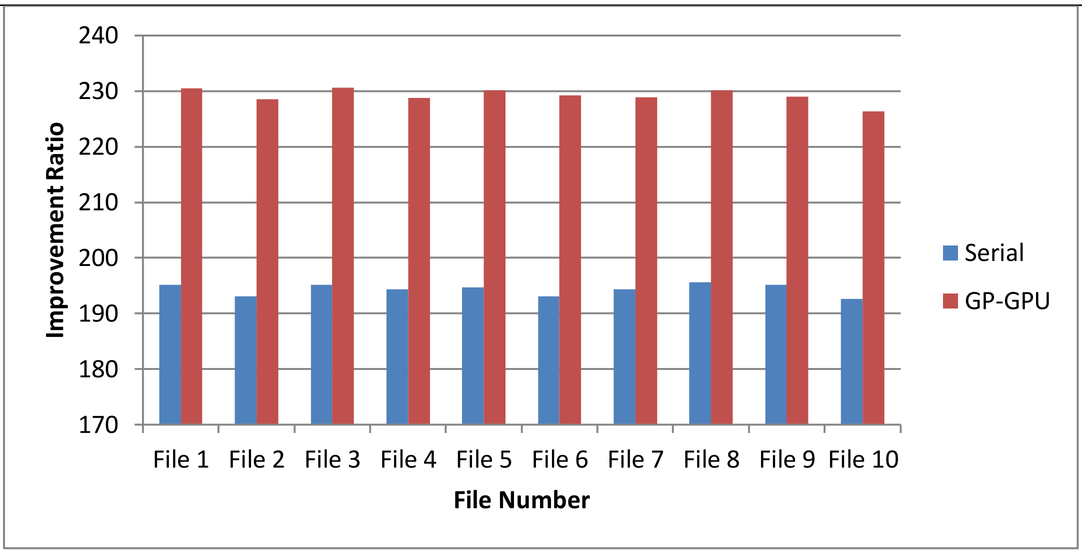


Figure 5.4: Average Improvement Ratios Achieved For 20000 people (Solution Quality)

While the execution time was equalized with a problem size of 5,000 people, the execution time difference between the serial and parallel implementations remains small for a problem size of 10,000 people as shown in Table 5.2. However, the performance of the GP-GPU implementation was progressively faster than the serial version for problem sizes 15000 and 20000. Table 5.2 shows the average runtimes for GP-GPU and serial versions for problem sizes 5000, 10000, 15000 and 20000 using file #1 for each problem size. It should be noted that file #1 is not the same file across different problem sizes because the files are generated independently for different problem sizes. The same trend in runtime difference is present from other files for each problem size. Figure 5.5 depicts the data from Table 5.1 graphically showing the difference between the serial and GP-GPU implementation average runtimes starting from problem size 5000 and how they diverge for the larger problem sizes.

Table 5.2: Average Runtimes Comparison between GP-GPU and Serial Version across Different Problem Sizes Using File #1 from Each Problem Size

Problem Size (Number of People)	File Used	Average runtime for Serial Version (seconds)	Average runtime for GP-GPU version (seconds)
5000	File #1 from problem size 5k	274.89	273.27
10000	File #1 from problem size 10k	299.46	299.62
15000	File #1 from problem size 15k	354.19	318.92
20000	File #1 from problem size 20k	432.89	327.14

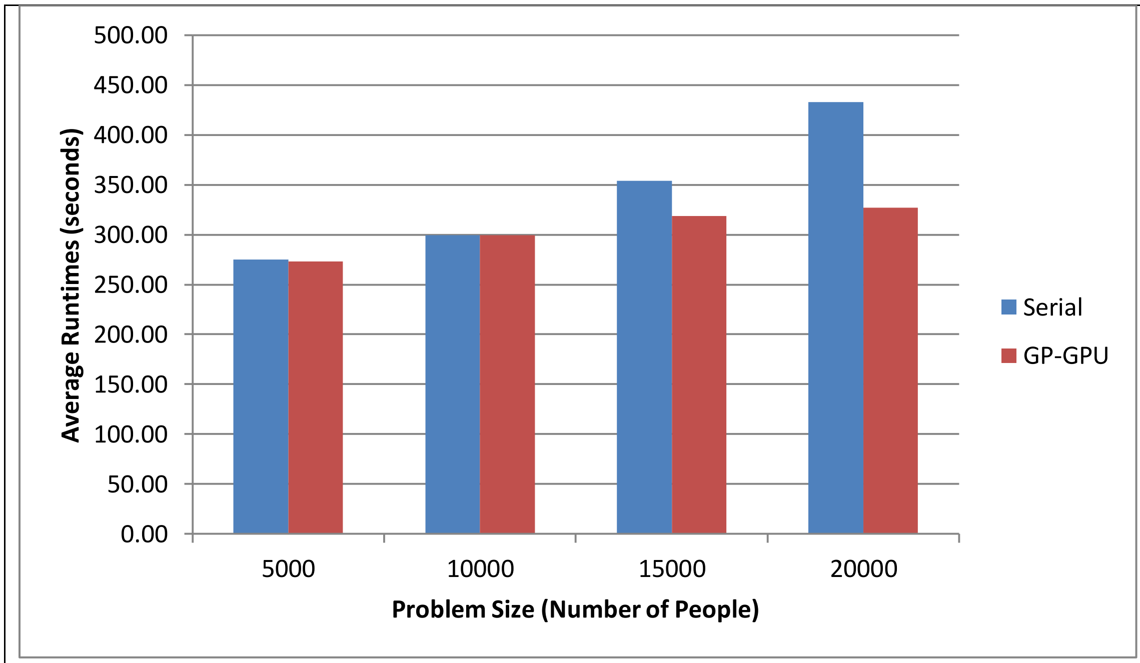


Figure 5.5: Average Runtimes (File #1 for Each problem size)

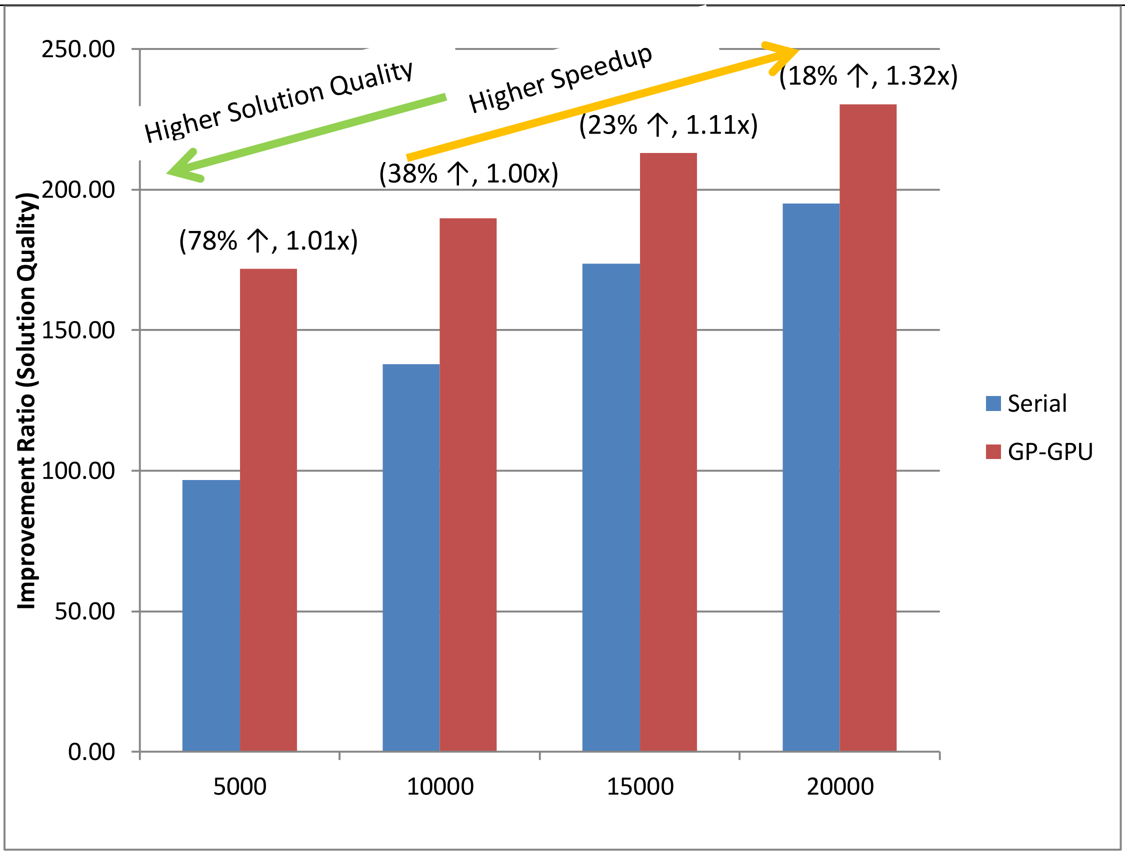


Figure 5.6: Relation Between Improvement Ratio and Performance Difference Comparing the GP-GPU and Serial implementations for Increasing Problem Sizes with File #1 from Each Size. Values in brackets show (i) The Percentage Difference In Solution Quality and (ii) the Speedup of the GP-GPU Implementation Over the Serial Implementation.

From Figure 5.6 we see that the GP-GPU solution is naturally moving towards a speedup with shorter runtimes than the serial version. Consider that at a problem size of 5000, the runtimes were equalized and the GP-GPU solution gave a 78% higher improvement ratio; and for a problem size of 20000 people, the GP-GPU solution gave an 18% higher improvement ratio 105 seconds faster than the serial implementation (1.3x speedup). Finally, the GP-GPU solution delivered a solution that was of higher quality than the serial version for all problem sizes.

Consider that the serial version of the Room Assignment problem goes through ~1.42×10^9 time steps, which remains constant as the problem size increases. Similarly, the GP-GPU version goes through ~1.42×10^6 time steps, which also remains constant with an increase in the problem size. Further, let '*x*' be the number of memory transactions performed by serial version per time step, which remains constant as the problem size increases. Similarly, let '*y*' be the number of memory transactions per time step by the GP-GPU version, which also remains constant as the problem size increases. Thus, for each implementation there is no increase in the number of computations or the number of memory accesses to be performed. The only difference as the problem size increases is an increase in the size of the data structures being accessed. Finally, consider Figure 5.5 which shows the average runtimes for both the serial and the GP-GPU versions with increasing problem size. As the problem size increases, the runtimes of the serial version are increasing at a much higher rate than the GP-GPU version. We surmise from this that the data distribution on the GP-GPU has more uniform access than the serial version, which is contributing as a significant factor in GP-GPU's increased speedup compared to the serial version with the increase in problem size.

5.2 Cannon's Algorithm

The runtime of Cannon's Algorithm on the GP-GPU was compared to a conventional O(n^3) matrix multiplication algorithm. Figures 5.7, 5.8, 5.9 and 5.10 show the speedup achieved with Cannon's Algorithm on the GP-GPU over the serial multiplication. The speedup is measured with varying number of threads for multiplying two matrices of a fixed size. Each thread computes a sub-matrix for the entire product matrix.

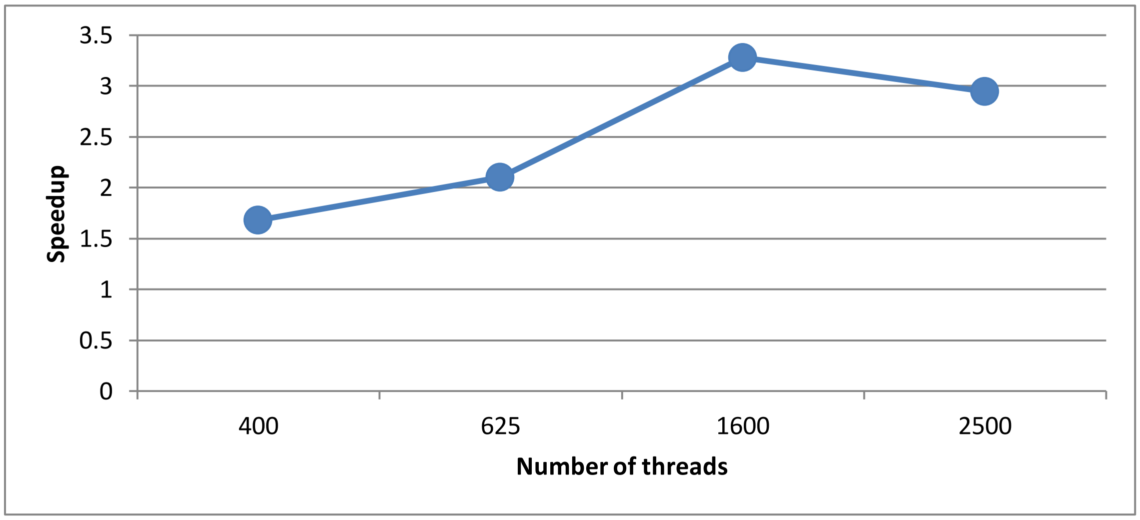


Figure 5.7: Speedup for Matrix Size: 1000x1000

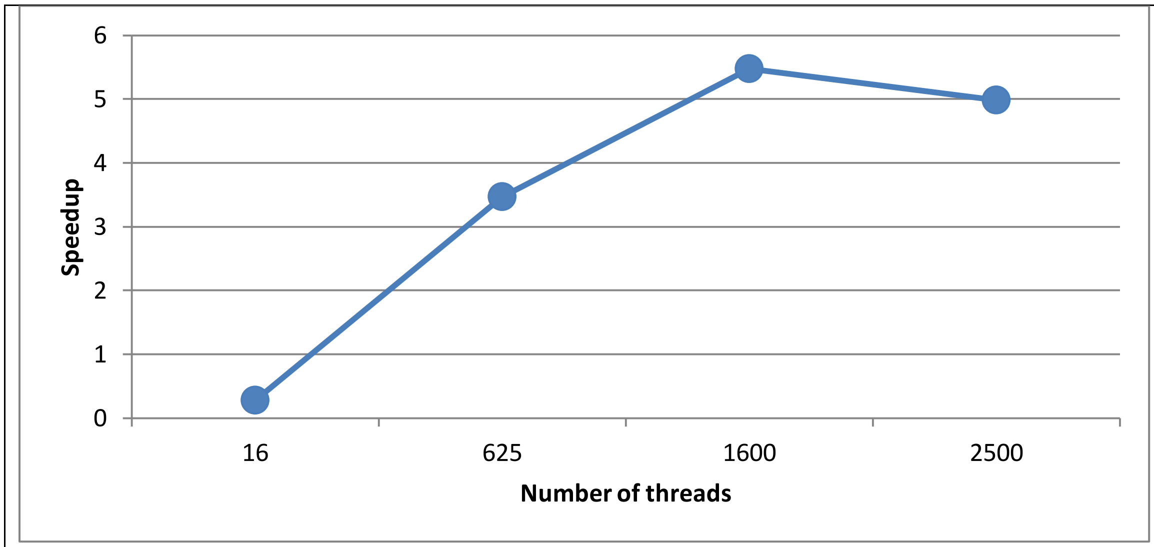


Figure 5.8: Speedup for Matrix Size: 2000x2000

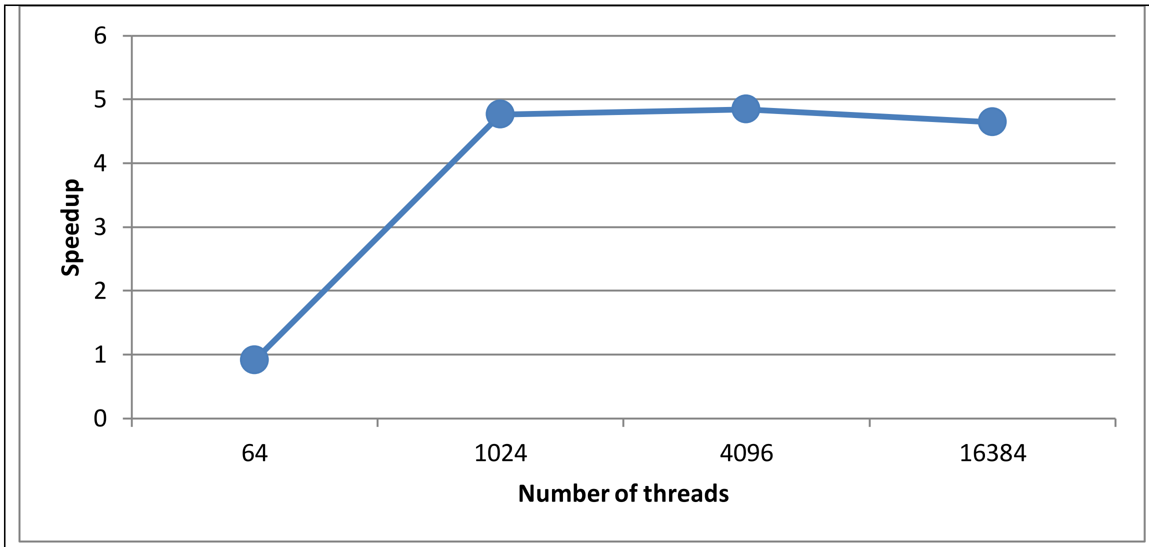


Figure 5.9: Speedup for Matrix Size: 4096x4096

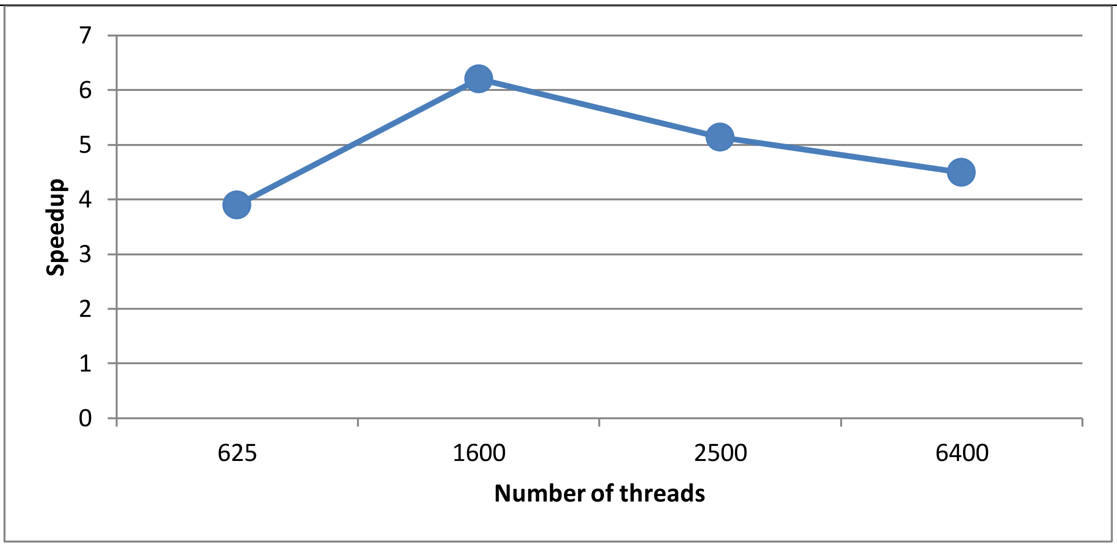


Figure 5.10: Speedup for Matrix Size: 8000x8000

In [28], the performance of a Cannon's algorithm MPI implementation is presented. A speedup of 150x with 64 processors is reported for a matrix size 4096x4096. In comparison, Cannon's Algorithm on the GP-GPU achieved a peak speedup of 4.8x with 4096 threads. One factor contributing to this performance gap is the difference in the cache size of the processing units for these implementations. To understand why cache size plays an important role in Cannon's Algorithm consider the code provided in Figure 5.11, which runs on the parallel processors used in Cannon's Algorithm. During each iteration of the outermost loop *i*, each element of sub-matrix *B* is read into the cache [1]. If the size of sub-matrix *B* is too large, then the elements of *B* that were read into the cache earlier are flushed, and they must be re-read on the next iteration of *i*. This overhead affects the performance of the code because of additional delay in fetching data from memory again when the cache is too small to hold the entire sub-matrix.

```
for (i=0; i<n; i++)
          for(j=0; j<n; j++)
          {
                    for(k=0; k<n; k++)
                    {
                              C[i][j]+=A[i][k]*B[k][j];
                    }
          }
```

Figure 5.11: Sample Code for Sub-Matrix Multiplication

The MPI implementation in [28] uses a quad core Intel Xeon 2.8 GHz processor, which has a L1 cache of 64 KB per core. This cache is divided into two equal parts: 32 KB for instruction cache and 32 KB for data cache. The GP-GPU implementation uses CUDA Streaming Processors as the executing units with a L1 cache of 64 KB for 32 Streaming Processors, because 32 streaming processors are grouped together to form a Streaming Multiprocessor (SM) and each SM has a L1 cache of 64 KB. Thus 32 streaming processors must share 64 KB. Additionally, the entire 64 KB is not available for caching purposes but rather it is split into two parts: one part of 16 KB and another of 48 KB. CUDA allows the programmer to configure which of these two parts should be used for L1 caching purpose and which should be used for shared memory purpose with the function `cudaFuncSetCacheConfig()`. To increase the performance on the GP-GPU, this function was configured to use 48 KB for caching purpose and 16 KB for shared memory. This optimization improved the speedup from 4.2x to 4.8x for a matrix size of 4096x4096 with 4096 threads. We also found that the speedup remained more or less constant for the same matrix size with 1024 threads. This is expected, since with fewer threads, the size of the sub-matrices operated on by a single thread increases and a small increase in cache size will not affect the run time in these cases. However, with a higher number of threads, the size of the sub-matrices operated on by each thread reduces and the same amount of cache size increase leads to more speedup. Therefore, even though the GP-GPU runs at 1.59 GHz with 512 CUDA cores, the difference in architecture, more specifically, the difference in cache size impacts the performance of the algorithm significantly.

5.3 Summary

This chapter presented results obtained from both case studies including discussion on both with varying problem sizes. The parallel version of the Room Assignment problem provided higher quality solutions than the serial version for all problem sizes tested. Up to 78% higher improvement ratio was achieved for the GP-GPU implementation over the serial version. The GP-GPU implementation exhibited a speedup over the serial version as the problem size increased. We attribute this to the data distribution on the GP-GPU being much more uniformly accessible than that on the serial version (x86) due to the memory partitioning of the GP-GPU. Cannon’s Algorithm on the GP-GPU achieved a peak speedup of 6.2x for a matrix size of 8000x8000. Its performance with matrix size 4096x4096 was contrasted with an MPI based Cannon’s Algorithm implementation. The next chapter offers some conclusions and outlines possible future work related to both case studies.

CHAPTER SIX

CONCLUSIONS AND FUTURE WORK

In this research, two application case studies were implemented on a GP-GPU platform, the Room Assignment problem solution with Simulated Annealing and Cannon's Algorithm for matrix-matrix multiplication. The performance characteristics of both applications were studied with varying problem sizes.

6.1 Conclusions

The Room Assignment Problem was implemented on the GP-GPU using CUDA and performance results were explored on this architecture. The GP-GPU architecture and implementation proved to be a suitable candidate to a previous implementation in [3], which was implemented on x86 processors. We could not find any published results in the literature for the Room Assignment problem with Simulated Annealing on a GP-GPU architecture and this is the first set of published results on the GP-GPU, to the best of our knowledge. The GP-GPU version was highly parallelized and outperformed the serial implementation in terms of the improvement ratio metric when both were allowed to execute for a comparable amount of runtime. The runtimes were made comparable for a problem size of 5000 by adjusting the cooling constants of both the serial and the parallel implementations to give comparable runtimes. Both versions were tested with these same constants for problem sizes 10000, 15000 and 20000 people. The GP-GPU version was able to achieve up to 78% higher improvement ratio over the sequential implementation in a comparable amount of runtime for a problem size of 5000 people. For the largest problem size tested, 20000 people, the GP-GPU solution provided up to 18% higher improvement ratio 105 seconds faster than the serial implementation, a speedup of 1.3x. We observe that our GP-GPU results are in agreement with expected Monte Carlo and Simulated Annealing-based solutions where higher quality of solutions is inversely related to runtime. The large number of parallel compute resources on the GP-GPU architecture allowed each parallel thread to generate and evaluate moves independently. Further, the availability of the CUDA *CURAND* library to generate parallel random numbers for random perturbations was also highly instrumental in the realization on the GP-GPU platform. Finally a GP-GPU parallelized search was used to search through these generated moves, select the best move, and apply it to the global configuration. The results from this implementation clearly establish the GP-GPU architecture as a suitable candidate for future problems that use Simulated Annealing techniques.

The second algorithm studied in this work was a GP-GPU implementation of Cannon's Algorithm. The GP-GPUs have reached a point where they offer a very high number of parallel compute cores and a large amount of device (global) memory. These characteristics make them an interesting candidate for cluster applications that have traditionally been implemented with MPI. In this regard, a common MPI application was implemented on the GP-GPU architecture to study how well it maps to the platform. The Cannon's Algorithm achieved a peak speedup of 6.2x over a $O(n^3)$ conventional matrix multiplication implementation. The GP-GPU implementation was also compared to a conventional MPI implementation of Cannon's Algorithm found in [28], which reported a speedup of up to 150x with 64 processors for matrix size 4096x4096. For the same matrix size, the GP-GPU implementation achieved a peak speedup of 4.8x. The wide gap in performance between the two architectures was concluded to be due to the small cache size of the GP-GPU architecture. The inference drawn from the results of Cannon's Algorithm was that while most algorithm implementations on an x86 architecture make memory management invisible to the programmer, to get the best results on the GP-GPU, it is essential to manually optimize memory access for the algorithm at each hierarchy of memory (i.e. device global memory, block shared memory, the L1 cache etc.). This need for manual control is mainly due to the GP-GPU architecture approach: very high ratio of compute-cores to memory and the fast memory resources are small and must be manually tuned.

6.2 Future Work

As a future work it would be interesting to implement the Room Assignment problem with OpenCL and compare its performance against the CUDA-based version. Additionally, it would be very insightful to study the performance obtained by evaluating more than one move in each parallel thread, in contrast to current implementation, where each thread is evaluating a single move. Another possible future work, would be to quantify how the quality of solution varies when the runtimes are equalized for every problem size, in contrast to our current implementation where we explored the performance of GP-GPU solution by keeping the cooling constant fixed. It would also be interesting to compare the performance of the GP-GPU version against a similar, highly-parallelized version using MPI.

For Cannon's Algorithm, it can also be implemented with OpenCL and run on different GP-GPUs (i.e. those from AMD) to compare and characterize performance versus the x86 platform. Intel's Many Integrated Cores (MIC) co-processors would also be an interesting architecture to investigate with this algorithm and compare performance against NVIDIA's GP-GPUs, since Intel claims the compute cores on this new platform to be very similar to Intel Processor cores that most developers are familiar with. Another possible future investigation would compare the performance of Cannon's algorithm where the threads that are organized in a block, cooperate among themselves, to better utilize the cache available to them. For example, using a block multiply scheme that is optimized based on the cache size.

6.3 Contributions

This research presented a solution for the Room Assignment problem on the GP-GPU architecture based on Simulated Annealing. The performance results from Cannon's algorithm on GP-GPUs highlighted the inherent architectural differences between the two platforms (x86 and GP-GPUs) and quantified the difference it can make in the performance of Cannon's algorithm.

APPENDICES

Appendix A

Room Assignment Problem GP-GPU and Serial Runtime Results

Table A-1: Results from Serial Implementation for Problem Size-5000 People

File #	Avg. Improvement Ratio	Improvement Std. Deviation	Avg. Runtime (s)	Runtime Std. Deviation (s)
File 1	96.75	0.92	274.89	0.34
File 2	99.46	1.94	275.09	0.33
File 3	96.77	1.44	275.60	1.93
File 4	97.25	1.41	275.10	0.44
File 5	97.63	1.57	274.96	0.33
File 6	98.24	1.70	275.77	1.82
File 7	98.49	2.10	275.03	0.35
File 8	97.36	1.69	275.03	0.39
File 9	98.90	0.82	274.97	0.15
File 10	97.84	2.04	274.89	0.14

Table A-2: Results from GP-GPU Implementation for Problem Size-5000 People

File #	Avg. Improvement Ratio	Improvement Std. Deviation	Avg. Runtime (s)	Runtime Std. Deviation (s)
File 1	171.77	1.92	273.2735	0.20
File 2	172.58	2.65	273.39	0.23
File 3	172.26	3.18	273.44	0.31
File 4	171.95	3.13	273.42	0.33
File 5	173.34	2.48	273.56	0.36
File 6	173.58	2.77	273.48	0.30
File 7	174.83	3.12	273.38	0.28
File 8	171.42	2.29	273.53	0.23
File 9	174.36	2.32	273.40	0.28
File 10	173.92	2.74	273.44	0.31

Table A-3: Results from Serial Implementation for Problem Size-10000 People

File #	Avg. Improvement Ratio	Improvement Std. Deviation	Avg. Runtime (s)	Runtime Std. Deviation (s)
File 1	137.96	1.98	299.46	0.68
File 2	137.28	2.26	298.84	0.39
File 3	137.96	1.56	298.60	0.59
File 4	137.26	1.15	298.60	0.75
File 5	141.29	1.50	299.27	1.81
File 6	140.41	1.19	298.83	0.69
File 7	140.14	2.02	298.57	0.63
File 8	138.71	0.94	299.03	0.74
File 9	139.58	1.72	298.93	0.62
File 10	136.46	1.15	298.62	0.60

Table A-4: Results from GP-GPU Implementation for Problem Size-10000 People

File #	Avg. Improvement Ratio	Improvement Std. Deviation	Avg. Runtime (s)	Runtime Std. Deviation (s)
File 1	189.94	1.93	299.6212	0.47
File 2	188.97	2.26	299.37	0.29
File 3	191.11	1.57	299.67	0.35
File 4	191.15	2.39	299.48	0.26
File 5	192.71	3.28	299.35	0.33
File 6	194.63	2.92	299.45	0.44
File 7	192.95	1.96	299.55	0.63
File 8	192.47	2.77	299.46	0.39
File 9	191.91	1.97	299.70	0.26
File 10	191.57	2.64	299.36	0.25

Table A-5: Results from Serial Implementation for Problem Size-15000 People

File #	Avg. Improvement Ratio	Improvement Std. Deviation	Avg. Runtime (s)	Runtime Std. Deviation (s)
File 1	173.74	2.13	354.19	1.19
File 2	170.56	1.17	353.42	1.73
File 3	172.23	2.39	354.61	1.65
File 4	172.65	1.64	355.83	2.49
File 5	171.57	1.75	354.80	2.17
File 6	172.09	1.79	354.49	1.79
File 7	170.80	1.67	354.81	1.52
File 8	171.28	1.26	354.96	1.46
File 9	172.65	1.84	355.41	1.36
File 10	171.17	1.01	355.94	1.80

Table A-6: Results from GP-GPU Implementation for Problem Size-15000 People

File #	Avg. Improvement Ratio	Improvement Std. Deviation	Avg. Runtime (s)	Runtime Std. Deviation (s)
File 1	212.95	1.24	318.92	0.25
File 2	210.70	1.62	318.64	0.37
File 3	212.96	1.73	318.70	0.40
File 4	212.83	2.04	318.66	0.36
File 5	210.81	1.87	318.65	0.40
File 6	212.34	2.50	318.76	0.45
File 7	211.09	1.80	318.57	0.43
File 8	210.84	1.86	318.79	0.37
File 9	212.39	2.11	318.45	0.39
File 10	211.06	2.65	318.67	0.33

Table A-7: Results from Serial Implementation for Problem Size-20000 People

File #	Avg. Improvement Ratio	Improvement Std. Deviation	Avg. Runtime (s)	Runtime Std. Deviation (s)
File 1	195.18	1.88	432.89	1.14
File 2	193.14	1.82	437.46	6.08
File 3	195.12	1.74	439.09	4.48
File 4	194.31	1.06	434.67	3.83
File 5	194.72	1.73	435.16	2.15
File 6	193.08	1.58	433.30	1.50
File 7	194.41	1.87	433.75	1.12
File 8	195.62	1.69	435.64	2.52
File 9	195.20	2.11	435.72	2.52
File 10	192.60	1.37	435.26	1.46

Table A-8 Results from GP-GPU Implementation for Problem Size-20000 People

File #	Avg. Improvement Ratio	Improvement Std. Deviation	Avg. Runtime (s)	Runtime Std. Deviation (s)
File 1	230.44	1.66	327.14	0.26
File 2	228.57	1.40	326.97	0.28
File 3	230.55	1.54	327.37	0.53
File 4	228.78	1.47	328.04	0.39
File 5	230.11	1.88	327.91	0.36
File 6	229.21	2.36	328.07	0.30
File 7	228.84	1.97	327.90	0.31
File 8	230.12	2.64	328.00	0.30
File 9	228.99	2.00	327.60	0.36
File 10	226.39	2.27	326.89	0.30

Appendix B

Results from Cannon's Algorithm Implementation

Table B-1: Conventional O(n^3) Matrix Multiplication Runtimes	
Matrix Size (N) for NxN	Average Runtime (seconds)
1000	5.37
2000	66.17
4096	640.56
8000	4810.56

Table B-2: Runtime Results from Cannon's Algorithm Implementations with Optimized Cache Size for Matrix Size 1000x1000

Number of Threads	Average Runtime(seconds)	Speedup
400	3.20	1.68
625	2.56	2.10
1600	1.64	3.28
2500	1.82	2.95

Table B-3: Runtime Results from Cannon's Algorithm Implementations with Optimized Cache Size for Matrix Size 2000x2000

Number of Threads	Average Runtime(seconds)	Speedup
16	237.98	0.28
625	19.05	3.47
1600	12.09	5.47
2500	13.28	4.98

Table B-4: Runtime Results from Cannon's Algorithm Implementations with Optimized Cache Size for Matrix Size 4096x4096

Number of Threads	Average Runtime(seconds)	Speedup
64	705.09	0.91
1024	134.41	4.77
4096	132.25	4.84
16384	138.07	4.64

Table B-5: Runtime Results from Cannon's Algorithm Implementation with Optimized Cache Size for Matrix Size 8000x8000

Number of Threads	Average Runtime(seconds)	Speedup
625	1234.27	3.90
1600	775.94	6.20
2500	936.76	5.14
6400	1069.79	4.50

Table B-6: Runtime Results from Cannon's Algorithm Implementation without Optimized Cache Size for Matrix Size 1000x1000

Number of Threads	Average Runtime(seconds)	Speedup
400	3.22	1.67
625	2.52	2.13
1600	1.70	3.16
2500	2.03	2.64

Table B-7: Runtime Results from Cannon's Algorithm Implementation without Optimized Cache Size for Matrix Size 2000x2000

Number of Threads	Average Runtime(seconds)	Speedup
16	237.97	0.28
625	19.24	3.44
1600	12.87	5.14
2500	15.30	4.33

Table B-8: Runtime Results from Cannon's Algorithm Implementation without Optimized Cache Size for Matrix Size 4096x4096

Number of Threads	Average Runtime(seconds)	Speedup
64	705.05	0.91
1024	133.36	4.80
4096	149.63	4.28
16384	143.76	4.46

Table B-9: Runtime Results from Cannon's Algorithm Implementation without Optimized Cache Size for Matrix Size 8000x8000

Number of Threads	Average Runtime(seconds)	Speedup
625	1277.00	3.77
1600	821.99	5.85
2500	1047.72	4.59
6400	1154.04	4.17

References

[1] Quinn, M. J., *Parallel Programming in C with MPI and OpenMP*, McGraw Hill, 2004.

[2] Kirckpatrick, S., C. Gelatt, and M. Vecchi, "Optimization by Simulated Annealing," *Science*, Vol. 220, Issue 4598, pp.671-680, 1983.

[3] Lazarova, M., "Parallel Simulated Annealing for Solving the Room Assignment Problem on Shared and Distributed Memory Platforms," *CompSysTech '08 Proceedings of the 9th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing*, ACM New York, NY, USA, 2008.

[4] Martinez-Alfaro, H., J. Minero, G. E. Alanis, N. A. Leal, and I. G. Avila, "Using Simulated Annealing to solve the Classroom Assignment Problem," *Proceedings of International Conference on Intelligent Systems Technologies*, Cancun, Mexico, 1996.

[5] Stivala, A. D., P. J. Stuckey, and A. I. Wirth, "Fast and Accurate protein substructure searching with simulated annealing and GPUs," *BMC Bioinformatics*, Vol. 11, 2010.

[6] Han, Y., S. Roy, and K. Chakraborty, "Optimizing Simulated Annealing on GPU: A Case Study with IC Floorplanning," *12th International Symposium on Quality Electronic Design*, Santa Clara, CA, pp.1-7, 2011.

[7] Lee, H.-J., J. P. Robertson, and J. A. Fortes, "Generalized Cannon's Algorithm for Parallel Matrix Multiplication," *ICS '97 Proceedings of the 11th international conference on Supercomputing*, ACM New York, NY, USA, 1997.

[8] Alpatov, P., G. Baker, C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, "PLAPACK: Parallel Linear Algebra Package," *Supercomputing Conference 1997*, pp. 29, 1997.

[9] Alqadi, Z. A., M. Aqel, and I. M. El Emary, "Performance Analysis and Evaluation of Parallel Matrix Multiplication Algorithms," *World Applied Sciences Journal*, Vol. 5, pp.211-214, 2008.

[10] Jaeyoung, C., D. W. Walker, and J. J. Dongarra, "Level 3 BLAS for distributed memory concurrent computers," *CNRS-NSF collaboration workshop on environments and tools for parallel scientific computing*, St. Hilaire du Touvet (France), 1992.

[11] Cannon, L. E., "A Cellular Computer to Implement the Kalman Filter Algorithm," *Ph.D. Thesis*, Montana State University, 1969.

[12] Agarwal, R. C., S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM Journal of Research and Development*, Vol. 39, Issue 5, pp. 575, 1995.

[13] Jaeyoung, C., D. W. Walker, and J. J. Dongarra, "Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers," *Concurrency: Practice and Experience*, Vol. 6, Issue 7, pp. 543-570, 1994.

[14] Chtchelkanova, A., J. Gunnels, G. Morrow, J. Overfelt, and R. A. van de Geijn, "Parallel Implementation of BLAS: General Techniques for Level 3 BLAS," *Concurrency: Practice and Experience*, Vol. 9, No. 9,pp.837-857, 1997.

[15] Choi, J., Dongarra, J. J., R. Pozo, and D. W. Walker, "ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers," *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, USA, 1992.

[16] Ismail, M. A., D. Mirza, and D. Altaf, "Concurrent Matrix Multiplication on Multi-Core Processors," *International Journal of Computer Science and Security*, Vol-5, Issue-2, pp.168-297, 2011.

[17] *NVIDIA CUDA C Programming Guide*, Ver-4, 2011.

[18] *CUDA Thrust Library*, http://code.google.com/p/thrust/.

[19] Gale, D., and L. Shapley, "College Admission and the Stability of Marriage," *The American Mathematical Monthly,* Vol. 69, No. 1 (Jan., 1962), pp. 9-15, 1962.

[20] Irving, R. W., "An Efficient Algorithm for the "Stable Roommates" Problem," *Journal of Algorithms*, Vol. 6, Issue 4, pp.577-595, 1985.

[21] Metropolis, N., A. W. Rosenbluth, M. N. Rosenbluth, and A. H. Teller, "Equation of State Calculations by Fast Computing Machines," *The Journal of Chemical Physics,* Vol. 21, No. 6, 1953.

[22] Nishimura, T., and M. Matsumoto, http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/CODES/mt19937ar.c. Retrieved from *http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html*.

[23] Landau, R. H., and M. J. Paez, *Computational Physics Problem Solving with Computers*, pp.303, John Wiley and Sons, 1997.

[24] *NVIDIA CURAND Library Guide*, 2010.

[25] Harris, M., "Optimizing Parallel Reduction in CUDA," *http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf*.

[26] *NVIDIA CUDA CUBLAS Library*, http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/CUBLAS_Library_3.1.pdf, 2010.

[27] Nath, R., S. Tomov, and J. Dongarra, "A Improved MAGMA GEMM for Fermi GPUs," *International Journal of High Performance Computing Applications*, Vol. 24, No. 4, 2010.

[28] Bartlett, S., "Comparison of Parallel Programming Paradigms," *B.Sc. Thesis*,University of Bath, 2010.

[29] Cannon Algorithm Schematic, *http://www.cs.berkeley.edu/~demmel/cs267/lecture11/lecture11.html*.