ABSTRACT

Scientific computation requires a great amount of computing power especially in floating-point operation but a high-end multi-cores processor is currently limited in terms of floating point operation performance and parallelization. Recent technological advancement has made parallel computing technically and financially feasible using Compute Unified Device Architecture (CUDA) developed by NVIDIA. This research focuses on measuring the performance of CUDA and implementing CUDA for a scientific computation involving the process of porting the source code from CPU to GPU using direct integration technique. The ported source code is then optimized by managing the resources to achieve performance gain over CPU. It is found that CUDA is able to boost the performance of the system up to 69 times in Parboil Benchmark Suite. Successful attempt at porting Serpent encryption algorithm and Lattice Boltzmann Method provided up to 7 times throughput performance gain and up to 10 times execution time performance gain respectively over the CPU. Direct integration guideline for porting the source code is then produced based on the two implementations.

ACKNOWLEDGEMENTS

My praise to Allah the Almighty, with His willing, I was able to complete my Final Year Project (FYP) in Universiti Teknologi PETRONAS.

The accomplishment of this project would have been impossible without the help of these people:

- Dr. Fawnizu Azmadi Hussin, my project supervisor for his valuable insight, guidance and support throughout the completion the project.
- Dr. Noohul Basheer Zain Ali, panel for my FYP seminar and Interim oral presentation. His comments and suggestions have allowed further improvement for the project.
- My family, for their support, giving me motivation and advices, understanding, and their encouragement for me to perform better.
- My fellow friends and individuals who were supportive throughout the project period.

Thank you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ALU	Arithmetic Logic Unit
API	Application Programming Interface
CUBLAS	Compute Unified Basic Linear Algebra Subprogram
CUFFT	Compute Unified Fast Fourier Transform
CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General-Purpose computing on Graphics Processing Unit
FFT	Fast Fourier Transform
FLOP/s	Floating Point Operation Per Second
GB/s	Gigabyte Per Second
GCC	GNU Collection Compiler
LB	Lattice Boltzmann
NVCC	NVIDIA CUDA Compiler
ICC	Intel C/C++ Compiler
PCI-E	Peripheral Component Interconnect Express
RHEL	Red Hat Enterprise Linux
SKU	Stock Keeping Unit
SMs	Streaming Multiprocessors
SPMD	Single Program Multiple Data

CHAPTER 1
INTRODUCTION

1.1 Background of Study

Compute Unified Device Architecture (CUDA) is an architecture designed by NVIDIA Corporation for General Purpose Computing using Graphic Processor Unit (GPGPU), the term which become increasingly popular as the trend grows in year 2002 [1]. In the early stage of GPGPU, one of the popular known methods for computation is using OpenGL. This method requires the algorithm to be reconstructed to match graphic processes (i.e. using textures and such) and constrained to a lot of limitations such as access to memory and data type limitation to only floating-point.

The advent of CUDA was in the November 2006, introduced to the public with the G80 architecture which eventually entered the mainstream later. A year later, some computational performance measurement results based on CUDA have been released and several more computational works has been ported to be performed by CUDA GPU instead of the Central Processing Units (CPU). These efforts were driven by the trend of GPU that follows Moore's Law growth faster than CPU. The fact that GPU exists as commodity hardware also became one of the motivation in pursuing the GPGPU interest.

CUDA-enabled graphic cards posses a higher computational performance (measured in FLOP/s) and higher bandwidth (measured in GB/s) when compared to CPU [2]. Although still to be further developed, early results suggested CUDA performance is indeed promising. The pinnacle of CUDA is demonstrated with the release of NVIDIA Tesla Personal Supercomputer which based on Tesla accelerator cards (similar with CUDA GPU in fact the origin of CUDA development [2]) and also the latest development of “Tsubame” supercomputer using CUDA GPUs which was ranked 29th fastest in the world [3].

1.2 Problem Statement

The advancement of technology has pushed scientific computing even further compared to a few years back as more theories are developed and tested using simulation and numerical computation. This requires a great amount of floating-point operations and parallelization to produce more reliable results over time. Mainstream CPU in personal computers mostly lack of the floating-point computation performance since the architecture focuses on integer performance for mass consumer computing.

There are a number of ways to overcome this situation which include using parallel computing, grid computing or distributed computing. Each of them might be costly or not practical to begin with, for example; parallel computing with supercomputer is highly expensive and maintaining a vast network of grid computers consumes resources and gathering data from distributed computing is rather tedious and complex. Hence, CUDA was introduced to improve the floating-point operation and parallelism of a personal computer given the overall specifications is met within the feasible cost.

Currently, there are not many applications that use CUDA that could be acquired out of the box. Many applications would require the source code to be ported and compiled using CUDA compiler. The process of porting of source code could sometimes produce inefficient code that performs worse than the CPU counterpart or only little performance gain. General guidelines are needed for the process of porting the source code efficiently with considerable performance gain without much time spent for the process.

1.3 Objectives and Scope of Study

The project objectives are to measure the CUDA performance to demonstrate the integer and floating-point operation using parallelization capabilities based on general application and scientific computations. Comparisons are made between CPU and CUDA computing performance. The final objectives of the project are to implement and study the effect of selected algorithms for integer and floating-point processes on CUDA. At the end of the project, we will come up with general guidelines to perform porting of the source code.

The scope of study covers general review of the architecture that contributes to the parallelism effect on performance, integer performance, floating-point performance and analysis of benchmark results. All of these are generally the fundamentals in understanding CUDA for successful implementation and occupy the time given for the project accordingly. As the project progress, the study is focused on source code optimization to increase performance gain.

CHAPTER 2
LITERATURE REVIEW

2.1 Computing with CUDA

Parallelization is not something new in the computer system but the implementations during the earlier time were only widely used at higher scale (i.e. supercomputer). Farber [4] explained much about CUDA based on his experience working with supercomputer with a larger scale of parallelization in national laboratories. CUDA provides parallelization with scalability, making it more attractive to programmers. According to [2] and [4], the programmers have the ability to program CUDA in high level language such as C, C++, Python and other high level languages. The great performance are also achieved since the compiler directly compile the source code to the CUDA device thus avoiding the performance overhead of Application Programming Interface (API).

Since CUDA has been primarily used for computation, it is very closely related to General-Purpose computing on Graphics Processing Unit (GPGPU). Owens *et al.* [5] describes the detail analysis of current GPGPU trend in architecture and also implementation. The architecture of CUDA GPU based on G80 architecture is reviewed to shed lights on multiprocessing and stream processing of current GPU. The paper [5] also explains the software performance libraries implementation (e.g. in the case of CUDA, CUBLAS and CUFFT libraries) and also the kernel performance which play a great role in CUDA performance.

Parallelization in handling multiple data by program benefits greatly when using CUDA. The GPU architecture is designed primarily to process data instead of data caching and flow control compared to CPU. Figure 1 illustrates the general architecture design of a CPU and GPU from [2]. GPU has a lot more Arithmetic Logic Units (ALUs) compared to a CPU; however GPU has limited amount of Control and Cache. The GPU architecture design also stresses on high throughput which uses data parallelization.

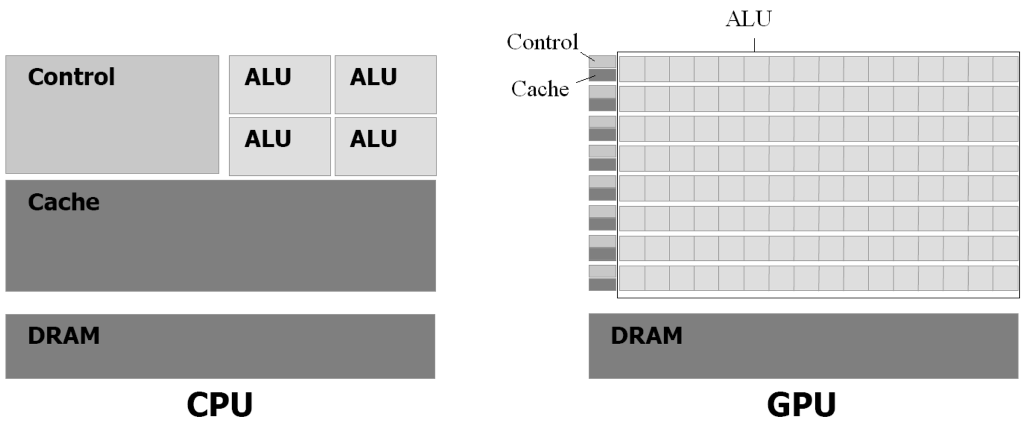


Figure 1: General illustration of CPU and GPU architecture

For the program to be able to handle multiple data in parallel, the programmer has to make the program initialize the kernel which runs on multiple threads at any given time. The program model is similar to Single Program Multiple Data (SPMD) model, with added advantage of scalability [6]. Figure 2 illustrates a program flow in kernel block initialization and scaling of the number of threads used according to the block specification. Each block can have a number of threads assigned to it in either one-dimension or up to three-dimension. Threads in the block will continue to run until the function for the threads to be synchronized is called. At that point, advanced threads will hold and wait for other threads to finish execution until the thread synchronization point.

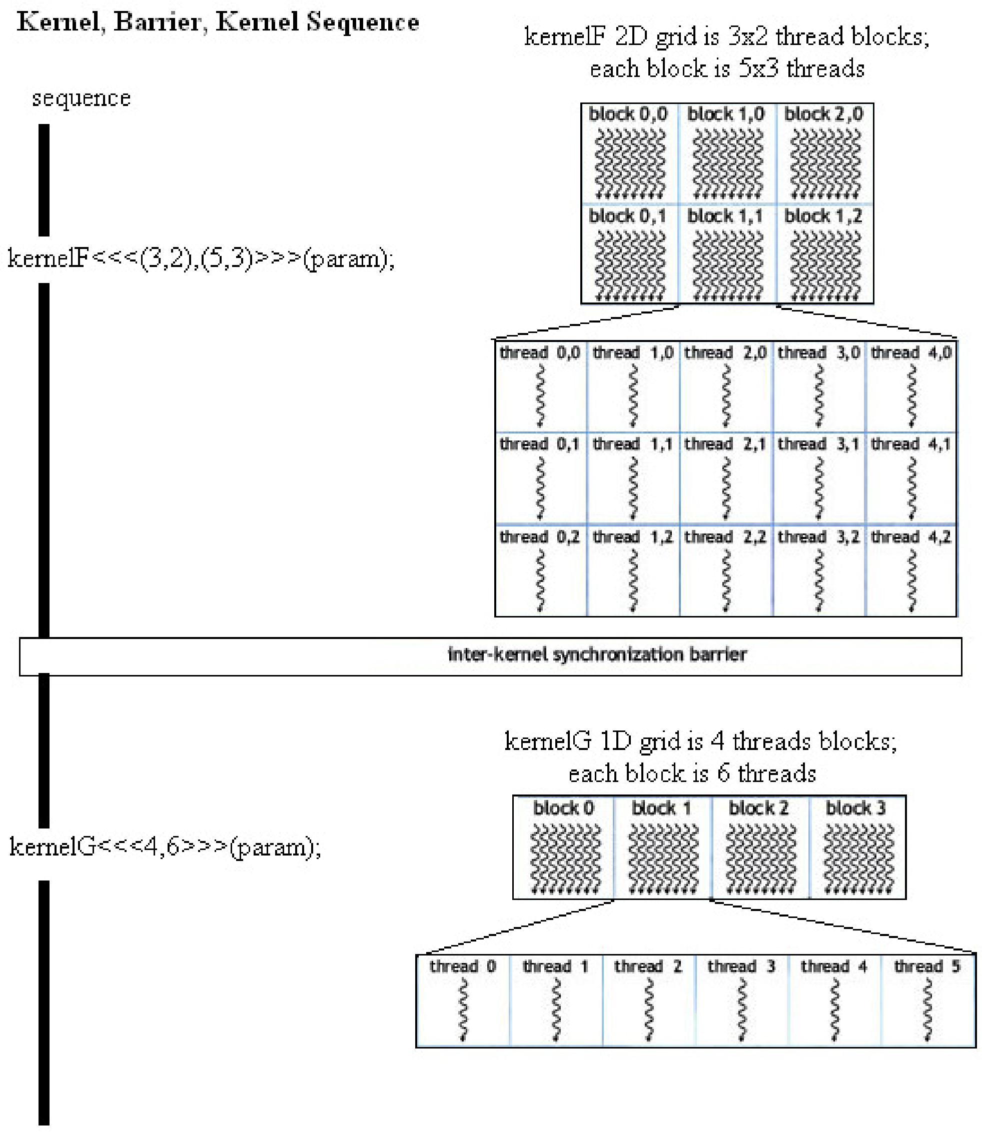


Figure 2: Diagram of Single Program Multiple Data (SPMD) kernel

CUDA function is defined as a kernel that is called by specifying the number of thread blocks, the number of threads per block and the parameters needed for the function as shown in Figure 2. Each thread executes the function in parallel and can communicate with other thread in the same thread block. As the threads are organized into thread blocks, the thread blocks are controlled by the Streaming Multiprocessors. Each thread block has a limited 16KB of Shared Memory for faster memory access compared to Device Memory and also used for communication between threads in the same block. Thread management is important to utilize the limited 16KB of Shared Memory to optimize memory access.

CUDA process flow is fairly straight-forward. Page-locked memory buffer is allocated in the host memory for faster memory transfer between host memory and device memory [2]. After the data transfer is done, the CPU will send instructions to the GPU for the program execution. The same program is executed on all of the threads inside the GPU. After all of the threads are synchronized, the data is transferred from the device memory back to the host memory. All of the process is summarized in Figure 3.

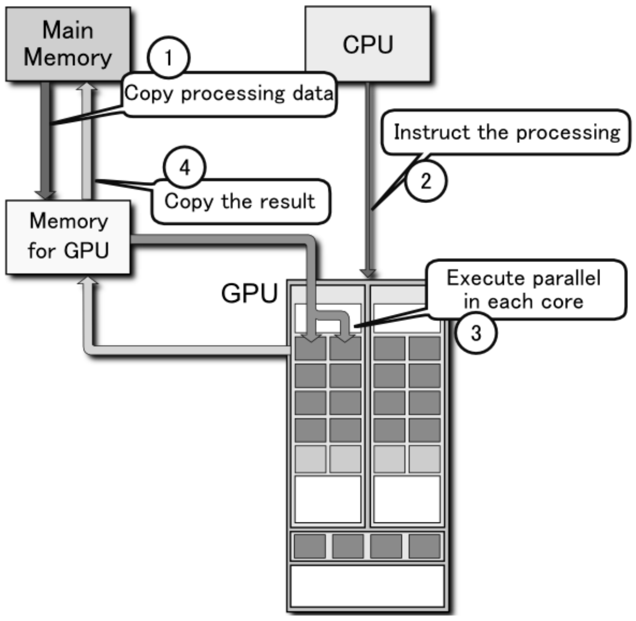


Figure 3: CUDA Process Flow.

Paper produced for performance study, [7]; makes use of the GPU design to improve the performance of general applications. Che *et al.* [7] compared CUDA with single-thread and multi-thread application executed on the multi-core CPU. The performance comparisons are given in speedup of CUDA over CPU. The authors also reported that a single graphic card was able to gain speedup over two dual-core high-end CPUs. Some of the general applications only require little or no optimization when ported from CPU to CUDA and already shown considerable speedup. This work has become motivation as to pursue parallel computing using GPU instead of multi-core CPU.

2.2 Performance Benchmark

In order to make sure the term “performance” is not loosely used throughout the project, a computer system text book is used for reference. Hennesy *et al.* [8] defined performance according to the execution time and also the number of instructions per second. In this research, the scoring system is also based on the execution time and normalized against processor benchmark score using basic compiler (i.e. GNU C Compiler). Regardless of how the scoring system is implemented, it should give an indication of the relative performance.

While there is no specific standard guideline for computer performance benchmarking, the obvious rule is that the benchmark must pass the output comparison for the executed benchmark programs. Some of the optimization flags of the compiler affect the accuracy of the output from the program. Our research focuses more on the floating-point benchmarks which is the data type commonly used for scientific calculation.

Parboil Benchmark Suite is one of the benchmarks that measure and can be used to compare both CPU and GPU performance [9]. The benchmark suite provided source codes, namely Base (basic source code with no optimization), CPU (optimized for processor) and CUDA (CUDA source code). Table 1 summarizes the benchmark programs in the suite.

Table 1: Parboil Benchmark Suite [9]

Application		Description
MRI-Q	Magnetic Resonance Imaging Q	Computation of a matrix Q, representing the scanner configuration, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.
MRI-FHD	Magnetic Resonance Imaging FHD	Computation of an image-specific matrix F^Hd, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.
CP	Coulombic Potential	Computes the coulombic potential at each grid point over on plane in a 3D grid in which point charges have been randomly distributed. Adapted from 'cionize' benchmark in VMD.
SAD	Sum of Absolute Differences	Sum of absolute differences kernel, used in MPEG video encoders. Based on the full-pixel motion estimation algorithm found in the JM reference H.264 video encoder.
TPACF	Two Point Angular Correlation Function	TPACF is an equation used here as a way to measure the probability of finding an astronomical body at a given angular distance from another astronomical body.
PNS	Petri Net Simulation	Implements a generic algorithm for Petri net simulation. Petri nets are commonly used to model distributed systems.
RPES	Rys Polynomial Equation Solver	Calculates 2-electron repulsion integrals which represent the Coulomb interaction between electrons in molecules.

Parboil benchmark is used in suitability study for CUDA by Hwu *et al.* [10]. Additionally, the suitability study includes Matrix Multiplication, Lattice Boltzmann method and Saxpy apart from Parboil which mostly derived from high-performance computing benchmark. The study done by [10] is accompanied by detail analysis of the architectural bottleneck from the implementation. Instruction issue caused most of the implementation bottleneck while a few others are caused by memory-related bottleneck such as capacity, bandwidth and latency.

2.3 Serpent Encryption

Serpent encryption operates based on 32-round SP-network with four 32-bit words as an input and up to 256-bit key as shown in Figure 4. The design of Serpent algorithm is presented with parallelism by bit-slicing [11]. The 4x4 S-boxes introduced within the SP-network has become the focus of previous works by [12] and [13] to improve Serpent's number of clock cycles. Different approaches were taken with Gladman's Serpent S-boxes [13] optimized for Intel Pentium 4 MMX while Osvik's Serpent [12] reduced the registers used by eliminating temporary variables thus fit in the number of registers inside x86 architecture processors. Both of the previous works provide good example of reducing the number of operations and managing memory for CUDA implementation.

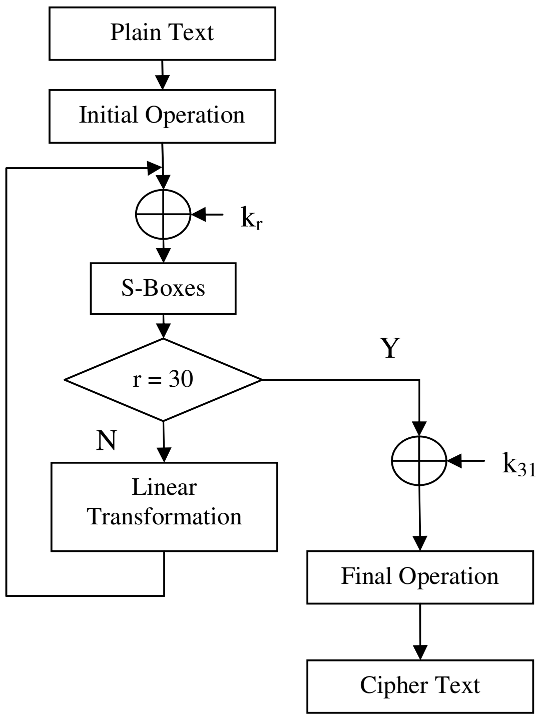


Figure 4: Serpent encryption flow chart

The Serpent encryption requires 132 32-bit words of key materials. The key length provided by the user is normally ranging from the minimum 128-bit to the maximum 256-bit key. Before any encryption could be done, the key provided by the user is expanded, mixed and went through S-boxes before becoming the key materials. The process is call “key scheduling” as summarized in Figure 5. The symbol π is a constant with the value 0x9e3779b9.

$$w_i := (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \pi \oplus i) <<< 11$$

$$\begin{aligned}
&\{k_0, k_1, k_2, k_3\} := S_3(w_0, w_1, w_2, w_3)\\
&\{k_4, k_5, k_6, k_7\} := S_2(w_4, w_5, w_6, w_7)\\
&\{k_8, k_9, k_{10}, k_{11}\} := S_1(w_8, w_9, w_{10}, w_{11})\\
&\{k_{12}, k_{13}, k_{14}, k_{15}\} := S_0(w_{12}, w_{13}, w_{14}, w_{15})\\
&\{k_{16}, k_{17}, k_{18}, k_{19}\} := S_7(w_{16}, w_{17}, w_{18}, w_{19})\\
&\ldots\\
&\{k_{124}, k_{125}, k_{126}, k_{127}\} := S_4(w_{124}, w_{125}, w_{126}, w_{127})\\
&\{k_{128}, k_{131}, k_{130}, k_{131}\} := S_3(w_{128}, w_{131}, w_{130}, w_{131})
\end{aligned}$$

Figure 5: Key Scheduling process

Graphic card as cryptography hardware is not entirely new, given the attempt is made around year 2005 using OpenGL for AES Cryptography [14]; however the performance suffered greatly from limited functionality. There were no successful attempts made after that until the arrival of CUDA. Manavski [14] managed to produce significant performance improvement using CUDA in AES Cryptography. The author managed to produce up to 20 times speedup over CPU using 8MB of data size and 128 bits of user key. Similar work is done for ARIA (cryptography originated from Korea), Yeom *et al.* [15] able to produce comparable result to AES by effectively using shared memory and registers inside the GPU.

2.4 Lattice Boltzmann for Fluid Flow

Ever since GPU is introduced to render computer images, the floating-point operations (FLOP) performance has been increasing significantly till today that has already reached hundreds of GFLOP/s compared to CPU which focused mainly on the integer operation performance for mass consumer. By utilizing the GPU computation capabilities along with low CUDA learning curve, we would produce methodology for a simple porting process for additional performance gain.

Lattice Boltzmann for Fluid Flow is known for the simple algorithm and its capabilities to be easily parallelized given the computation for each element inside the lattice corresponds only to the element function [16]. Consequently, the amount of resources needed for the computation is demandingly large and requires intensive memory access. Boghosian [16] reviewed on the Lattice Boltzmann Equation based on the book “The Lattice Boltzmann Equation: For Fluid Dynamics and Beyond” by Sauro Succi mentioned the algorithm originality and practical use. The algorithm is made through discreet understanding of Navier-Stokes equation. This has made parallel computation using multi-core platform possible and efficient.

Lattice Boltzmann has become increasingly popular in computational dynamics. Numerous attempts were made to increase the performance of algorithm that include study on the algorithm [17], optimization on multi-core platform [18] and multi-GPU implementation for Lattice Boltzmann [19]. All of the studies suggested significant performance gain by implementing the computation in parallel, however specific methodology for the implementation process on CUDA is not shown.

A phenomenon known as Karman Vortex Street is used for the computation fluid flow for this project. The unsteady separation of the flow of the fluid over a body causes a repeating pattern of swirling vortices that is illustrated in Figure 6. Detail methodology for CUDA implementation is described in Section 3.5.2.

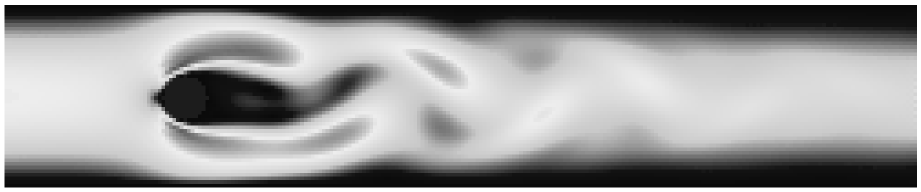


Figure 6: Karman Vortex Street illustration

CHAPTER 3
METHODOLOGY

3.1 Procedure Identification

Figure 7 illustrates the method and the work flow of the project.

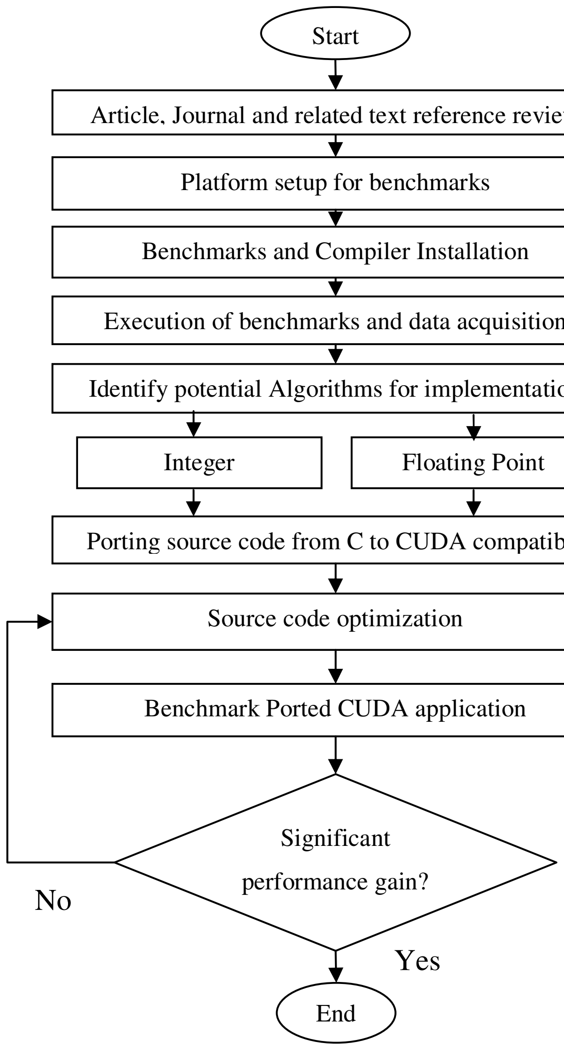

Start
Article, Journal and related text reference review
Platform setup for benchmarks
Benchmarks and Compiler Installation
Execution of benchmarks and data acquisition
Identify potential Algorithms for implementation
Integer
Floating Point
Porting source code from C to CUDA compatible
Source code optimization
Benchmark Ported CUDA application
Significant performance gain?
No
Yes
End

Figure 7: Flow Chart of Methodology

The first approach in understanding CUDA concept and features is by acquiring reference materials covering from the basic to implementation. The understanding ensures the methods and approaches used are proper and correct to avoid fallacies. The references are summarized and included in literature review. Theory and practical analysis is essential to provide explanation in performance measurement and programming algorithm.

After some of the theoretical review is done, the test platform was setup for initial benchmarking purpose. The benchmark suite was later identified for the project. This part is crucial in expecting the performance of the test setup and also understanding how CUDA will perform for floating point and parallelization computing.

CUDA performance measurement is done based on successful execution of benchmarks that depends on the ability to produce results and not only to acquire the output of the performance measurement. The measurement is to be done in a controlled environment with both essential and non-essential factors to be as constant as possible. The controlled environment includes but not limited to the Operating System, the number of background programs and most important the computer system specification. All of these are to make sure the highest possible results which are re-produce-able.

Based on the previous approaches, the computation on CUDA is done by analyzing results taken from the performance measurement. The analysis is done using profiling tools and time consuming process or instruction is identified. Numerical method is chosen for the implementation of CUDA. Graphical interface on the other hand, would require more time to be spent on the programming and the usage of more complicated libraries. This approach increases the chances of successful compilation of the program and proper execution.

3.2 Experimental Setup

The chosen hardware for the project is based on the availability and also budget available at the time. Table 2 provides detailed information of the hardware for the test system.

Table 2: Hardware Information for the Test System

CPU	Model Name	Intel Core 2 Duo Processor E4500
	Frequency	2200 MHz
	Front Side Bus	800 MHz
	L1 Cache	32 KB
	L2 Cache	2048 KB
Motherboard	Chipset	Intel P35 Express
	PCI-E	1.1
Memory	Type	Corsair CM2X1024-6400 DDR2 800MHz
	Size	2048 MB
GPU	Model Name	Gigabyte Nvidia GeForce GTX 260
	Memory	896MB GDDR3
	Core Clock	1242 MHz
	Memory Clock	2000 MHz

Some parts for the test system may already be at the low-end category at this current time but both of the processors and the motherboard still provide good performance-to-cost ratio. The PCI-E 1.1 the motherboard bottlenecks the graphic card which uses PCI-E 2.0 that has a faster transfer rate. The graphic card was chosen from the high-end category because of the compute capability in double precision [2] and because it has the same Stock-Keeping-Unit (SKU) with the top-of-the-line product. It is also decided based on the performance-to-cost ratio by the given specification. Detail information for CPU and GPU can be found in Appendix B.

Some more details that need to be covered are the software part for the system. The software is used throughout the project from benchmarks to implementation of Super Calculator. Table 3 shows the details of the software information in the test system.

Table 3: Software Information of the Test System

Operating System	Distributor Kernel	Red Hat Enterprise Linux 11 32-bit 2.6.18-92.el5
Compiler	Intel GNU Nvidia	Intel C/C++ Compiler 11.0 GNU Compiler Collection 4.1.2 Nvidia CUDA Compiler 2.0
Driver	Nvidia	177.11

The software chosen are distributed under open source license with some limitation. Software development under open source can be done within minimal budget and also a lot of dependencies can be solved by searching in the repository website in the Internet. As of the time benchmarking is done, the configuration used such as the compilers and the driver is of the latest possible. The number of processes running in the background is limited and controlled. The kernel performance is also relatively better and does not hog the memory resources.

3.3 Benchmark

Benchmark is done at the early stages to provide preliminary data that is needed for the progress of the project. The benchmark provides information of the test system performance according to the programs in the benchmark suite. Later on, based on the benchmark score, the performance-to-cost calculation can be done for relative comparison. The objective of the project also depends on the representation of the benchmark performance score.

The benchmark suite was executed under level five of Linux which in X11 mode in as no performance benefit is gained from level three command-line interface mode. The benchmark test run was executed at least three times to ensure consistency in the results. A range of five-percent deviation was set to define the consistency. The final results were taken based on the best score achieved and within the consistency range. The execution time is recorded into table and represented as graph for easier visual comparison.

Four different benchmarks were executed to test the system that is using two different compilers for the CPU benchmark and two different CPU frequencies for the GPU benchmark. The compilers used for CPU benchmark are GNU Compiler Collection (GCC) as reference and Intel C/C++ Compiler 11.0 (ICC) for Intel CPU optimization. Different CPU frequencies were achieved through manipulation of the CPU clock ratio to study the performance of CUDA with varying CPU frequency.

In the benchmark process, the source codes for CPU are compiled using GCC with full optimization level (-O3) and SSE3 instruction flag (-msse3), providing added performance to the CPU benchmark result. Additional flag that is also used for the ICC is inter-procedural optimization (-ipo) that enables in-lining of the code while processor specific optimization flag for Intel Core 2 Duo Processor (-xT) is used. These are the optimization flags that are enabled by default if the fast optimization flag (-fast) is used [22] since the flag is accepted as the base benchmark in Standard Performance Evaluation Corporation (SPEC) CPU benchmark [23]. Further optimization may affect the program size to become bigger and the accuracy of the benchmark timing. Bigger program size may not fit inside the L2 cache thus data transfer between cache and the main memory will be frequent resulting in longer execution time.

CUDA GPU benchmark source codes are compiled using the Nvidia CUDA Compiler (NVCC) and the CPU host codes are compiled using GCC as specified within [2]. As the optimizations provided by the compiler are more closely related to the hardware and source code, no other compiler related flag could be used for optimization. The compiler itself is relatively new compared to the compiler developed for the CPU, thus more features from NVCC is expected.

The score is reciprocal of the total time taken by the programs to run. The Input / Output (I/O) time is omitted as the data is not critical for the benchmark study. The overall score is calculated using geomean and normalized against processor with basic compiler score. Geomean is an average calculation using total of multiplication and square root. Thus the overall score will not be swayed by one or two large numbers. Normalization provides an easy view for relative comparison. The calculations involved in determining comparison data in Figure 24 and Figure 25 are shown in (1) and (2).

$$\text{Score} = \frac{1}{\text{GPU Execution} + \text{Copy time} + \text{CPU Execution}} \tag{1}$$

$$\text{Geomean} = \left(\prod_{k=i}^{n} \text{Score(k)} \right)^{\frac{1}{\text{n}}} \tag{2}$$

3.4 Potential Algorithms for Super Calculator using CUDA

Potential algorithms need to be identified and evaluated for feasibility before being adopted for implementation. As time is a major constraint, the selected application must be able to be ported within the time limitation of the project. Potential algorithm with source code written in C language could be considered a head start in porting the algorithm. CUDA implements the extension of C language which can provide an advantage in porting the source code that is written in C language for the application, thus shorten the time needed for programming and more time for program debug and optimization. Direct integration of the source code is possible either by linking or rewriting portion of the source code to be executed on the graphic card. All of these will require knowledge of the selected application's algorithm to benefit from CUDA parallelism.

3.4.1 Integer Operation

It is well known that graphic cards native operation is in floating-point for graphic processes. At a glance, integer operation might not receive any benefit from CUDA and the performance maybe less compared to CPU. This fact is supported by previous work by [21], as database operation mainly involves integer operation. Nvidia [2] also stated numerous clock cycles taken when performing integer operation. However, there are still integer operations that do not cost performance reduction that much, that is the bitwise operations. CUDA could sustain up to eight bitwise operations per clock cycle.

Cryptography is considered to have an intensive bitwise operations performed for converting plain text to cipher text. Recent success in implementing cryptography on CUDA for AES [14], DES [7] and ARIA [15] making it as motivation in using Serpent encryption as potential algorithm for the implementation. Additionally, Serpent encryption is distributed under GNU Public License (GPL), making it possible for modification and redistribution. A number of previous works in speeding up Serpent provided valuable data and information in providing basic understanding of Serpent performance so far.

3.4.2 Floating Point Operation

Floating Point operation consisted of single precision and double precision. Each of them must correspond to IEEE-754 floating point standard for computation. Single precision computation performs faster compared to double precision however it is less accurate. CUDA support for double precision floating point is also limited to graphics card with *Compute Capability 1.3* [2]. Lattice Boltzmann computation utilizes double precision floating point and since the graphics card used supports the double precision features, it has become the motivation for the implementation.

3.5 Porting to CUDA and Optimization

After the acquisition of the source code, the source code is divided into three sections that are data initialization, computation and result. Computation section is analyzed to identify the most compute intensive function. This can be done by using profiler or by manually adding in “Wall-clock” function to identify which computation took the most time. The compute intensive function is then converted as CUDA kernel function. This direct integration method is usually dubbed as naïve implementation method as some algorithms can already gain performance benefit using this method [7]. A function operating on an array containing a large number of elements can be used as an example.

A data with 1MB of size is fitted into an integer array with 256K of elements. Memory space is allocated using “`malloc`” since the array size is considered large to avoid segmentation fault error. After the array has been initialized, the function “`function_array`” is called to perform the operation to the array. The function has to loop through the array’s element to perform the operation. Figure 8 shows the source code fragment for the function.

```
#define array_size 256*1024

int *array_sample;
// memory allocation
array_sample = (int *) malloc(array_size*sizeof(int));

... // array initialization

function_array(array_sample); // function call

function_array(int *array_sample) {
      int i;
      for(i = 0; i < array_size; i++) {
            array_sample[i] = array_sample[i] + 5;
      }
}
```

Figure 8: C source code fragment for sample array function.

The function "`function_array`" operated on the array elements in independently with no relation between each element. Thus the function can be ported safely to CUDA without any complication. The porting process involves adding in CUDA header file and specifying identifier "`__global__`" for the CUDA function. CUDA's function call syntax is also different from a standard C language by using different identifier such as "`function_array<<<NumBlock, NumThread>>>(datatype variable)`". Apart from the parameters for the function argument, the syntax includes the number of blocks and the number of threads per block respectively. The ported code is shown in Figure 9.

```
#define array_size 256*1024
#define NumThread 256

__global__ void function_array(int *array_CUDA) {
 int idx = blockIdx.x * blockDim.x + threadIdx.x;

 array_CUDA[idx] = array_CUDA[idx] + 5;

}


void main() {
 int *array_sample;
 int *array_CUDA;
 int NumBlock;
 size_t size;

 size = array_size*sizeof(int);

 // request page-locked memory buffer
 cudaMallocHost((void**)&array_sample, size);

 ... // initialization for array_sample

 cudaMalloc((void**)&array_CUDA, size);
 // copy data to device
 cudaMemcyp(array_CUDA, array_sample, size,
            cudaMemcpyHostToDevice);

 NumBlock = array_size/NumThread; // determine number of blocks

 //function call
 function_array<<<NumBlock, NumThread>>>(array_CUDA);
 cudaThreadSynchronize();
 // copy data to host
 cudaMemcpy(array_sample, array_CUDA, size,
            cudaMemcpyDeviceToHost);

}
```

Figure 9: Source code fragment for ported CUDA-compatible code

Although source code from Figure 9 is longer than Figure 8, all of the added functions are the basic requirement for the CUDA application to work with additional performance gain. Page-locked memory buffer is requested by the CUDA host memory allocation (“`cudaMallocHost`”) for faster data transfer between the host and the device. The data is moved to and from the device by using memory copy function. From here onward, the porting of the source code will become more specific to the algorithm implemented. The common processes between Serpent Encryption implementation and Lattice Boltzmann implementation on CUDA will be used to produce the guideline.

3.5.1 Serpent Encryption Algorithm Implementation

From the algorithm analysis, it is known that Serpent encryption involves data dependencies from previous iteration. The algorithm also occupies resources in the execution of sequential functions as well as load and store data. Since the algorithm itself is in serial sequence, this makes it almost impossible to parallelize the processes (unless using bit-slice method similar to the hardware implementation). Nevertheless, the encryption process could be executed in single thread without consuming much resource, therefore opening another possibility in parallelization.

The data initialization section of the source code is modified to handle multiple data stored in an array. Since the original input data is already in array form (i.e. x[4], with ‘x’ as the variable for input), another dimension has to be added to the array. Although it is possible, adding in another dimension caused some confusion as the variable is in pointer-array form as it involves dynamic memory allocation and fixed array all at the same time. Instead of adding in another dimension, we used structured variables to make the input block as a data type. The structure is aligned to 16-byte boundary to enable single read/write memory instruction for 128-bit. The structured variables and data types are shown in Figure 10.

```
typedef struct __align__(16) {
unsigned long x0, x1, x2, x3;
} SER_BLOCK;

typedef struct __align__(16) {
unsigned long k0, k1, k2, k3;
} SUBKEY;

typedef struct {
subkey k[33];
} SER_KEY;
```

Figure 10: Source code fragment for structured variables and data types

The initial attempt made concentrate solely on the encryption function of the application with each thread performing encryption for 16-byte of plain text. The structured variables and data types are applied to the encryption process as shown. The structured data types are now able to be declared as pointers for dynamic memory allocation and page-locked memory buffer for faster memory access.

```
Line  Command
   __global__ void cuda_encrypt(SER_BLOCK *enc_block,  SER_KEY
   *keys) {

   int idx = (blockIdx.x * blockDim.x + threadIdx.x);

   enc_block[idx] = keying(enc_block[idx], keys[idx].k[ 0]);
   enc_block[idx] = SBOX00 (enc_block[idx]);
   enc_block[idx] = transform(enc_block[idx]);
   …
   enc_block[idx] = keying(enc_block[idx], keys[idx].k[31]);
   enc_block[idx] = SBOX31(enc_block[idx]);
   enc_block[idx] = keying(enc_block[idx], keys[idx].k[32]);
         }
```

Figure 11: Encryption process on graphic card using CUDA

The parallel encryption is described in source code in Figure 11. The variable "`idx`" the thread number identifier. Each of the plain text is encrypted within the thread specified by the "`idx`". The structured data types avoid confusion from having to build two-dimensional array.

Figure 12 illustrates an example of 16KB of plain text encrypted using parallel thread. The time taken for key scheduling is excluded initially as it was done on the CPU and transferred to the GPU for the encryption. The performance gain from this method only showed around 1.67 times faster than the application executed on the CPU as shown in Figure 12. This is caused by the huge memory transfer of the key materials for the encryption. For example, 16MB of plain text would require around 528MB of key materials as 528-byte key materials are for every 16 bytes of data.

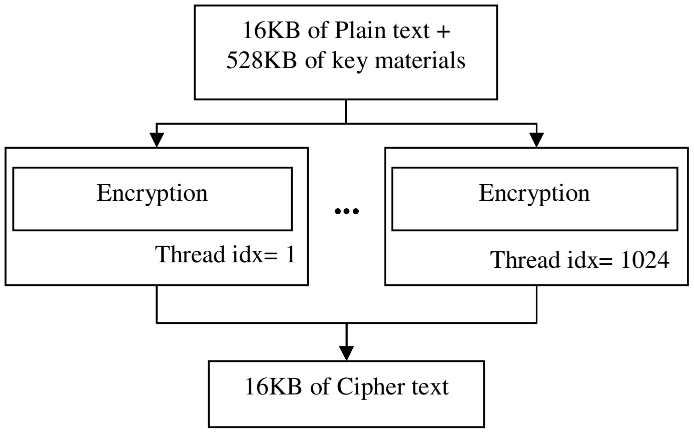


Figure 12: Block diagram for 16KB of data size for the initial attempt

Profiler result of the initial attempt indicates the performance suffered greatly from huge memory transfer of the key materials. This has been previously observed based on the Parboil benchmark result which resulted in poor performance for Sum of Absolute Difference (SAD) algorithm. This is known through benchmark profile in Section 4.2 and discussed in Section 4.4.3. The only way to overcome the performance degradation is to include key scheduling process inside each thread before encryption as shown in Figure 13. Consequently, the memory transfer size for the encryption key materials are reduced to 32-byte for every 16-byte of plain text. This implementation managed up to 7.5 times performance gain in throughput. Memory transfer is identified as one of inefficiencies in CUDA if it was not managed correctly. Detailed benchmark result is discussed further in Section 4.3 of this report.

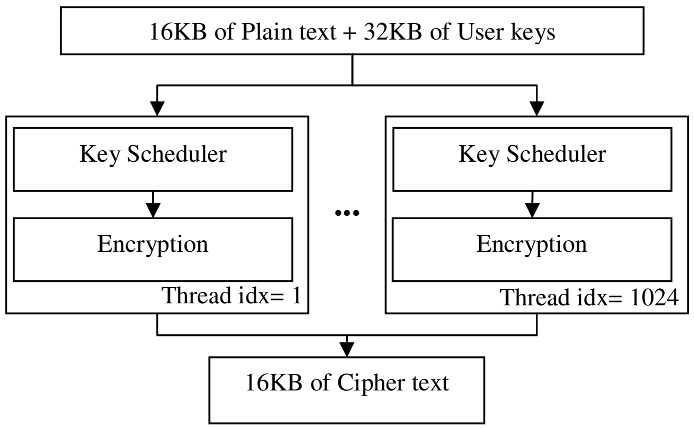


Figure 13: Block diagram for 16KB of data size in Complete Application

Figure 14 shows the fraction time spent in the GPU for both of the attempts by using CUDA Visual Profiler. The fraction of time taken for the initial attempt's memory transfer occupied a lot more compared to the complete application's memory transfer. Thus, in this case, optimization is done by limiting memory transfer.

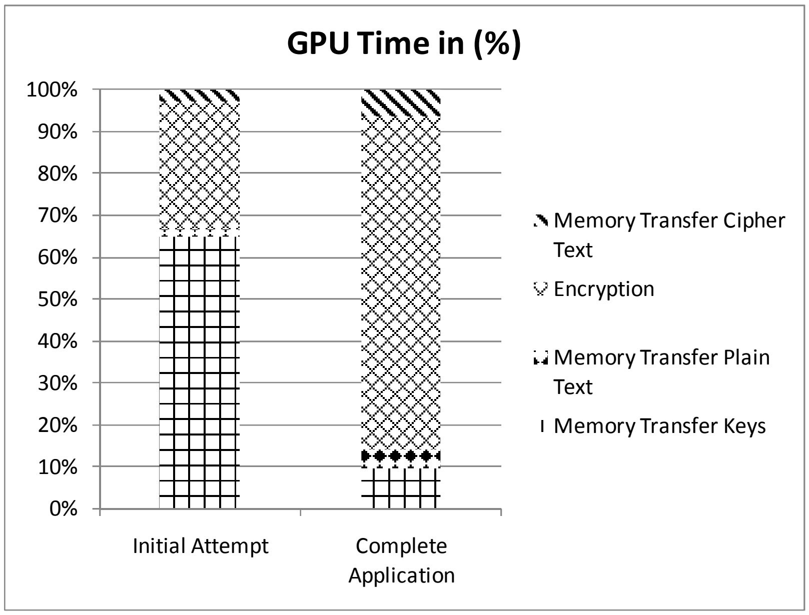


Figure 14: GPU time spent in percentage

A few more attempts were made to push the performance gain further that includes changing the number of threads per block, varying the input block size and specifying the number of registers to be used. The results are discussed in Section 4.3.

3.5.2 Lattice Boltzmann for Fluid Flow

We used a simple simulation in utilizing the Lattice Boltzmann Method (LBM) to simplify the porting process. The program simulates Karman Vortex Street phenomenon that has been discussed in Section 2.4 of the report. The source code is acquired from an open source solution, that is the OpenLB [20] and it is still maintained by the users although depreciated. It is written in modular form for readability and flexibility for future extension. Unlike Serpent encryption, OpenLB consisted of many processes for a complete simulation. Thus porting the source code from C to CUDA-compatible would not be just one or two functions, but instead it needs several functions to perform correctly.

The OpenLB's "Unsteady" program source code utilizes a lot of pointers for the data and functions. Additionally, OpenLB is managed with external function link between the main file with the files containing the computation function. All of the computations for the Lattice Boltzmann are done with double-precision floating-point. The complete simulation flow chart is shown in Figure 15.

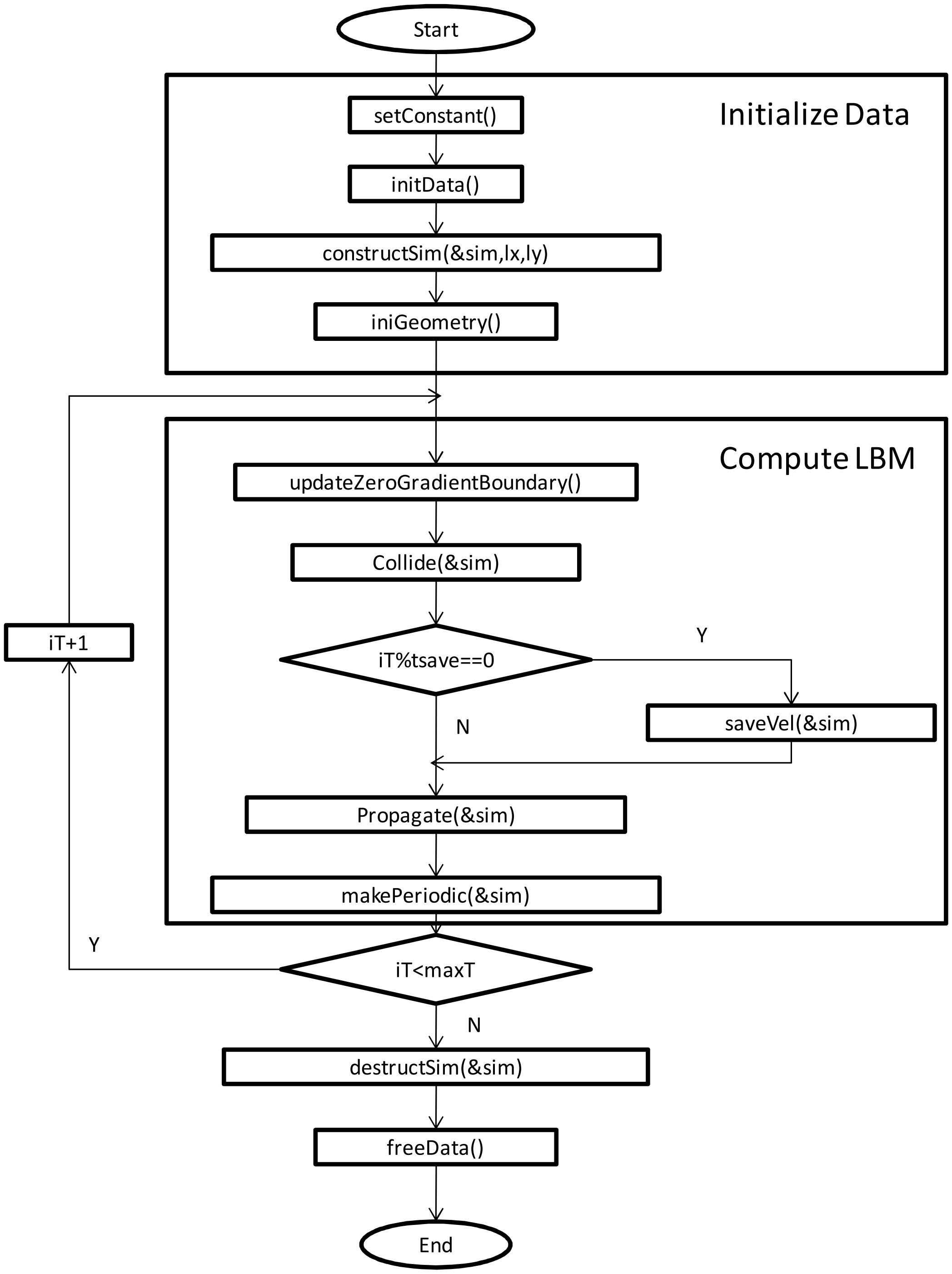


Figure 15: Flow chart for “Unsteady” simulation

We profiled the program by adding a "Wall-clock" function to the program for time measurement. The "Wall-clock" function measures the time taken from one point to another point in the program. Although, the timing might not be accurate compared to a real profiler application, it served the purpose in determining the ratio of a function runtime to the whole program execution time. The iteration of the program, which is controlled by the variable "maxT" is set at 100000. Based on the method used, the execution time for the function "Collide(&sim)" is measured for comparison.

The "Collide" collision simulation function consisted of several modular functions that are differentiated according to the element position inside the lattice. The functions represent the computational dynamics for Bounce Back, Bhatnagar–Gross–Krook (BGK) for the bulk dynamics and all of the side boundaries include upper, lower, left and right. The elements function are determined in geometry initialization function (i.e. "iniGeometry()"). The initialization is summarized and shown in Figure 16 and Figure 17.

The original source code utilized function pointers and function template parameters that is not supported by the Nvidia CUDA Compiler 2.3. Additionally, data declared as pointer will have to be properly dereferenced for the GPU execution otherwise the output would not be correct. The adjustment in the data structure is necessary for the source code to be compatible with CUDA and executes correctly on the GPU. The original source code uses 2D array for the lattice, although CUDA compiler support 2D array, proper indexing of the 2D array proved to be difficult.

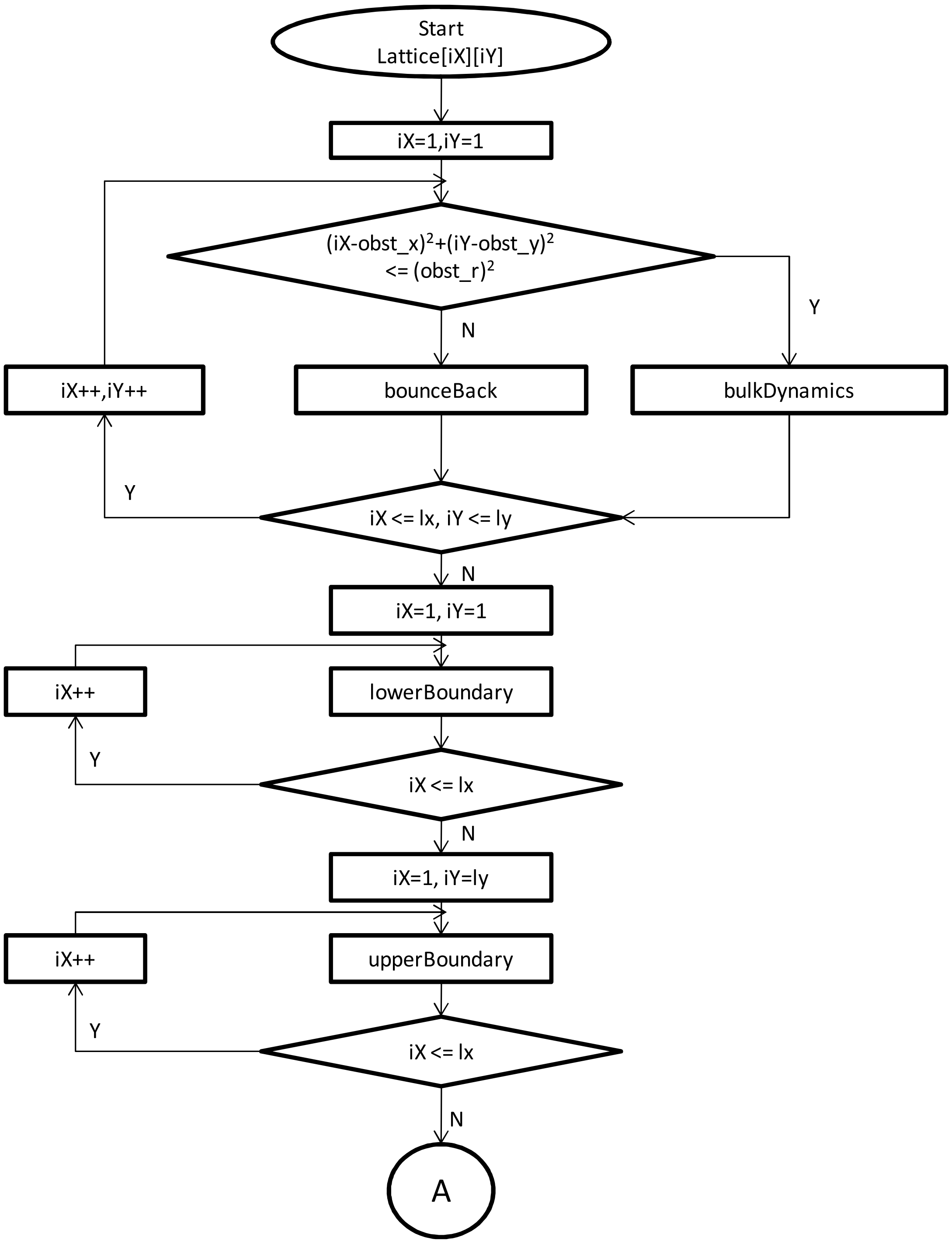


Figure 16: Flow chart for the first part of geometry initialization function

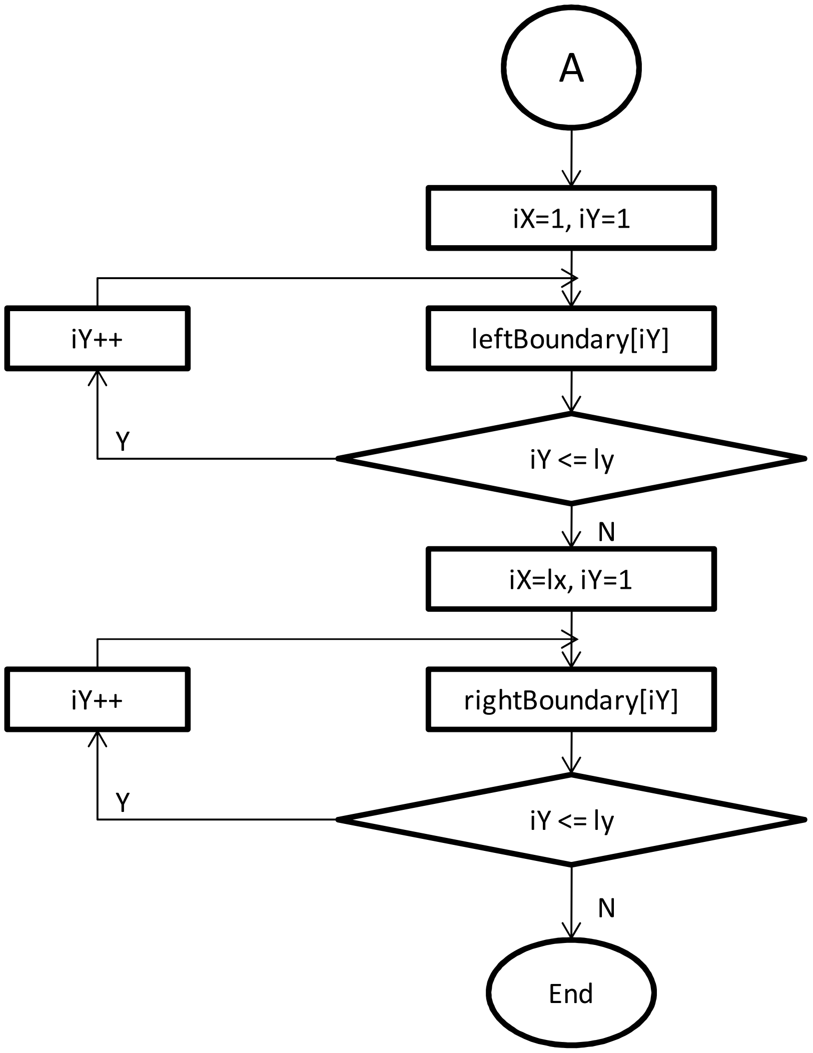


Figure 17: Flow chart for the second part of geometry initialization function

The elements of the lattice are sorted out according to its function. Since the compiler does not support pointer function, each of the function is assigned a numerical value to be used in the conditionals elements sorting for faster comparison. The functions are ported to CUDA-compatible source code by adding parallel access using arrays. The amount of data handled for the computation is smaller compared to Serpent Encryption. The amount of computations that is done on the GPU is also less intensive compared to Serpent Encryption. Consequently, this would make it hard to hide the global memory latency as suggested in the guideline for Serpent Encryption. However, the small amount of data allows the usage of shared memory and constant memory cache to be used.

Based on the profile of the program, the function "Collide" occupies most of the program execution time. The goal of the initial attempt is to minimize the time taken for the function to execute and shorten the execution time altogether. Initial attempt was made by porting each of the function by using naïve implementation as shown in the flow chart in Figure 18.

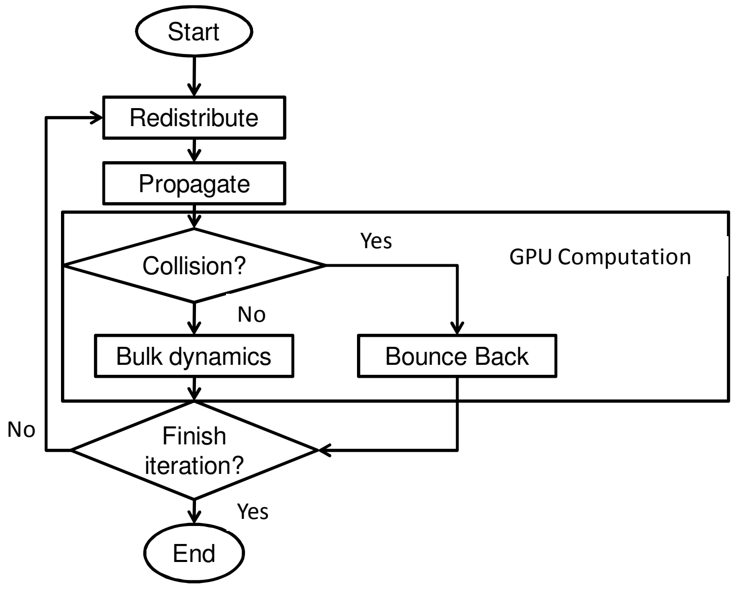


Figure 18: Flow chart for the initial attempt on LBM CUDA implementation

Although this is the easiest method in getting the functions to be executed on the GPU, the code has become very inefficient. As a result, the performance for the implementation degraded significantly. Figure 19 shows the comparison between CUDA implementation with CPU. The CPU compilers used as comparison are Intel C/C++ Compiler 11 (ICC) and GNU C Compiler 4.4 (GCC).

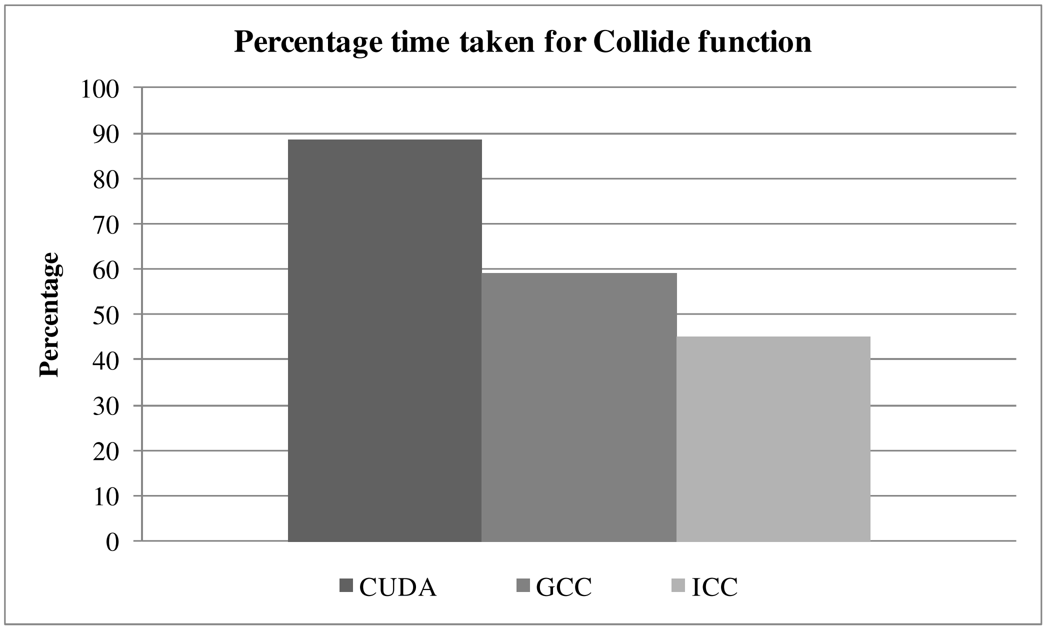


Figure 19: Bar chart for percentage time taken for function "Collide"

The performance degradation is caused by multiple requests for page-locked memory buffers (i.e. "`cudaMallocHost`"). The time taken for the memory allocation occupies most of the function's execution time. Although page-locked memory buffer offers high data transfer rate compared to usual memory allocation using standard C function, the amount of time taken for the transfer does not overcome the performance degradation caused by the page-locked request.

The last attempt made resulted in significant degradation of performance. Another attempt is made by including the data initialization and overall computation into the GPU. Since the initial value for the entire lattice elements are the same, the data initialization can be done in parallel using the GPU. Although the computation involved a few conditional branching, it is assumed the number of conditional branching is within the GPU limited capabilities. The general flow of the program is illustrated in Figure 20.

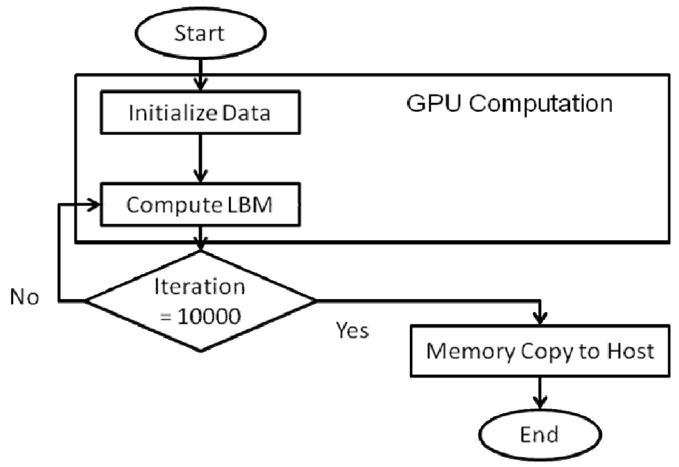


Figure 20: Flow chart for Lattice Boltzmann Method complete application implementation on CUDA

This is done not only to minimize the amount of size for memory transfer but also to reduce the complexity of the memory copy. Although CUDA supports multi-dimensional array operation, the array stored in the device memory is “flattened” (i.e. from to two-dimensional array to one-dimensional array) to reduce the complexity of the memory allocation. However, even though the array inside the GPU is flattened, it can still be addressed and computed in two-dimensional fashion. Figure 21 illustrates the two-dimensional computing model supported by CUDA and Figure 22 shows the source code fragment for indexing method used to address one-dimensional array for two-dimensional computation.

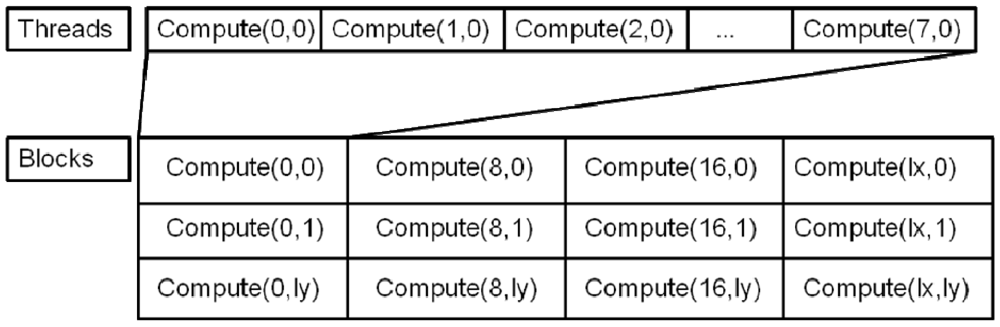


Figure 21: Block diagram showing 2D computational model on CUDA

```
int idx = blockDim.x*blockIdx.x+threadIdx.x;
int by = blockIdx.y;
// node[x][y] = node_gpu[idx+lx*by];

node_gpu[idx+lx*by] = compute(node_gpu[idx+lx*by];
```

Figure 22: Source code fragment for addressing 1D array for 2D computation model

The direct integration for the complete application has shown significant performance gain in the execution time. The complete application was benchmarked with varying lattice size to study the performance effect. The result is further discussed in Section 4.4. After the successful source code porting process, an optimization was done to further improve the performance gain of the application.

The optimization done has affected only small part of the source code. Blocks and threads that are managed well will have equal distribution of resources and tasks, thus the device can perform efficiently. However, the higher the number of blocks or the number of threads per block does not mean it will give the better performance. In this case, we measure the performance for each number of threads per block to determine the best number of threads per block. Figure 23 shows the result for the number of threads optimization with the result normalized over 8-thread per block.

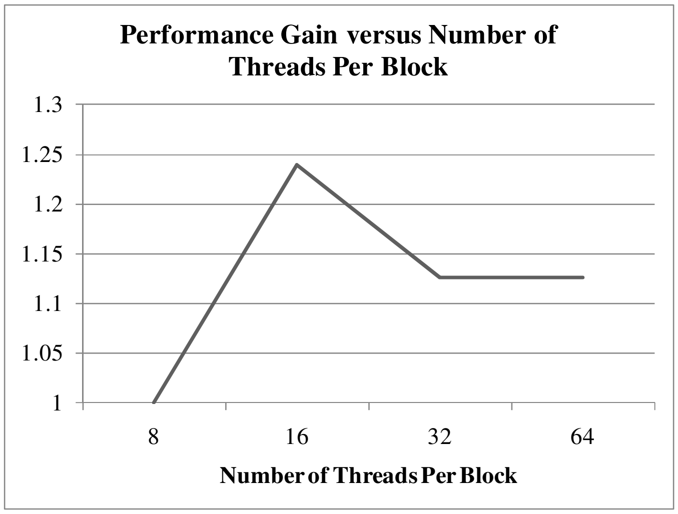


Figure 23: Graph for performance gain in execution time versus number of threads per block with normalized result

3.6 General Guideline for CUDA porting process

Based on the porting process of Serpent encryption algorithm and Lattice Boltzmann a number of key points is summarized for the general guideline:

- Identify parts in the source code for data initialization, compute intensive task and data retrieval.
- Aligning a structure type containing 16 bytes of data to 16-byte boundary provide coalesced memory transaction for the global memory thus increasing global memory bandwidth efficiency.
- Page-locked memory buffer can be requested for higher data transfer rate however multiple requests over small amount of data degrade the overall performance significantly.
- Examine computation demanding task for data dependencies as well as sequence. This is important to recognize whether the computation can be done in parallel. Usually an arrays or matrices operation that is done in a loop without any dependencies can be directly ported.
- Although the global memory has the highest latency among the other types of memory in the device, the latency can be hidden by compute intensive instructions.
- Memory management is crucial in getting most of the performance from CUDA. By using shared memory as a temporary memory for operations could increase the performance.
- The amount of size for memory transfer must be kept at minimum as possible for efficiency. Generally, a big ratio between computations to memory transfer must be maintained.
- For multi-dimensional array computation, it is more convenient to flatten the array for memory allocation and memory transfer but the data will still be able to be computed in multi-dimensional computational model by managing the kernel's blocks and threads.

CHAPTER 4
RESULTS AND DISCUSSION

4.1 Parboil Benchmark

This part of report displays the result from the benchmark and some findings that are worth mentioning. The benchmark result is obtained as at the time using the hardware and software details mentioned in the previous section. Figure 24 shows the summary of the benchmark result. The scoring system used was explained earlier in previous section. It is meant to provide relative comparison as well clear indication of the performance improvement.

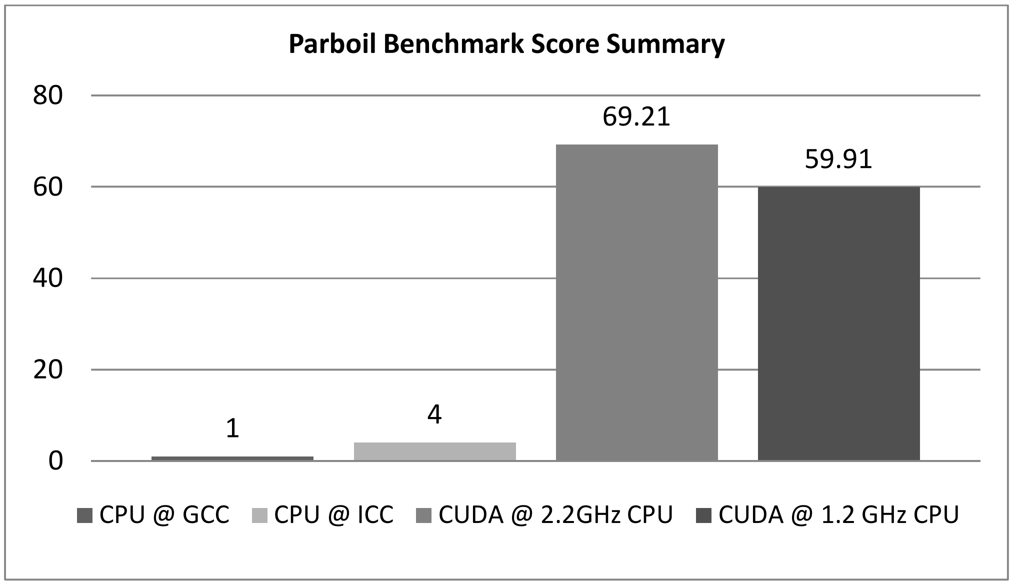
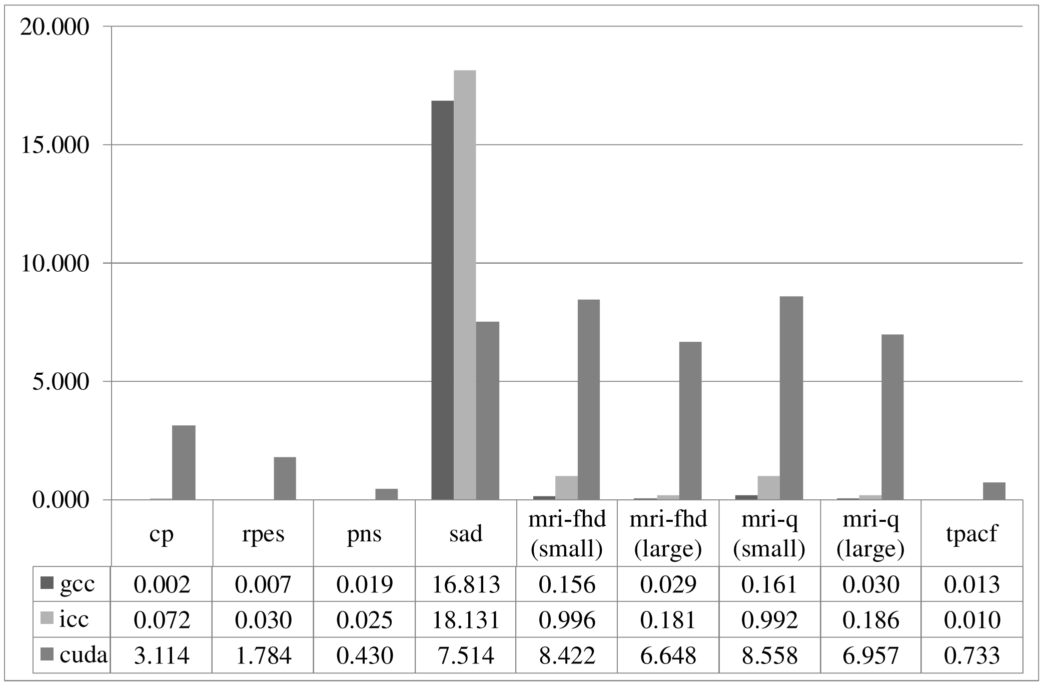


Figure 24: Bar chart for normalized Parboil benchmark score

The benchmark was executed a number of times for precision timing as to make sure that each execution time does not vary too much from each other. The data was then recorded based on the last execution and calculations were done to come out with the figure. The overall score was calculated as explained in the Methodology section of the report. The benchmark explains the bigger picture of the project as it is one of the objectives of the project. However, the benchmark details must be examined for the result to be more meaningful. Detail elaborations of the benchmark result are described within this section.

As the score summary depict, the benchmark performs 4 times better using ICC even when using GCC with generated processor instruction set. Unfortunately, the source codes for the benchmark programs do not support processor parallelization for it to execute on multiple thread. Thus the comparison can only be made with single thread processor. Aggressive compiler optimization flag were not tested as it may affect the accuracy of the benchmark such as when specifying the loop unroll, thus the compiler optimization flag were kept standard and more focused towards the effect on the hardware itself. The benchmark comparison between GCC and ICC translate the compiler performance in enhancing program execution time.

The benchmark shows that GPU performance has big advantage over the CPU with up to 69 times from GCC compiler and 17 times from ICC compiler. As for GPU benchmark, it is found that the GPU performance indeed scale with the CPU frequency of 1.15 times for 1 GHz. The CPU host code may occupy a small fraction of code, thus the scale between CPU performance and GPU performance is within the factor of the fraction. This comparison is only valid as clock-per-clock basis with similar CPU architecture since newer CPU architecture design computes at lesser clock cycle. Further elaboration in Discussion section explains more on CPU affect on GPU performance. Based on the result, it can be conclude that CUDA increase performance on a system even on low-end system with low CPU frequency.



	cp	rpes	pns	sad	mri-fhd (small)	mri-fhd (large)	mri-q (small)	mri-q (large)	tpacf
■ gcc	0.002	0.007	0.019	16.813	0.156	0.029	0.161	0.030	0.013
■ icc	0.072	0.030	0.025	18.131	0.996	0.181	0.992	0.186	0.010
■ cuda	3.114	1.784	0.430	7.514	8.422	6.648	8.558	6.957	0.733

Figure 25: Bar graph for detail benchmark score

Figure 25 shows the detail benchmark scores for all of the benchmark programs. Two of the benchmark programs (MRI-FHD and MRI-Q) have two inputs in varying size that is noted by small and large that represents the size of the image used for the MRI application. Each of the programs implements different data structure in calculation. The detail benchmark scores provide further breakdown of the benchmark result. It can be seen that most of the programs have large improvement over the CPU counterpart except for Sum of Absolute Difference (SAD) benchmark and Two Point Angular Correlation Function. In order to avoid having the same performance decline in the implementation part, the source codes are profiled using Nvidia CUDA Visual Profiler to provide more information for the benchmarks.

4.2 Benchmark Profile

Source code profile provides detail information of the program flow indicated by the timestamp, routine executed during the time, length of time and elapsed time from the beginning of the program. The interest in profile is scale down to GPU as the implementation of the application requires the information from the GPU side. Figure 26 represents the data of the benchmark profile based on time percentage for the GPU.

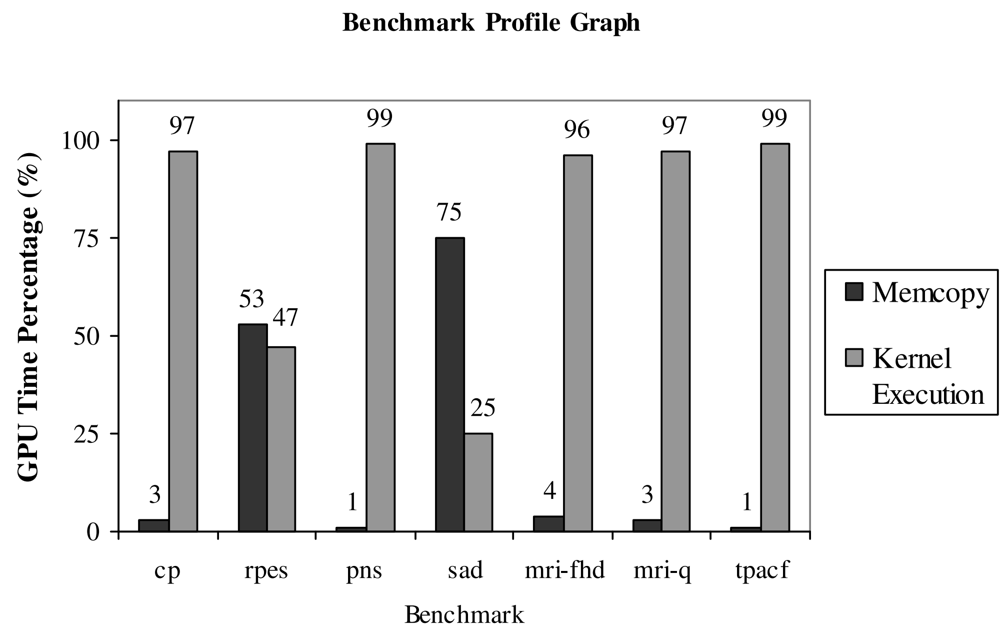


Figure 26: Graph for Benchmark Profile according to GPU Time Percentage

The graph shows two of the general routine executed by the benchmark. The "Memcopy" represents the routine of memory copy involving device and host. The "Routine" represents the calculations involved in the GPU and the percentage time taken.

4.3. Serpent Encryption Benchmark

Based on the successful attempt in optimizing serpent encryption using CUDA, benchmark is then done to study the effect of the number of block size with the throughput performance. This data is valuable as the effect studied is the scalability of CUDA and the maximum number of block size capable before all of the resources on the graphic card are used. Figure 27 shows the throughput performance of the Serpent encryption for CUDA and CPU. The maximum data size achieved is 16MB, while maintaining the throughput performance. The performance gain maintained for all of the block sizes that is 7 times more throughput. The benchmark was done using Nvidia CUDA Compiler 2.3 and GNU C Compiler 4.4 for CPU.

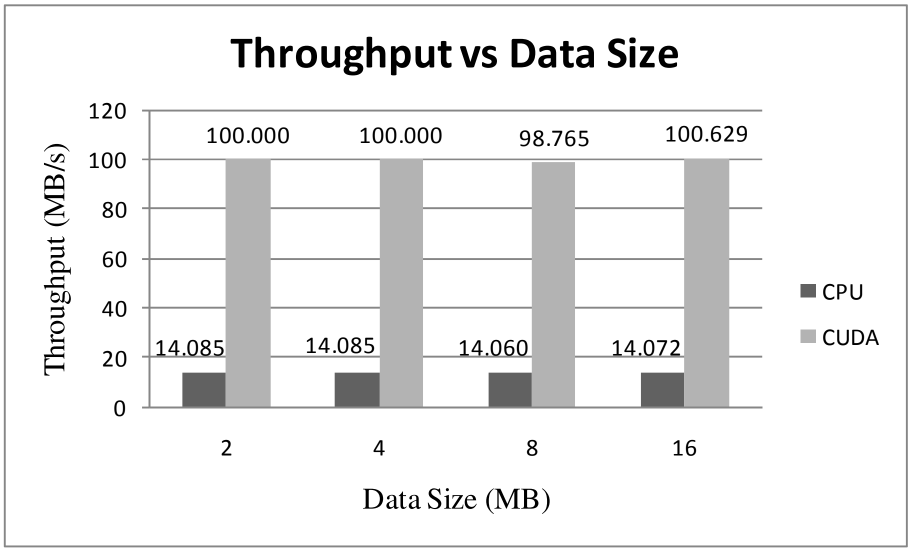


Figure 27: Graph for Serpent encryption benchmark result

The result is compared side-by-side for initial attempt and complete application. The improvement achieved by limiting the memory transfer was significant. The throughput for CPU degraded in second attempt comparison as the benchmark also includes key scheduling process for CPU. The graph is shown in Figure 28.

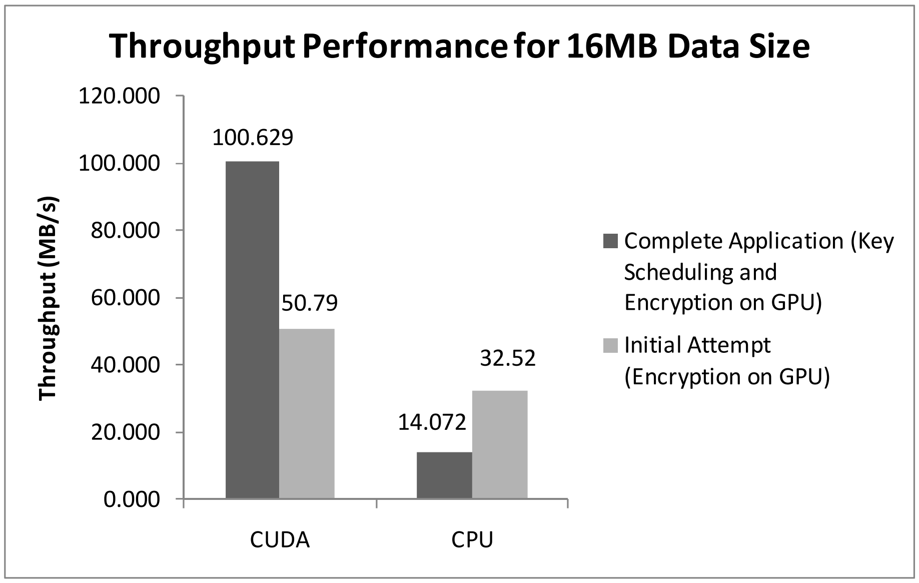


Figure 28: Throughput comparison for complete application with initial attempt

Two more benchmarks were done to study the effect of number of threads per block and also the number of registers per thread. These benchmarks were using 16MB data size as the plain text input. The number of threads per block was varied by specified manually inside of the source code while the number of registers is passed through to the compiler as the maximum registers count. The results are shown in the graphs in Figure 29 and Figure 30.

Although no significant performance gain can be achieved through the attempts, it is safe to assume the current algorithm has already reached the maximum possible throughput. It is interesting to mention the throughput result for varied number of threads per block for 64 threads is slightly better than 256 threads. Further detail is discussed in Section 4.5.4.

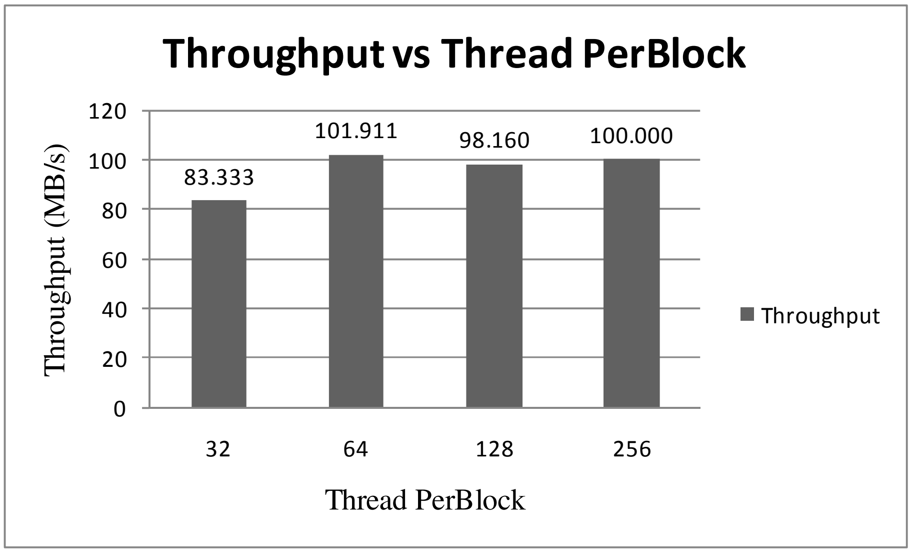


Figure 29: Throughput performance for varied number of threads per block

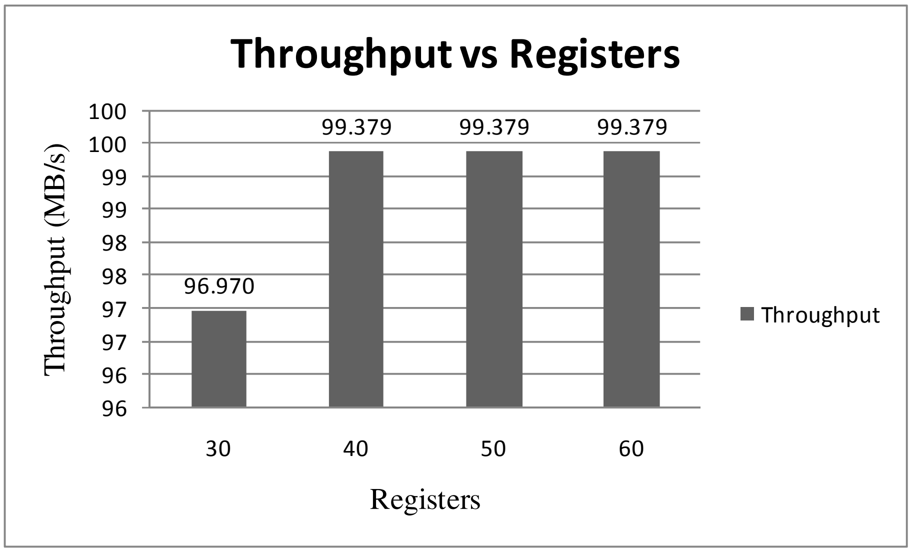


Figure 30: Throughput performance for varied number of registers per thread

4.4 Lattice Boltzmann Method benchmark result

Based on the direct integration implementation for Lattice Boltzmann method on CUDA, the performance of the implementation was measured against the CPU source code with GCC 4.4 compiler as reference and ICC 11.0 as high-performance compiler for CPU. The implementation managed a performance gain up to 10 times with the performance gain increase with the increment of the lattice size. The optimized thread number performance gain over the un-optimized thread number also increases as the lattice size increases. Figure 31 shows the graph of the performance gain in terms of execution time with the result normalized over CPU based code compiled using GCC 4.4.

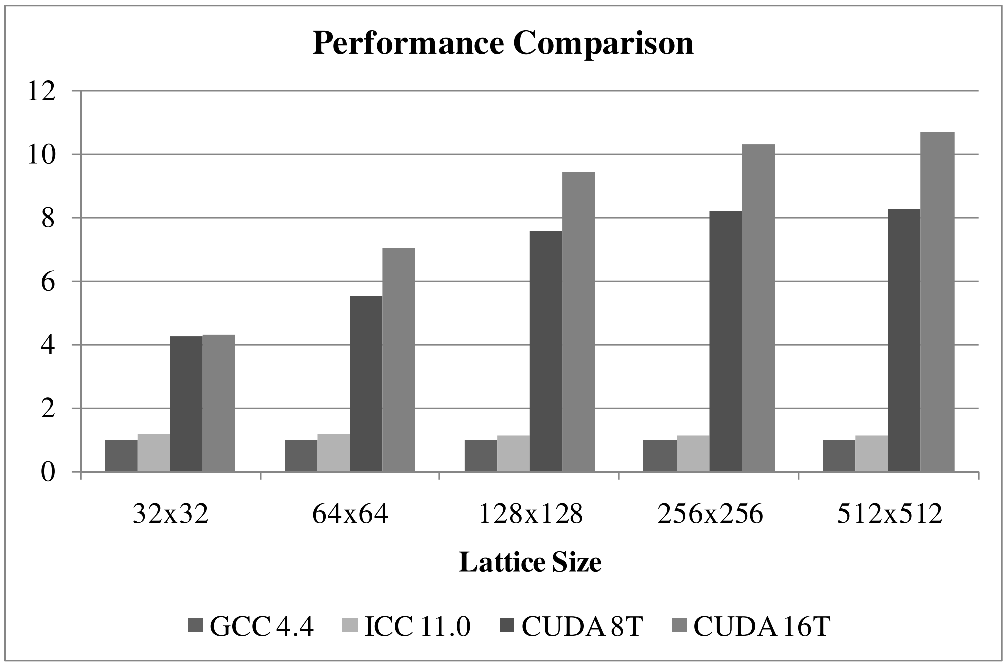


Figure 31: Bar chart for execution time performance comparison. The higher is the better performance

The data trend shows an increasing performance gain as the occupancy of the GPU increases and as the occupancy of the GPU nearing the maximum, the performance gain only increases slightly.

4.5 Discussions

4.5.1 Compiler Flag Optimization

The C/C++ Compiler for CPU has been extensively developed for quite some time. As the CPU technology advance, new processor flag instruction is introduced to execute instructions at lesser clock cycles. Thus, to make use of the processor flag instruction, the compiler has to be optimized for the specific CPU to be able to generate special instruction code. This is where programmer normally optimized the program to make it run faster by compiling the source code with optimization flag. In the benchmark process, the source codes for CPU are compiled with full optimization level (-O3) and SSE3 instruction flag (-msse3), providing added performance to the CPU benchmark result. Additional flag that is also used for the ICC is inter-procedural optimization (-ipo) that enables in-lining of the code while processor specific instruction.

Optimization can unleash the true potential of the hardware but it can also be misleading. For instance, generating profile is one of the compiler optimization features that can improve the execution time of the program however at the same time requires the program to be executed at least two times in order to generate the profile for optimization. The profile is compatible with system that has similar configuration to the test setup but may not be for others. This does not state the true fact about the performance of the program and also the system when execution time is concerned. However in some cases, profile does benefit greatly if the implementation involves similar algorithm.

4.5.2 GPU Benchmark Score

Execution time for the GPU to execute the benchmark programs include the time taken for host code to execute, data copied into the share memory and device code to execute. All of these are necessary in order for a program to execute on GPU, thus it cannot be neglected. Nevertheless, input/output time was omitted from the execution time calculation as mentioned in Methodology chapter. GPU Benchmark Profile provide more information as the GPU

The CPU frequency scaling and its effect on GPU performance are studied in the benchmark. As stated before, the CUDA source code contains lines of codes for the host to execute before being offloaded to the GPU. A good CUDA code will take minimal time in host execution and minimal amount of time for the data to be copied to the shared memory in the GPU. From the benchmark, MRI application which is applied in matrices benefits greatly from CUDA. Other application such as Cuolombic Potentials, make use of CUDA Three Dimensional (3D) vector capabilities to compute 150 times faster compared to GCC and 42 times faster compared to ICC.

4.5.3 Memory Copy

One of the concerns in developing application on interconnected device is transfer of data. In the case of Super Calculator test setup, GPU is connected through PCI-E v1.1 which has a data rate of 250 MB/s. The memory copy routine as shown by the benchmark, affects the performance of the application. Figure 26 show that SAD application took performance degradation from the high percentage time of memory copy. One of the factors to be considered however is the use of PCI-E v1.1 which only half the data rate of the latest PCI-E v2.0. The memory copy may have been too large, thus this factor should be considered in developing the application.

4.5.4 *Serpent Encryption benchmark data trend*

The benchmark data only shows a flat trend over increasing data size. However, this only shows the scalability up to the maximum data size the GPU could handle in single execution. The maximum data size is achieved at 16MB because of the memory limitation within the graphic card. Larger data size will need to be broken into chunks for parallel encryption. The optimum number of threads per block for Serpent encryption algorithm is 64 while maximum number of registers could be maintained at 40.

The number of blocks corresponded to the number of thread per block for a given data size. Larger number of thread per block would result in smaller number of blocks. While having small number of threads per block will result in poor performance, having large number of thread per block in this case does not give any performance gain either. Each of the blocks has shared memory that is shared between the threads in the same block. However the size of the shared memory is limited to 16KB [2]. Thus the right number of threads per block will give better memory access, resulting in better performance.

The maximum number of registers specified limit the number of registers used for storage purpose, thus making it available for computation. Both shared memory and registers per block are limited, thus device memory is used to store the large amount of data. Although the device memory storage is a lot bigger (896MB in this case), access latency is much higher. Compute intensive and memory intensive application need to consider these factors as compute intensive program would be able to hide the latency; however memory intensive program may suffer performance degradation cause by the latency.

CHAPTER 5
CONCLUSION

Based on the work done to both Serpent encryption and Lattice Boltzmann, a general guideline is produced for simple and direct integration method from CPU-based algorithm in C language to CUDA-compatible source code with minimal effort. Both of the programs are based on compute intensive application that usually stresses the CPU extensively, thus becoming perfect candidates as an example for the implementation.

The benchmark data and profile data provide information and expectation for the development of application for the Super Calculator. The analysis has provided knowledge and understanding on the performance study of CUDA. Serpent encryption algorithm which is based on integer and bitwise manipulation managed to achieve up to 7 times performance gain compared to the CPU by using multiple block encryption in parallel. The data for the multiple blocks is handled independently without any relation with the other blocks. The Lattice Boltzmann was chosen for the floating-point implementation because of its parallel algorithm for the lattice's elements computation. By using direct integration method, the performance gain in the execution time is up to 10 times over that of the CPU.

Although CUDA is able to provide massive parallelism, it is limited by only one kernel can be executed at the same instance. Nevertheless, it was able to give considerable performance gain and still holds much potential. Direct integration method also worked well, given the programmer could spend some time in profiling the ported source code. Optimizations such as limiting memory transfer, thread block management and memory management could already give considerable performance gain over system with just CPU as computation processor.

REFERENCES

[1] M. Harris, “General-Purpose computation on Graphic Processing Units”, [Online]. Available: http://www.gpgpu.org. Last Accessed: September 10, 2009.

[2] NVIDIA Corporation**,** “Programming Guide,” in *NVIDIA CUDA Compute Unified Device Architecture,* Ver. 2, June 2008, [Online]. Available: http://www.nvidia.com/object/cuda_home.

[3] D. Adams**,** *Inside Tsubame: Japan’s NVIDIA GPU Supercomputer,* Dec. 2008, [Online]. Available: http://www.osnews.com/story/20635/Inside_Tsubame_Japan_s_NVIDIA_GPU_Supercomputer. Last Accessed: May 3, 2009.

[4] R. Farber**,** “Architecture and Design,” in *CUDA, Supercomputing for the Masses: Part 1,* April 2008, [Online]. Available: http://www.ddj.com/architect/207200659.

[5] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips**,** “GPU Computing,” in *Proceedings Of The IEEE*, Vol. 96, No. 5, pp 879-899, May 2008.

[6] J. Nickolls, I. Buck, M Garlanda and K. Skadron, “Scalable Parallel Programming with CUDA,” in *GPUs for Computing*, Vol. 6, No. 2, April 2008.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. "A Performance Study of General Purpose Applications on Graphics Processors using CUDA," in *Journal of Parallel and Distributed Computing*, Elsevier, 68(10):1370-80, Oct. 2008, DOI http://dx.doi.org/10.1016/j.jpdc.2008.05.014.

[8] J. L. Hennesy and D. A. Patterson**,** “The Role of Performance”, in *Computer Organization and Design,* San Mateo, California, Morgan Kaufman Publisher, pp 46 – 77, 1994.

[9] Parboil benchmark suite, [Online]. Available: http://www.crhc.uiuc.edu/impact/parboil.php. Last Accessed: April 28, 2009.

[10] Wen-Mei Hwu, Rodrigues, C., Ryoo, S. and Stratton, J., "Compute Unified Device Architecture Application Suitability," *Computing in Science & Engineering* , vol.11, no.3, pp.16-26, May-June 2009

[11] R. Anderson, E. Biham, and L. Knudsen, “Serpent: a proposal for the advance encryption standard.” [Online]. Available: http://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf, Accessed: July 1, 2009.

[12] D. A Osvik, “Speeding up serpent,” in AES Candidate Conference, New York, NY, USA, April 2000.

[13] B. Gladman, http://gladman.plushost.co.uk/oldsite/cryptography_technology /serpent/index.php, Accessed: July 26, 2009.

[14] S. A. Manavski, “CUDA compatible GPU as an efficient hardware accelerator for AES cryptography,” in *IEEE International Conference on Signal Processing and Communication*, ICSPC 2007, Nov. 2007, pp. 65–68.

[15] Y. Yeom, Y. Cho, and M. Yung, “High-Speed Implementations of Block Cipher ARIA Using Graphics Processing Units,” in *Proceedings of the 2008 international Conference on Multimedia and Ubiquitous Engineering* (April 24 - 26, 2008). MUE. IEEE Computer Society, Washington, DC, 271-275. 2008.

[16] Boghosian, B.M., "A look at lattice boltzmann equations [Book Review]," *Computing in Science & Engineering* , vol.5, no.2, pp. 86-87, Mar/Apr 2003

[17] Weibin Guo, Cheqing Jin and Jianhua Li, "High Performance Lattice Boltzmann Algorithms for Fluid Flows," *Information Science and Engieering, 2008. ISISE '08. International Symposium on*, vol.1, no., pp.33-37, 20-22 Dec. 2008.

[18] Williams, S., Carter, J., Oliker, L., Shalf, J. and Yelick, K., "Lattice Boltzmann simulation optimization on leading multicore platforms," *Parallel*

and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on , vol., no., pp.1-14, 14-18 April 2008.

[19] Jifu Zhou, Chengwen Zhong, Jianfei Xie and Shiqun Yin, "Multiple-GPUs Algorithm for Lattice Boltzmann Method," *Information Science and Engieering, 2008. ISISE '08. International Symposium on* , vol.2, no., pp.793-796, 20-22 Dec. 2008.

[20] OpenLB (Open source lattice Boltzmann method), [Online]. Available: http://www.lbmethod.org/openlb/index.html. Last Accessed: October 19, 2009.

[21] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, “Fast computation of database operations using graphics processors,” in *Proc. of SIGMOD*, 2004.

[22] Intel Corporation, “Intel Software Development Products,” in *Quick-Reference Guide to Optimization with Intel® Compilers Version 11*, Accessed: April 16, 2009, [Online]. Available: cache-www.intel.com/cd/00/00/22/23/222300_
222300.pdf

[23] Standard Performance Evaluation Corporation, SPEC CPU2006 Results, [Online]. Available: http://www.spec.org/cpu2006/results/

APPENDICES

Appendix A

Publication

1. Anas Mohd Nazlee, Fawnizu Azmadi Hussin and Noohul Basheer Zain Ali, "*Serpent Encryption Algorithm Implementation on Compute Unified Device Architecture (CUDA)*", In Proceeding IEEE Student Conference on Research and Development, November 16-18, 2009.

Appendix B

System Information

CPU Information

```
processor   : 0
vendor_id   : GenuineIntel
cpu family  : 6
model       : 15
model name  : Intel(R) Core(TM)2 Duo CPU     E4500  @ 2.20GHz
stepping    : 13
cpu MHz           : 2200.000
cache size  : 2048 KB
physical id : 0
siblings    : 2
core id           : 0
cpu cores   : 2
apicid            : 0
initial apicid    : 0
fdiv_bug    : no
hlt_bug           : no
f00f_bug    : no
coma_bug    : no
fpu         : yes
fpu_exception     : yes
cpuid level : 10
wp          : yes
flags       : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe nx
lm constant_tsc arch_perfmon pebs bts pni dtes64 monitor ds_cpl est
tm2 ssse3 cx16 xtpr pdcm lahf_lm
bogomips    : 4400.11
clflush size      : 64
power management:

processor   : 1
vendor_id   : GenuineIntel
cpu family  : 6
model       : 15
model name  : Intel(R) Core(TM)2 Duo CPU     E4500  @ 2.20GHz
stepping    : 13
cpu MHz           : 2200.000
cache size  : 2048 KB
physical id : 0
siblings    : 2
core id           : 1
cpu cores   : 2
apicid            : 1
initial apicid    : 1
fdiv_bug    : no
hlt_bug           : no
f00f_bug    : no
coma_bug    : no
fpu         : yes
fpu_exception     : yes
cpuid level : 10
```

```
wp          : yes
flags       : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe nx
lm constant_tsc arch_perfmon pebs bts pni dtes64 monitor ds_cpl est
tm2 ssse3 cx16 xtpr pdcm lahf_lm
bogomips    : 4399.69
clflush size      : 64
power management:
```

Graphic Card Information

```
CUDA Device Query (Runtime API) version (CUDART static linking)
There is 1 device supporting CUDA

Device 0: "GeForce GTX 260"
  CUDA Driver Version:                           2.30
  CUDA Runtime Version:                          2.30
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         3
  Total amount of global memory:                 938803200 bytes
  Number of multiprocessors:                     27
  Number of cores:                               216
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.24 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     Yes
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple
host threads can use this device simultaneously)
```

Appendix C

Parboil Benchmark data

Compiler	Intel C/C++ Compiler 11.0
Compiler Flags	-O3 -xT

PARBOIL BENCHMARK

Benchmark		IO	GPU	Copy	Compute	Score
Base	cp	0.010220	0.000000	0.000000	38.157044	0.026207
	rpes	0.017431	0.000000	0.000000	43.362025	0.023062
	pns	0.000171	0.000000	0.000000	49.227340	0.020314
	sad	0.184050	0.000000	0.000000	0.118138	8.464677
	mri-fhd (small)	0.001230	0.000000	0.000000	0.973062	1.027684
	mri-fhd (large)	0.009014	0.000000	0.000000	5.402111	0.185113
	mri-q (small)	0.001280	0.000000	0.000000	2.856392	0.350092
	mri-q (large)	0.008727	0.000000	0.000000	15.385966	0.064994
	tpacf	0.478070	0.000000	0.000000	105.746986	0.009457
CPU	cp	0.010238	0.000000	0.000000	14.119911	0.070822
	pns	0.000169	0.000000	0.000000	39.126251	0.025558
	sad	0.187726	0.000000	0.000000	0.055288	18.087108
	mri-fhd (small)	0.001307	0.000000	0.000000	1.077504	0.928071
	mri-fhd (large)	0.008613	0.000000	0.000000	5.751265	0.173875
	mri-q (small)	0.009305	0.000000	0.000000	1.010185	0.989918
	mri-q (large)	0.008709	0.000000	0.000000	5.371924	0.186153

NOTES:

1) IO is the time taken for the Input/Output of the system

2) GPU is the time taken for the program execution inside the GPU

3) Copy is the time taken for the program to be copied inside the shared memory of the GPU

4) Compute is the time taken for the program execution inside the CPU

5) Score is defined as 1/(GPU+Copy+Compute)

Compiler	GNU C Collection 4.1.2
Compiler Flags	-O3 -msse3

PARBOIL BENCHMARK

Benchmark		IO	GPU	Copy	Compute	Score
Base	cp	0.032949	0.000000	0.000000	501.844121	0.001993
	rpes	0.017617	0.000000	0.000000	150.546424	0.006642
	pns	0.000169	0.000000	0.000000	71.271529	0.014031
	sad	0.188217	0.000000	0.000000	0.160405	6.234220
	mri-fhd (small)	0.034531	0.000000	0.000000	6.393511	0.156409
	mri-fhd (large)	0.082410	0.000000	0.000000	34.263895	0.029185
	mri-q (small)	0.022346	0.000000	0.000000	6.068583	0.164783
	mri-q (large)	0.073838	0.000000	0.000000	32.603966	0.030671
	tpacf	1.466546	0.000000	0.000000	78.562207	0.012729
CPU	cp	0.012978	0.000000	0.000000	502.250742	0.001991
	pns	0.000166	0.000000	0.000000	52.458300	0.019063
	sad	0.183087	0.000000	0.000000	0.059477	16.813222
	mri-fhd (small)	0.001319	0.000000	0.000000	6.461661	0.154759
	mri-fhd (large)	0.009133	0.000000	0.000000	34.427118	0.029047
	mri-q (small)	0.001292	0.000000	0.000000	6.192104	0.161496
	mri-q (large)	0.009277	0.000000	0.000000	33.035517	0.030270

NOTES:

1) IO is the time taken for the Input/Output of the system

2) GPU is the time taken for the program execution inside the GPU

3) Copy is the time taken for the program to be copied inside the shared memory of the GPU

4) Compute is the time taken for the program execution inside the CPU

5) Score is defined as 1/(GPU+Copy+Compute)

Compiler	Intel C/C++ Compiler 11.0
Compiler Flags	-O3 -xT -ipo

PARBOIL BENCHMARK

Benchmark		IO	GPU	Copy	Compute	Score
Base	cp	0.061839	0.000000	0.000000	36.055435	0.027735
	rpes	0.034432	0.000000	0.000000	33.804972	0.029581
	pns	0.000467	0.000000	0.000000	46.818074	0.021359
	sad	0.169085	0.000000	0.000000	0.140549	7.114956
	mri-fhd (small)	0.038362	0.000000	0.000000	1.004024	0.995992
	mri-fhd (large)	0.072529	0.000000	0.000000	5.523443	0.181046
	mri-q (small)	0.037232	0.000000	0.000000	2.915938	0.342943
	mri-q (large)	0.065463	0.000000	0.000000	15.476124	0.064616
	tpacf	1.503969	0.000000	0.000000	103.854404	0.009629
CPU	cp	0.102240	0.000000	0.000000	13.809612	0.072413
	pns	0.000175	0.000000	0.000000	39.249689	0.025478
	sad	0.152698	0.000000	0.000000	0.055155	18.130723
	mri-fhd (small)	0.001271	0.000000	0.000000	1.077723	0.927882
	mri-fhd (large)	0.009012	0.000000	0.000000	5.750769	0.173890
	mri-q (small)	0.001245	0.000000	0.000000	1.007732	0.992327
	mri-q (large)	0.009077	0.000000	0.000000	5.374488	0.186064

NOTES:

1) IO is the time taken for the Input/Output of the system

2) GPU is the time taken for the program execution inside the GPU

3) Copy is the time taken for the program to be copied inside the shared memory of the GPU

4) Compute is the time taken for the program execution inside the CPU

5) Score is defined as 1/(GPU+Copy+Compute)

CPU Frequency	2200 MHz
Compiler (Device)	Nvidia CUDA Compiler 2.0
Compiler (Host)	GNU Compiler Collection 4.1.2

PARBOIL BENCHMARK

Benchmark		IO	GPU	Copy	Compute	Score
CUDA	cp	0.013888	0.187609	0.129066	0.004418	3.114363
	rpes	0.022802	0.177085	0.176380	0.206931	1.784452
	pns	0.015593	2.140182	0.185902	0.000318	0.429848
	sad	0.202321	0.001066	0.126997	0.005020	7.514108
	mri-fhd (small)	0.001408	0.006426	0.112163	0.000152	8.421691
	mri-fhd (large)	0.008662	0.031053	0.118984	0.000384	6.648008
	mri-q (small)	0.001393	0.005103	0.111635	0.000105	8.558493
	mri-q (large)	0.009001	0.024938	0.118441	0.000354	6.957345
	tpacf	1.349266	1.225661	0.114639	0.023142	0.733438
Geomean Score		3.279578				

NOTES:

1) IO is the time taken for the Input/Output of the system

2) GPU is the time taken for the program execution inside the GPU

3) Copy is the time taken for the program to be copied inside the shared memory of the GPU

4) Compute is the time taken for the program execution inside the CPU

5) Score is defined as 1/(GPU+Copy+Compute)

CPU Frequency	1200 MHz
Compiler (Device)	Nvidia CUDA Compiler 2.0
Compiler (Host)	GNU Compiler Collection 4.1.2

PARBOIL BENCHMARK

Benchmark		IO	GPU	Copy	Compute	Score
CUDA	cp	0.053813	0.185729	0.144260	0.008053	2.958212
	rpes	0.042267	0.172555	0.200605	0.364360	1.355895
	pns	0.000397	2.145531	0.237047	0.000635	0.419602
	sad	0.356111	0.001072	0.158113	0.000933	6.245394
	mri-fhd (small)	0.005235	0.006495	0.137339	0.000253	6.940251
	mri-fhd (large)	0.073647	0.031099	0.145790	0.000678	5.631677
	mri-q (small)	0.012598	0.005145	0.137375	0.000185	7.007463
	mri-q (large)	0.068388	0.024980	0.144972	0.000636	5.862077
	tpacf	1.799922	1.225601	0.142054	0.041766	0.709511
Geomean Score		2.838817				

NOTES:

1) IO is the time taken for the Input/Output of the system

2) GPU is the time taken for the program execution inside the GPU

3) Copy is the time taken for the program to be copied inside the shared memory of the GPU

4) Compute is the time taken for the program execution inside the CPU

5) Score is defined as 1/(GPU+Copy+Compute)

Appendix D

Serpent Encryption Algorithm Implementation

Serpent Encryption Algorithm Data Structure

```
// Data Structure
typedef struct {
   uint32_t x0, x1, x2, x3;
}SER_BLOCK;

typedef struct {
   int keyLen;
   uint32_t rkey[8];
}RAW_KEYS;

typedef struct {
   uint32_t x0, x1, x2, x3;
}SUBKEY;

typedef struct {
   SUBKEY k[33];
}SER_KEY;
```

Serpent Encryption Algorithm CPU Source Code

```
void cpu_serpent(SER_BLOCK *pt2ct, RAW_KEYS *keys) {

  SER_KEY skey;
  const int N = ENCRYPT_BLOCKS;
  int idx;

  for(idx = 0; idx<N; idx++) {
   skey = makeKey(keys[idx]);
   pt2ct[idx] = encrypt(pt2ct[idx], skey);
  }

}

int main(void) {

   SER_BLOCK *host_databuffer;
   RAW_KEYS *keycpu;

//Data initialization
cpu_serpent(host_databuffer, keycpu);

}
```

Serpent Encryption Algorithm CUDA source code

```
__global__ void cuda_serpent(SER_BLOCK *pt2ct, RAW_KEYS *keys) {
   const uint32_t idx = (blockIdx.x * blockDim.x + threadIdx.x);
   __shared__ SER_KEY skey;

   skey = cuda_makeKey(keys[idx]);
   pt2ct[idx] = cuda_encrypt(pt2ct[idx], skey);
}

int serpent_encrypt(unsigned char* inbuffer, size_t MemSize, unsigned
char* outbuffer, unsigned char* keycpu, size_t KeySize) {

   SER_BLOCK *gpubuffer;
   RAW_KEYS *key;
   size_t BufferBlocks, NumBlock, TransferSize;



   BufferBlocks = KeySize / sizeof(RAW_KEYS);
   if ((int)BufferBlocks <= 0 ) {
      printf("ERROR! Not Enough BufferBlock! (%d)", BufferBlocks);
      return 0;
   }



   cudaMalloc((void **) &gpubuffer, MemSize);
   cudaMalloc((void **) &key, KeySize);



   while (BufferBlocks > 0) {
      TransferSize = BufferBlocks > BLOCKBUFFER_SIZE ?
BLOCKBUFFER_SIZE : BufferBlocks;
      NumBlock = TransferSize / THREADPERBLOCK;
      dim3 dimGrid(NumBlock);
      dim3 NumThread(THREADPERBLOCK);

      cudaMemcpy(key, keycpu, TransferSize * sizeof(RAW_KEYS),
cudaMemcpyHostToDevice);
      cudaMemcpy(gpubuffer, inbuffer, TransferSize *
sizeof(SER_BLOCK), cudaMemcpyHostToDevice);
      cuda_serpent<<<dimGrid, NumThread>>>(gpubuffer, key);
      cudaThreadSynchronize();

      checkCUDAError("kernel invocation");

      // device to host copy
      cudaMemcpy( outbuffer, gpubuffer, TransferSize *
sizeof(SER_BLOCK), cudaMemcpyDeviceToHost );
      // Check for any CUDA errors
      checkCUDAError("memcpy");

      // Aligning data according to buffer
      inbuffer += TransferSize * sizeof(SER_BLOCK);
      outbuffer += TransferSize * sizeof(SER_BLOCK);
      keycpu += TransferSize * sizeof(SER_KEY);
      BufferBlocks -= TransferSize;
    }
   // Free Device Memory
   cudaFree(gpubuffer);
   cudaFree(key);

   return 1;
}


int main(void) {

   SER_BLOCK *host_databuffer;
   RAW_KEYS *keycpu;
   size_t MemSize, KeySize;
//Memory Allocation
//Data initialization
enc = serpent_encrypt((unsigned char*) host_databuffer, MemSize,
(unsigned char*) host_databuffer, (unsigned char*) keycpu, KeySize);

}
```

Appendix E

Lattice Boltzmann Method Implementation

Lattice Boltzmann Method Data Structure

```
//CPU Data structure
typedef struct {
      int lx; //nodes number in axis x
      int ly; //nodes number in axis y
      int n; //lattice dimension elements
      bool **obst; //Obstacle Array lx * ly
      double ***node; //n-speed lattice  n * lx * ly
      double ***temp; //temporarely storage of fluid densities
} s_lattice;
```

```
//CUDA Data Structure
typedef struct{
      double n[9];
} Node;

//lattice structure
typedef struct {
      int lx; //nodes number in axis x
      int ly; //nodes number in axis y
      int **obst; //Obstacle Array lx * ly
      Node **node; //n-speed lattice  n * lx * ly
      Node **temp; //temporarely storage of fluid densities
} s_lattice;
```

Lattice Boltzmann Method CPU Source Code

```
int main(int argc, char **argv) {

//Data Initialization

for (time = 0; time < properties->t_max; time++) {

      redistribute(lattice, properties->accel, properties->density);

      propagate(lattice);

      bounceback(lattice);

      relaxation(lattice, properties->density, properties->omega);

      }

}
```

Lattice Boltzmann CUDA Source Code

```
__global__ void lb_cuda_kernel(int *dev_obst, Node *dev_node, Node
*dev_temp, int lx, int ly, double accel,
                        double density, double omega)
{
      int idx = blockDim.x*blockIdx.x+threadIdx.x;
      int idy = blockIdx.y;
      int tx = threadIdx.x;

      //Redistribute
      if(idx == 0)
        dev_node[idx+lx*idy]=redistribute(dev_obst[idx+lx*idy],
dev_node[idx+lx*idy], accel, density);

      //Propagate
      int x_e, x_w, y_n, y_s;
      //compute upper and right next neighbour nodes
      x_e = (idx + 1)%lx;
      y_n = (idy + 1)%ly;

      //compute lower and left next neighbour nodes
      x_w = (idx - 1 + lx)%lx;
      y_s = (idy - 1 + ly)%ly;
      //density propagation

      //zero
      dev_temp[idx+lx*idy].n[0] = dev_node[idx+lx*idy].n[0];
      //east
      dev_temp[x_e+lx*idy].n[1] = dev_node[idx+lx*idy].n[1];
      //north
      dev_temp[idx+lx*y_n].n[2] = dev_node[idx+lx*idy].n[2];
      //west
      dev_temp[x_w+lx*idy].n[3] = dev_node[idx+lx*idy].n[3];
      //south
      dev_temp[idx+lx*y_s].n[4] = dev_node[idx+lx*idy].n[4];
      //north-east
      dev_temp[x_e+lx*y_n].n[5] = dev_node[idx+lx*idy].n[5];
      //north-west
      dev_temp[x_w+lx*y_n].n[6] = dev_node[idx+lx*idy].n[6];
      //south-west
      dev_temp[x_w+lx*y_s].n[7] = dev_node[idx+lx*idy].n[7];
      //south-east
      dev_temp[x_e+lx*y_s].n[8] = dev_node[idx+lx*idy].n[8];

      //Bounce Back
      if (dev_obst[idx+lx*idy] == 1)
            dev_node[idx+lx*idy]=bounceback(dev_node[idx+lx*idy],
dev_temp[idx+lx*idy]);

      //Relaxation
      else  {
            int i;
            double c_squ = 1.0 / 3.0;
            double t_0 = 4.0 / 9.0;
            double t_1 = 1.0 / 9.0;
            double t_2 = 1.0 / 36.0;
            __shared__ double u_x[THREADS];
            __shared__ double u_y[THREADS];
            __shared__ double u_squ[THREADS];
            __shared__ double d_loc[THREADS];
            __shared__ Node u_n[THREADS], n_equ[THREADS];

            d_loc[tx] = 0.0;
            for (i = 0; i < 9; i++) {
                  d_loc[tx] += dev_temp[idx+lx*idy].n[i];
                  __syncthreads();
            }

            //x-, and y- velocity components
            u_x[tx] = (dev_temp[idx+lx*idy].n[1] +
dev_temp[idx+lx*idy].n[5] + dev_temp[idx+lx*idy].n[8] -
(dev_temp[idx+lx*idy].n[3] + dev_temp[idx+lx*idy].n[6] +
dev_temp[idx+lx*idy].n[7])) / d_loc[tx];

            u_y[tx] = (dev_temp[idx+lx*idy].n[2] +
dev_temp[idx+lx*idy].n[5] + dev_temp[idx+lx*idy].n[6] -
(dev_temp[idx+lx*idy].n[4] + dev_temp[idx+lx*idy].n[7] +
dev_temp[idx+lx*idy].n[8])) / d_loc[tx];

            __syncthreads();
            //square velocity
            u_squ[tx] = u_x[tx] * u_x[tx] + u_y[tx] * u_y[tx];
            //n- velocity components
            //only 3 speeds would be necessary
            u_n[tx].n[1] = u_x[tx];
            u_n[tx].n[2] = u_y[tx];
            u_n[tx].n[3] = -u_x[tx];
            u_n[tx].n[4] = -u_y[tx];
            u_n[tx].n[5] = u_x[tx] + u_y[tx];
            u_n[tx].n[6] = -u_x[tx] + u_y[tx];
            u_n[tx].n[7] = -u_x[tx] - u_y[tx];
            u_n[tx].n[8] = u_x[tx] - u_y[tx];

            //zero velocity density
            n_equ[tx].n[0] = t_0 * d_loc[tx] * (1.0 - u_squ[tx] /
(2.0 * c_squ));

            //axis speeds: factor: t_1
            n_equ[tx].n[1] = t_1 * d_loc[tx] * (1.0 + u_n[tx].n[1] /
c_squ + u_n[tx].n[1] * u_n[tx].n[1] / (2.0 * c_squ * c_squ) -
u_squ[tx] / (2.0 * c_squ));

            n_equ[tx].n[2] = t_1 * d_loc[tx] * (1.0 + u_n[tx].n[2] /
c_squ + u_n[tx].n[2] * u_n[tx].n[2] / (2.0 * c_squ * c_squ) -
u_squ[tx] / (2.0 * c_squ));

            n_equ[tx].n[3] = t_1 * d_loc[tx] * (1.0 + u_n[tx].n[3] /
c_squ + u_n[tx].n[3] * u_n[tx].n[3] / (2.0 * c_squ * c_squ) -
u_squ[tx] / (2.0 * c_squ));

            n_equ[tx].n[4] = t_1 * d_loc[tx] * (1.0 + u_n[tx].n[4] /
c_squ + u_n[tx].n[4] * u_n[tx].n[4] / (2.0 * c_squ * c_squ) -
u_squ[tx] / (2.0 * c_squ));

            //diagonal speeds: factor t_2
            n_equ[tx].n[5] = t_2 * d_loc[tx] * (1.0 + u_n[tx].n[5] /
c_squ + u_n[tx].n[5] * u_n[tx].n[5] / (2.0 * c_squ * c_squ) -
u_squ[tx] / (2.0 * c_squ));
```

```
            n_equ[tx].n[6] = t_2 * d_loc[tx] * (1.0 + u_n[tx].n[6] /
c_squ + u_n[tx].n[6] * u_n[tx].n[6] / (2.0 * c_squ * c_squ) -
u_squ[tx] / (2.0 * c_squ));

            n_equ[tx].n[7] = t_2 * d_loc[tx] * (1.0 + u_n[tx].n[7] /
c_squ + u_n[tx].n[7] * u_n[tx].n[7] / (2.0 * c_squ * c_squ) -
u_squ[tx] / (2.0 * c_squ));

            n_equ[tx].n[8] = t_2 * d_loc[tx] * (1.0 + u_n[tx].n[8] /
c_squ + u_n[tx].n[8] * u_n[tx].n[8] / (2.0 * c_squ * c_squ) -
u_squ[tx] / (2.0 * c_squ));


            //relaxation step
            for (i = 0; i < 9; i++) {
                  dev_node[idx+lx*idy].n[i] =
dev_temp[idx+lx*idy].n[i] + omega * (n_equ[tx].n[i] -
dev_temp[idx+lx*idy].n[i]);

                  }
            }


}

void lb_cuda(s_lattice *l, int t_max, double accel, double density,
double omega) {
      int i;
      int x,y;
      //device variables
      int *dev_obst;
      Node *dev_node;
      Node *dev_temp;
      Node *h_node;

      cudaMallocHost((void **)&h_node, l->lx*l->ly*sizeof(Node));
      //device memory allocation
      CUDA_SAFE_CALL(cudaMalloc((void **)&dev_obst, l->lx*l-
>ly*sizeof(int)));
      CUDA_SAFE_CALL(cudaMalloc((void **)&dev_node, l->lx*l-
>ly*sizeof(Node)));
      CUDA_SAFE_CALL(cudaMalloc((void **)&dev_temp, l->lx*l-
>ly*sizeof(Node)));
//printf("p_obst = %d\n", pitch_obst);
//printf("p_node = %d\n", pitch_node);
//printf("p_temp = %d\n", pitch_temp);

      //memory copy
      CUDA_SAFE_CALL(cudaMemset(dev_obst, 0, l->lx*l-
>ly*sizeof(int)));
      CUDA_SAFE_CALL(cudaMemset(dev_node, 0, l->lx*l-
>ly*sizeof(Node)));
      CUDA_SAFE_CALL(cudaMemset(dev_temp, 0, l->lx*l-
>ly*sizeof(Node)));
      //kernel parameter
      dim3 dimBlock(THREADS);
      dim3 dimGrid(l->lx/THREADS, l->ly);
      lb_cuda_initial<<<dimGrid, dimBlock>>>(dev_obst, dev_node, l-
>lx, l->ly, density);
      cudaThreadSynchronize();
      for(i = 0; i < t_max; i++) {
            lb_cuda_kernel<<<dimGrid, dimBlock>>>(dev_obst, dev_node,
dev_temp, l->lx, l->ly, accel, density, omega);
            cudaThreadSynchronize();
      }
printf("pass kernel\n");
      //copy from device
      //CUDA_SAFE_CALL(cudaMemcpy(h_obst, dev_obst, l->lx*l-
>ly*sizeof(int), cudaMemcpyDeviceToHost));
      CUDA_SAFE_CALL(cudaMemcpy(h_node, dev_node, l->lx*l-
>ly*sizeof(Node), cudaMemcpyDeviceToHost));

      for(y=0;y<l->ly;y++)
        for(x=0;x<l->lx;x++)
          for(i=0;i<9;i++) {
            l->node[x][y].n[i] = h_node[x+l->lx*y].n[i];
          }
      //free memory allocation
      cudaFree(dev_obst);
      cudaFree(dev_node);
      cudaFree(dev_temp);
      cudaFreeHost(h_node);

}


int main(int argc, char **argv) {

//Data initialization

lb_cuda(lattice, properties->t_max, properties->accel, properties-
>density, properties->omega);

}
```