1-1-2014

Gpu Optimizing And Accelerating Of Gibbs Ensemble On The Cuda Kepler Architecture

Yuanzhe Li
Wayne State University,

GPU Optimizing and Accelerating Of Gibbs Ensemble On the CUDA Kepler Architecture

by

Yuanzhe Li

THESIS

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

2014

MAJOR: COMPUTER SCIENCE

Approved by:

_____

Advisor Date

DEDICATION

Dedicated to my parents, Zhiyong Li and Tong Sun. To my fiancee, Jingwen Zhang.

ACKNOWLEDGEMENT

This is not a easy job for a beginner to conquer by himself. So, I am deeply grateful to my advisor, Dr. Loren Schwiebert for his help guiding me to the right direction. Without his guidance, I would not have the honor to show my research in this article. Also, I would like to thank Dr. Jeffery Potoff, Eyad Hailat, Kamel Rushaidat and Jason Mick for their support.

In addition, It is my honor to invite Dr. Jeffery Potoff and Dr. Daniel Grosu to serve on my defense committee.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

With the already high demand of big data processing growing rapidly and continuously in recent years, traditional CPUs are no longer keeping pace with the workload for two major reasons: first, the bottleneck of processing speed on a single core and second, the huge demand for power. By solving these two problems, general purpose computation on graphics processing units (*GPGPU*) achieves high performance from multi-core computing. Because of its significantly lower cost, GPGPU computing has become popular and more accessible than other parallel systems.

Nowadays, GPU is a big family consisting of a great variety of processors with different architectures. In this thesis, we focus only on Fermi architecture and Kepler architecture, which are manufactured by NVIDIA. NVIDIA developed an exclusive parallel computing platform and programming model called Compute Unified Device Architecture (*CUDA*). In CUDA, there is an elementary unit called a *warp*. A warp is like a SIMD (Single Instruction Multiple Data) machine, which can only execute one instruction with multiple data at one time. It is also a bundle of 32 threads with which to capture respective data from on-chip or off-chip memory. On the GPU, there is a basic unit for integer and floating-point arithmetic functions called a *core*. The number of cores integrated on the GPU is determined by the compute capability of the device. With this basic understanding of a warp, two of Fermi's properties are clear. One, it is necessary to create tasks in the main function (it is run on the CPU) and dispatch them to the warps on the device (GPU). The other property is that the most efficient circumstance can be achieved when all the cores are kept busy. In other words, applications that can run well on the GPU are data parallel. High performance requires the data to be divided into pieces and operated on by the cores piece by piece, so there should be no data

dependency.

1.1 Thesis Objective

The main objective of this thesis is studying the differences between the CUDA Fermi and Kepler architectures, new features of the CUDA Kepler architecture, and optimization methods in order to get the best performance. The main research goals are as follows:

- Develop a deep appreciation of CUDA architecture to find out the performance bottlenecks in the code.
- Study the existing parallel code of the Gibbs Ensemble example used to simulate molecular systems and realized under the Fermi architecture in order to understand the algorithm and determine which parts can be accelerated.
- Optimize the accelerated code to make full use of the Kepler architecture.
- Compare the running time of the Fermi CUDA code and the Kepler CUDA code.

1.2 Thesis Motivation

The Fermi architecture provides significantly better performance than older GPUs [3] but has some hardware constraints that reduce the performance. One is that warps are twice as large as any block of functional units, which is where the *shader clock* comes in. Therefore, with Fermi, a warp would be split up and executed over 2 cycles of the shader clock. The other limitation is that a GPU function can only be invoked from a CPU function, which means all the results that are needed before starting the next GPU function have to be returned to the CPU when a GPU function is finished. These two restrictions result in two kinds of latency on data processing corresponding to the issues above. The first type of latency is caused by shared memory, which is used to share

data among the threads. The second type of latency is caused by the data transmission between CPU and GPU.

These two limitations are performance bottlenecks caused by the architecture, so they cannot be improved under Fermi. However, the situation may be different on the Kepler architecture. As the next generation of the CUDA family, Kepler has some exclusive hardware features that differ from Fermi. It is possible that with improved architecture, the bottlenecks may be resolved. To determine how any of the changes on Kepler architecture can be used to improve performance, deep research on the Kepler architecture is required.

1.3 Thesis Organization

To accomplish the goals of this thesis, the second chapter gives a brief introduction to the Kepler architecture to illustrate the hardware changes. An introduction to the relevant features of Kepler as well as what benefits these features can contribute will also be discussed in chapter 2. In chapter 3, the molecular simulation example will be analyzed together with a succinct explanation of the Gibbs Ensemble. Chapter 4, the core of this thesis, describes my research contributions from this experiment. As the last part of the experiment, chapter 5 contains all the results and makes the performance comparisons to show the speedup. The conclusion in chapter 6 summarizes the progress and highlights where further improvements can be sought.

CHAPTER 2

TECHNICAL BACKGROUND

The Fermi architecture is the penultimate NVIDIA architecture. It is the first computational GPU in the world which embeds all the experience gained from the previous vision and experience [3]. The utilization of the Third Generation Streaming Multiprocessor (*SM*) is the key architectural highlight of Fermi (shown in Figure 2.1). Thirty-two CUDA cores and a dual warp scheduler are integrated into each SM to process data and simultaneously schedule and dispatch instructions from two independent warps. 64 KB of RAM with a configurable partitioning between the shared memory and the L1 cache is also implemented to reduce data transmission overhead and support efficient data sharing.

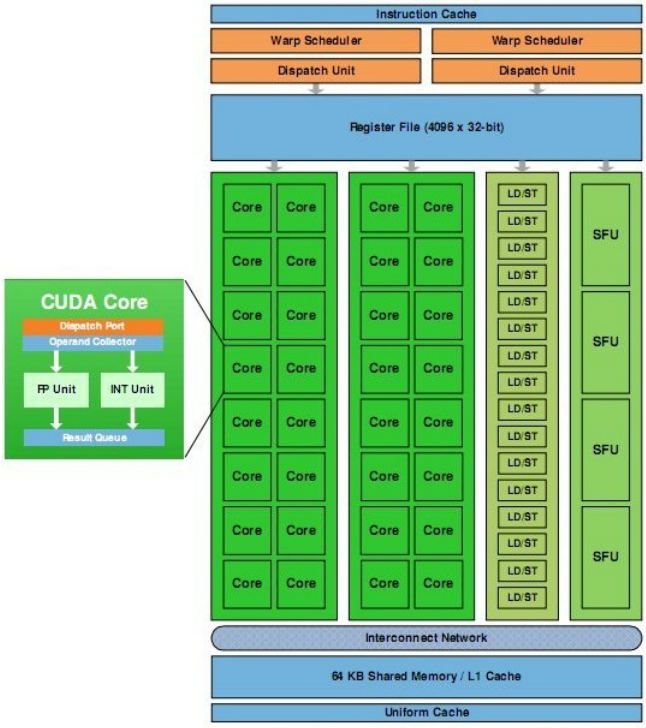

Figure 2.1: Fermi Streaming Multiprocessor [3]

Kepler is the successor of Fermi. The first Kepler processor was announced by NVIDIA in March 2012. Compared with the Fermi architecture, Kepler has some significant improvements in architecture. Figure 2.2 shows the difference in the streaming multiprocessor.

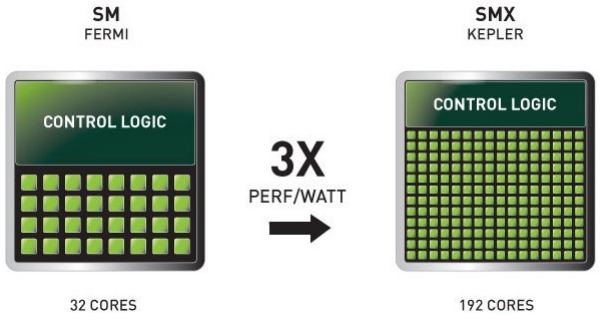

Figure 2.2: Comparison of SM/SMX [5]

The streaming multiprocessor is redefined and also renamed *SMX* on Kepler. In the Kepler GK110 GPU, a SMX has 192 cores, which is four to six times more than Fermi (Fermi 2.0 has 32 processors per SM; Fermi 2.1 has 48); there are two more warp schedulers and also more ALUs as well [5]. With the expansion of operating resources, much higher performance can be achieved without question. Figure 2.3 shows the structure of the SMX.

2.1 Introduction to Kepler K20c

The NVIDIA Tesla K20c card is a sub-product of the Tesla K20 equipped with a cooling part. The K20 is based on one GK110 GPU, which integrates 2496 cores in total (find more details from Figure 2.4). NVIDIA Tesla GPUs are designed for high-performance computing; they deliver the best performance and power efficiency for seismic processing, financial computing, computational chemistry and physics, data analytics, image processing, and weather modeling [4].

Figure 2.3: Kepler Streaming Multiprocessor [5]

2.2 Two Specific Features of K20c

The GK110 architecture has some new features to gain more parallelism and higher performance. In this thesis, two new features, dynamic parallelism and the warp shuffle function, will be introduced as new strategies to improve the performance of the application compared to what is possible with the Fermi architecture as mentioned in the first chapter.

Dynamic Parallelism

Dynamic parallelism simplifies GPU programming by allowing a CUDA kernel function to create and synchronize nested kernel functions, which means a GPU can dynamically spawn new threads on its own without returning to the CPU. Figure 2.5 shows how dynamic parallelism works.

Dynamic parallelism is not only a powerful tool to make kernels run more parallel but also very easy to realize. The programmer simply launches a kernel function directly from a running kernel. The launching kernel function is called the *parent*, and the function

```
Device 0: "Tesla K20c"
  CUDA Driver Version / Runtime Version          5.5 / 5.5
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 4800 MBytes (5032706048 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Clock rate:                                706 MHz (0.71 GHz)
  Memory Clock rate:                             2600 Mhz
  Memory Bus Width:                              320-bit
  L2 Cache Size:                                 1310720 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536),
3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simu
ltaneously) >
```

Figure 2.4: Device Query of K20c shows all its technical specifications

launched from the parent is called the *child*. A parent can spawn multiple children, and a children function can even have its own children. Thus, a multilevel nested calling stack is allowed.

A simple "Hello World!" example is shown in Figure 2.6. In this example, the main function launched on the CPU launches parentKernel; Next parentKernel launches childKernel first and then prints out "World!". However, parentKernel waits for the completion of childKernel first and then prints out"World!" only after "Hello " is printed by childKernel.

Warp Shuffle Function

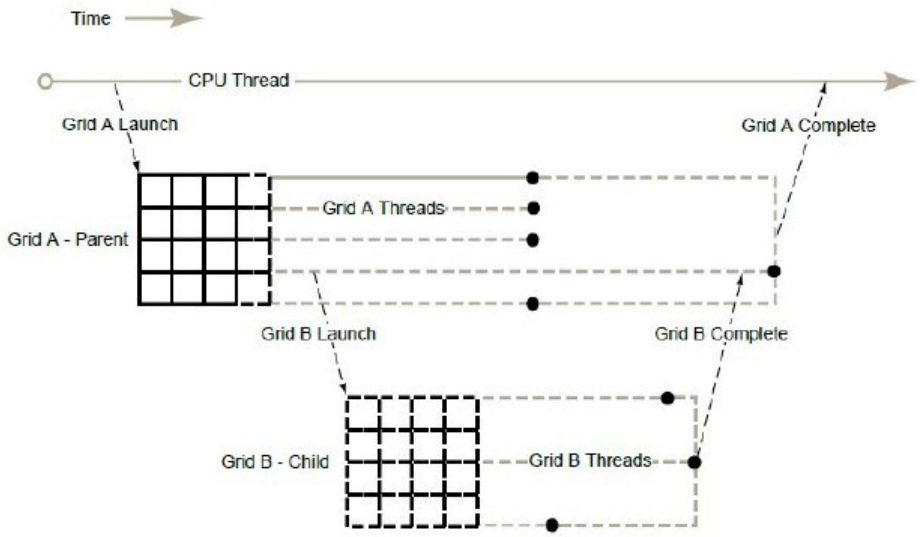

Figure 2.5: Dynamic Parallelism [6]

The warp shuffle function is designed to reduce the overhead of transferring values among threads and the use of the shared memory. Threads within the same warp can share registers with each other by using this function. The warp shuffle function is different from the fetching data from the shared memory. Since threads can directly access the registers of other threads, we avoid the higher delay of accessing shared memory, including any serialization due to multiple threads accessing the same shared memory bank. The programmer can use the functions listed in Figure 2.7 to access registers from the source thread by the caller's thread. There are two restrictions: first, a warp shuffle function can only be used between two threads within the same warp. Second, a warp shuffle function can only exchange 4 bytes of data each time. When 8-byte quantities need to be exchanged, the process must be broken into two separate invocations of a warp shuffle function.

In Figure 2.7, the functions are int type and float type. The type of the function is related to the size of the data. An invocation of a function like double __shfl() is not allowed because a double occupies 8 bytes, which is beyond the capability of any warp shuffle function. To make the function easy to describe, threads within a warp

```
#include <stdio.h>

__global__ void childKernel()
{
    printf("Hello ");
}

__global__ void parentKernel()
{
    // launch child
    childKernel<<<1,1>>>();
    if (cudaSuccess != cudaGetLastError()) {
        return;
    }

    // wait for child to complete
    if (cudaSuccess != cudaDeviceSynchronize()) {
        return;
    }

    printf("World!\n");
}

int main(int argc, char *argv[])
{
    // launch parent
    parentKernel<<<1,1>>>();
    if (cudaSuccess != cudaGetLastError()) {
        return 1;
    }

    // wait for parent to complete
    if (cudaSuccess != cudaDeviceSynchronize()) {
        return 2;
    }

    return 0;
}
```

Figure 2.6: Code example of Dynamic Parallelism [6]

are referred to as *lanes*, and the lane where the required data comes from is called the *source lane*. The ID of the source lane is only given to __shfl(). It is a direct data copy function, which returns the required value held by a source lane. For the other three functions, it is a little different as the source lane does not have to be explicitly specified. In __shfl_up(), the source lane ID is calculated by subtracting the variable *delta* from the caller's lane ID (the lane where the function is invoked). Conversely, in __shfl_down(), the source lane is calculated by adding delta to the caller's lane ID. As indicated by the name, __shfl_xor() calculates the source lane ID by performing a bitwise XOR of the caller's lane ID with the variable *laneMask*. In the latter three functions, there is a *width* variable that determines the boundary that the source lane should be in. If the source lane is outside the boundary, the function just returns the value of the caller's lane. As

```
int __shfl(int var, int srcLane, int width=warpSize);
int __shfl_up(int var, unsigned int delta, int width=warpSize);
int __shfl_down(int var, unsigned int delta, int width=warpSize);
int __shfl_xor(int var, int laneMask, int width=warpSize);

float __shfl(float var, int srcLane, int width=warpSize);
float __shfl_up(float var, unsigned int delta,
                int width=warpSize);
float __shfl_down(float var, unsigned int delta,
                  int width=warpSize);
float __shfl_xor(float var, int laneMask, int width=warpSize);
```

Figure 2.7: Warp Shuffle functions

a result, the value of the width has to be smaller or equal to the warp size and it must be a power of 2.

In the example shown in Figure 2.8, a __shfl_up() is used to collect data from the lanes with lower IDs. The value of the width is 8. Therefore, the source lane ID cannot be lower than the caller's ID, which is 8. Since the value of delta is i, which could be 1, 2, and 4 as these are all within the scope of 8, this function is safe, and the caller lane is able to collect data from source lanes properly.

```
__global__ void scan4() {
    // Seed sample starting value (inverse of lane ID)
    int value = 31 - laneId;

    // Loop to accumulate scan within my partition.
    // Scan requires log2(n) == 3 steps for 8 threads
    // It works by an accumulated sum up the warp
    // by 1, 2, 4, 8 etc. steps.
    for (int i=1; i<=4; i*=2) {
        // Note: shfl requires all threads being
        // accessed to be active. Therefore we do
        // the __shfl unconditionally so that we
        // can read even from threads which won't do a
        // sum, and then conditionally assign the result.
        int n = __shfl_up(value, i, 8);
        if (laneId >= i)
            value += n;
    }

    printf("Thread %d final value = %d\n", threadIdx.x, value);
}
```

Figure 2.8: Code example of Warp Shuffle Function [6]

2.3 Advantages of Kepler K20c

In my research, following are three main advantages that I obtained from Kepler K20c. First, with more resources, it is easy to get much better performance than running on Fermi. The execution time is reduced significantly. Second, the interaction between CPU and GPU can be reduced by using Dynamic Parallelism. It is a good strategy to eliminate the latency from the data transmission between the host and the device. Third, proper use of the warp shuffle can eliminate the latency from data transmission between threads; however, it is not applicable to all conditions. In some specific scenarios, performance suffers when using the warp shuffle.

CHAPTER 3

RELATED WORK

Currently, due to highly accurate results and high-performance computation, computer simulations are considered by researchers to be important substitutes for lab experiments. This chapter presents an overview of the related simulation methods that have been used to create a molecular system. In a typical molecular system, it is normal to evaluate a high density environment with millions of particles that need to be calculated. By realizing the benefits of GPUs, the running time can be reduced significantly by hundreds or even thousands of times under a reasonable power usage.

3.1 Monte Carlo Simulation

Monte Carlo simulation is one of the two most popular approaches that researchers would like to use to study atomistic systems. The other method is Molecular Dynamics, for which there are already several existing codes, and some have been reorganized to support GPU executions, such as NAMD [17]. The existing codes of Molecular Dynamics attract much interest but, in some cases, cannot meet the needs of many biomolecular systems [8]. Conversely, Monte Carlo simulation with the Markov Chain has a big advantage compared with molecular dynamics because it allows the study of open systems [1] due to the fluctuation property that Molecule Dynamics does not support.

Monte Carlo is usually used to predict the interactions among molecules and reflect the local movements of molecules. Monte Carlo methods are a broad class of computational algorithms that rely on generating random samples repeatedly, and they are especially efficient for simulation. The computational cost is really the chief limitation to regular serial computing, which could lead to lengthy execution times if we suppose there are one million particles that need one million iterations per particle. However, it is not

an unrealistic problem size for high performance parallel computing. In previous work, our research group created a model system and evaluated it on GPUs [9, 10, 15]. Monte Carlo simulation is used to randomly change the current configuration of the system and sequence all the configurations following the distribution of the random numbers. By generating millions of random configurations, the desirable properties can be retained and accumulated to get other characteristics of the system, like potential, pressure, etc.

3.2 Lennard-Jones Potential

The *Lennard-Jones potential* is a mathematical model that calculates the interaction between a pair of molecules [11]. The expression is

$$V_{LJ} = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

where ϵ is the depth of the region surrounding a local minimum of potential energy, σ is the distance between the particle and the position where its potential is zero, and r is the distance between particles [18]. These parameters are optimized to reproduce experimental data. The Lennard-Jones potential is used as a function to simulate the potential between two molecules on a computer. It is widely used by researchers because of its straightforward calculation and high-accuracy especially for inert gas molecules though it is more expensive than Ising or hard sphere models because the interactions between particles within a certain cutoff radius must be calculated [8].

3.3 Gibbs Ensemble

In statistics, the Gibbs ensemble is a Markov Chain Monte Carlo algorithm which is used to obtain a sequence of configurations based on random variables in a specified distribution.

For research purposes, the Gibbs ensemble simulation is necessary in order to model a system for calculating vapor-liquid phase coexistence, which is conducted by running

one simulation with two boxes [1, 11]. To evaluate this kind of modeled system, the Monte Carlo simulation, as the crucial method, is taken to randomly change the status of the particles with applied potential. The limitation of the algorithm, which is also my motivation to do the research, is that the computational cost of a dense system could be enormous and the execution time for a large system could be weeks or even months. For this problem, the GPU is the best choice to cut costs and speed up the execution.

An existing GPU code for the Monte Carlo simulation of the Gibbs ensemble was developed in previous work by our research group and was able to achieve some better performance compared with the CPU code [15]. In this research, I enhanced the GPU code to transform the algorithm to run on the NVIDIA Tesla K20c to take advantage of the new features of Dynamic Parallelism and Warp Shuffle.

3.4 Monte Carlo Simulation on Gibbs Ensemble

In the system, there are some variables that need to be set: box volumes, particle numbers of each box, temperature, and chemical potential. Also, some other variables need to be calculated: system energy, system potential, and pressure [16]. For a Gibbs ensemble simulation, there are some variables that are fixed like the total number of particles and the total volume of the system, but others are independent.

The Gibbs ensemble simulation modeled system contains two boxes with particles inside. There are three kinds of movements, explained below, occurring with fixed probabilities that are chosen at the beginning. Each of these movements has its own acceptance requirements; the acceptance conditions actually depend on a random exponential function [16].

1. Particle Displacement: In this move, the system randomly selects a box to conduct the displacement and randomly picks a particle as well. Then it attempts to move the particle to other locations within the same box.

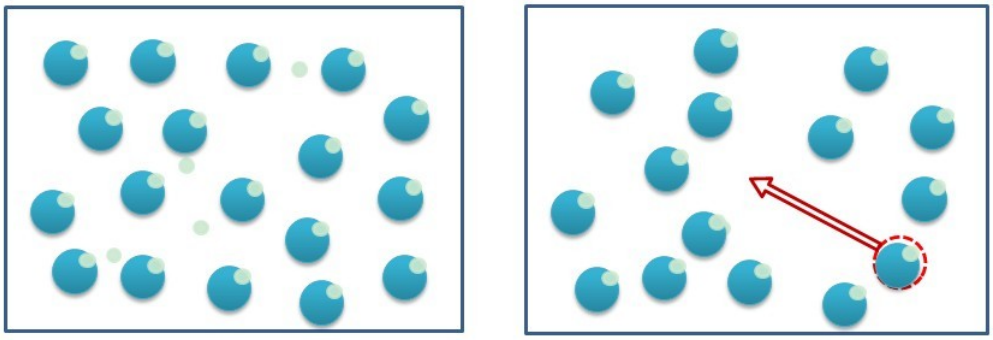

Figure 3.1: The illustration of particle movement inside a box

2. Volume Swap: This move is for both boxes. The sizes of these two boxes would be changed by an equal value but in the opposite direction. An equal and opposite random change in the volume is shown below. The volume swap is a high-cost movement because it would change the positions of all the particles in the box, which requires the energy of all the particles in each box to be recomputed.

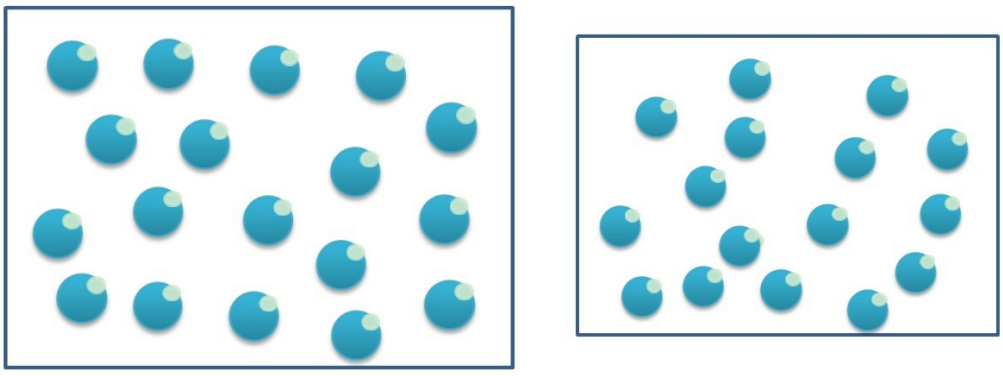

Figure 3.2: The illustration of volume swap; when the size of one box is increased, the size of the other box is decreased.

3. Particle Transfer: The system transfers a randomly selected particle in a randomly chosen box to the other box. In the destination box, a random position is selected to contain the incoming particle. It can be regarded as removing a random particle from one of the two boxes, and adding this particle to a random position of the other box.

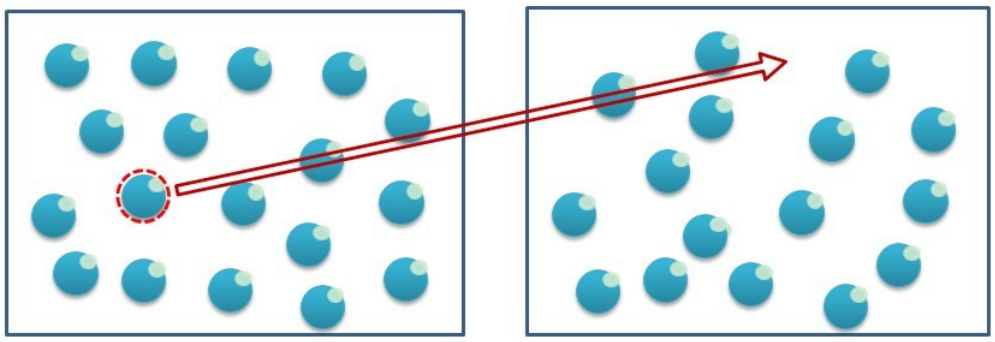

Figure 3.3: An illustration of particle transfer, moving one particle from the source box to the destination box.

If the displacement is accepted, the new configuration, which is based on this change, will replace the old system configuration and be used for the next configuration. If it is not accepted, the new configuration is abandoned and the old retained.

3.5 Method to Implement GEMC

The implementation of the Gibbs ensemble is shown in Figure 3.4. The main loop in Figure 3.4 controls the number of the samples, which should be chosen to be big enough to achieve the desired degree of accuracy. The three movement functions are inside the main loop, but only one movement can be simulated in each iteration. The movement which is to be simulated is determined according to the random value "R" in each iteration. To generate high-quality pseudorandom integers, the Mersenne twister [14] was used in my research. It is the most widely used pseudorandom number generator for these types of simulations, in part because the Mersenne twister has an extremely long periodicity.

```
1: Input: Two boxes of equal size (N) and volume (V)
2: //Main simulation loop
3: for i= 1 to N_steps, step=1 do
4:     //Randomly select a move type
5:     R ← rand()
6:     if (R ≤ N_disp) then
7:         //Attempt atom displacement
8:         //Randomly select a box
9:         Source ← rand()
10:        //Pick a box
11:        if (Source < 0.5 ) then
12:            //(K) Attempt to displace an atom in box 1
13:        else
14:            //(K) Attempt to displace an atom in box 2
15:        end if
16:    else if ( R ≤ ( N_disp + N_vol ) ) then
17:        //(K) Volume Transfer
18:    else
19:        //Attempt atom transfer
20:        //Randomly select a box
21:        Source ← rand()
22:        //Pick a box
23:        if (Source < 0.5 ) then
24:            //(K) Source box is box 1
25:        else
26:            //(K) Source box is box 2
27:        end if
28:    end if
29:    //Solve if the system is in equilibrium
30:    //Periodically write system status to disk
31: end for
```

Figure 3.4: The implementation of the Gibbs Ensemble simulation [8].

At least two memory copy operations are needed per iteration to provide the data that are required by these three functions and also to transfer the generated data back. Although the cost of a single memory copy is tiny, the total cost of transferring the data is still considerable because of the enormous loop.

CHAPTER 4

RESEARCH CONTRIBUTION

The ideal condition of a SIMD-based task executed via massively parallel programming actually is to process the independent data in parallel while utilizing reasonable computational resources. In theory, a program with a higher ratio of occupied threads, which is called the *occupancy*, could achieve better performance [12]. A higher occupancy cannot only reduce the workload of each thread but also hide the memory latency. However, this is not a hard and fast rule; pursuing a higher occupancy or the highest one sometimes makes the performance worse. Once all the memory latency has been hidden, increasing the occupancy further may degrade the performance due to other factors [2]. Therefore, I need to trade off the performances with different parameters, which can give me different versions of the same kernel to consider the most appropriate occupancy for the program.

In my research on achieving the best performance with the NVIDIA Tesla K20c, the speedup is not only from the extra computational resources contributed by the K20c but also from the reduction of some delay existing in the old code. Running all device functions in parallel and shifting data using no shared memory or less shared memory are two approaches, corresponding to Dynamic Parallelism and Warp Shuffle, to achieve significant performance improvement.

4.1 Warp Shuffle Function

The Warp Shuffle operation is a new warp-level intrinsic, which is introduced in the Kepler architecture [7]. The Warp Shuffle operation allows threads within the same warp to exchange data with each other directly without transferring data to the shared memory (transferring the data to the global memory is so much slower that there is no need to

compare it with the shuffle operation and transferring data to shared memory).

4.1.1 Use of Shuffle Function

In the old code (Figure 4.1), which is implemented under the Fermi architecture, a different implementation is required. To transfer the data from one thread to another, whether or not they are in the same warp, it is necessary to copy the data to the shared memory first; then, the destination thread loads the data from the shared memory. Although going through the shared memory is not an expensive action for threads, there are still two steps that need to be done: store and load. Now, with the Warp Shuffle function, it can be cut down to a single step. The threads exchange data directly, and the shared memory can also be released from storing transferred data. Therefore, the new shuffle operation is more efficient in data transferring. Nevertheless, we still need shared memory for the data exchange among threads that are in the same block but are not in the same warp. The Warp Shuffle operation only works on the warp level. As a result, trading off the usage of shared memory and the shuffle operation is important for optimizing performance.

```
cEnergy[threadIdx.x] += cEnergy[threadIdx.x + 16];
cVirial[threadIdx.x] += cVirial[threadIdx.x + 16];
```

Figure 4.1: Data exchange with shared memory only

In Figure 4.1, "cEnergy" and "cVirial" are two arrays stored in shared memory, and the corresponding value of each thread is stored in the arrays by using the thread ID to identify its address. To sum two threads, I have to first store the value held by each thread to the shared array before the exchange and then load the value when the exchange happens.

How about Warp Shuffle functions? The shuffle functions consist of four basic operations: *__shfl*, *__shfl_up*, *__shuf_down*, and *__shfl_xor*. As illustrated in Figure 4.2, I use __shfl_down here to add the latter thread to the prior one with an offset number which is

the interval of these two threads.

```
newVal.ival[0] = __shfl_down(EnerVal.ival[0], 16, 32);
newVal.ival[1] = __shfl_down(EnerVal.ival[1], 16, 32);
EnerVal.dval += newVal.dval;
newVal.ival[0] = __shfl_down(ViriVal.ival[0], 16, 32);
newVal.ival[1] = __shfl_down(ViriVal.ival[1], 16, 32);
ViriVal.dval += newVal.dval;
```

Figure 4.2: Data exchange with Warp Shuffle function

One challenge is that the Warp Shuffle function can only transfer the data with a size no more than 4 bytes. Therefore, it is not possible to transfer a double-precision value via Warp Shuffle directly. Because the shuffle function is based on the registers, the size of the transferred data cannot be more than one register. The register's size for the Kepler GK110 architecture is 32 bits [5]. Therefore, if I want to transfer a double-precision value from one thread to another via a Warp Shuffle function, I have to divide the data equally into two segments, and there will be two registers that each handle one segment, but not the scheme that one register handles both segments with a time division. This is because CUDA's implementation of double precision arithmetic requires that each double variable uses two registers [6]. Hence, in Figure 4.2, the transferred data is divided into ival[0] and ival[1].

```
union ShflUnion{
    double dval;
    int ival[2];
};
```

Figure 4.3: Union type

In my research, I use a union to achieve the data separation. As shown in Figure 4.3, I use two int type variables, which are 4 bytes each, to hold the two equal segments of a double type. It does not matter if the structure of the double variable would be changed

when it is separated into two int variables. This does not affect the result because the two int variables are only used to transfer the data but don't execute any arithmetic.

4.1.2 Data Merging

In my research, since it is necessary to add up the energy of all particles to compute the total energy and each thread in the program processes just one particle in the box, I need to merge the data processed by all threads into a single sum.

To avoid data conflict and finish the entire merging process in a reasonable time, our group carried out a solution which is summing all the data in several steps; in other words, it means performing a parallel reduction operation. To create the reduction, we created one offset, which is half of the number of the total threads and added the data processed by the second half of the threads to the data on the first part. In this way, there is no data conflict because we only added one to another, and no other operations like read or write were performed. Then, we divided the offset by 2 and re-sized the scale of the threads to do the data addition on the newly merged data. We continued doing the same operation in each reduction round until the scale of the threads is reduced to one. By doing so, the data are merged in $\log_2 n$ steps. If we simply added all the data together, n iterations will be needed to complete the reduction because only two threads can add their data together each time to ensure correctness.

This method is also applicable to the Warp Shuffle operation. The difference from the new operation is that the merging process can be treated as two cases and completed by using either shared memory or the Warp Shuffle operation.

4.1.3 Reasonable Use of Shuffle Function

Therefore, another challenge of using Warp Shuffle functions is trading off the usage of shuffle functions and shared memory. The shuffle functions can only work on the warp level, and there could be multiple warps in a block. This results in two different strategies for thread merging.

These two strategies could have significantly different performances because of the different impact they make on the program's parallelism. When there is a large amount of data to be merged into one subtotal, the parallelism and the hidden latency are the most important factors. The general idea is first loading the data into the shared memory, and then merging the data which are loaded by threads step-by-step with a reasonable offset. Therefore, Warp Shuffle functions can be used either at the beginning of the merging process, which is to combine all the threads within the same warp to one specified thread, or at the end of the merging process when all the data have been reduced to the point where the number of remaining values is the same as the size of a warp. Figures 4.5 and 4.6 display examples of these two different merge movements.

In the example shown in Figure 4.5, there are two warps: 0 and 1. At the beginning of this merging method, I combine the data values of the threads in one warp with the corresponding data values in the threads of the other warp. When there are more than two warps, if I keep merging the values of all the warps to the first warp, it will cause an access conflict when multiple threads add the values they hold to the corresponding thread value in a specified warp simultaneously. To avoid this condition, I adjust the offset at each iteration and set it to half the total number of threads (see Figure 4.4).

```
for (int i = offset >> 1;i>=32;i>>=1) {
    if (threadIdx.x < i) {
        cacheEnergy[threadIdx.x] += cacheEnergy[threadIdx.x + i];
        cacheVirial[threadIdx.x] += cacheVirial[threadIdx.x + i];
    }
    __syncthreads();
}
```

Figure 4.4: Example of merge iteration

At each iteration, I only merge the values stored in the lower half of the warps to the upper half to insure only one access per thread each time. Once the results of all the warps have been merged to one warp, just like the second half of Figure 4.5, I use Warp Shuffle functions to merge the thread values in that warp to one thread, and this thread will hold the sum.

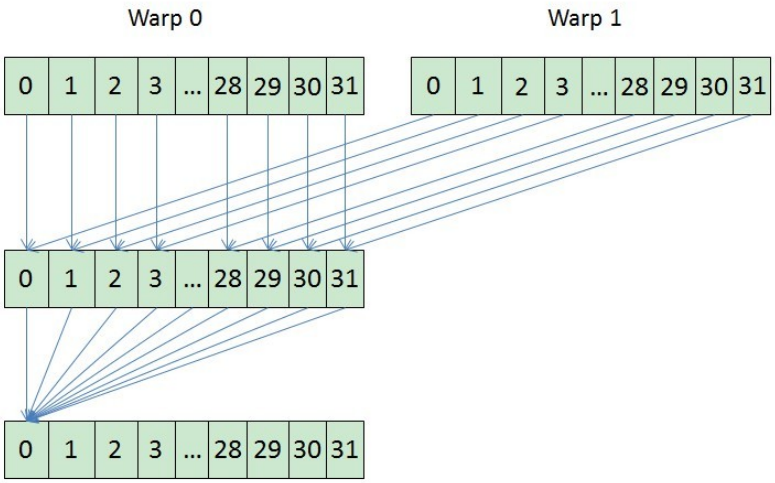

Figure 4.5: Merging thread values starting in shared memory

A drawback to this method is that the shuffle functions are used much less than shared memory, so the data summation process still relies principally on the shared memory. Especially, when there are many warps in a block and a high occupancy is needed, the whole implementation would still waste time on this step.

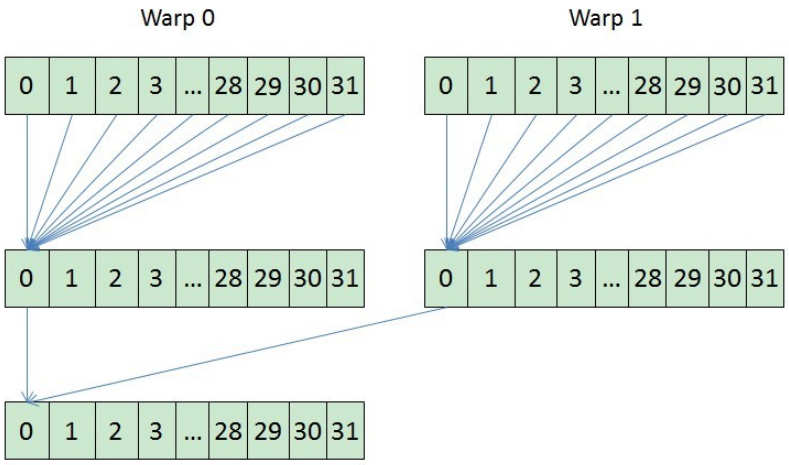

Figure 4.6: Merging thread values starting with the shuffle function

The second strategy, displayed in Figure 4.6, shows the opposite idea. First, I merge the data values of all threads in each warp to one specified thread, usually the first thread in the warp, by using Warp Shuffle functions. After that, each warp contains the partial sum in one thread, which means the shuffle function can no longer be used and shared memory is needed for the remaining merging process.

Comparing the second strategy to the first one, I can use many more shuffle functions in the second one to minimize dependence on the shared memory, especially when there are plenty of warps in the same block. Consequently, implementing the data transfers in the second method should be more efficient due to the higher usage of shuffle functions. Nevertheless, increasing the use of the shuffle functions does not necessarily equate to better performance. A high usage of shuffle functions also reduces the program's parallelism significantly. As shown in Figure 4.6, after merging the partial sums of each warp to their first threads, the number of threads that would still perform tasks reduces to one over thirty-two of the original amount. With many fewer threads like this situation, the running threads are no longer able to totally hide the memory latency. Therefore, the memory latency would become the main bottleneck, which is more serious than the

Table 4.1: Performance comparison of two Warp Shuffle implementing methods.

System Size (particles)	Approach 1 (seconds)	Approach 2 (seconds)
1024	54.946	54.543
2048	59.153	59.167
4096	67.568	67.922
8192	80.844	82.388
16384	125.836	128.24
32768	248.891	255.036
65536	665.288	678.603
131072	2251.97	2290.32
262144	8033.6	8134.39

additional overhead of accessing shared memory.

These two methods actually are two extremes. Based on the collected data in Table 4.1, their performances are close to each other, but the second implementation is slightly slower. It proves that memory latency may appear when there is a significant decrease in the number of threads, and totally hiding the memory latency should be the first objective when optimizing performance. Therefore, the first method is the option I use in my research.

4.2 Dynamic Parallelism

Dynamic Parallelism is also a new feature of the Kepler GK110 architecture [5]. The attractive part of Dynamic Parallelism is that the kernels are now able to spawn new kernels just inside of themselves. This single change provides a wide variety of possibilities for programmers to implement their tasks and gain benefits from them.

4.2.1 Data Transfers

According to the function of Dynamic Parallelism, there is the possibility that all the kernels except the one that has to be invoked from the host could be called from the device directly. This attribute makes it possible to eliminate the intermediate data transfers between the device and host.

In my research, there is a scenario that some of the parameters of the latter kernel functions rely on the results of the prior kernels. This may result in a great demand for global synchronizations of the results. Unfortunately, the synchronization for the entire device is really expensive. Utilizing Dynamic Parallelism can also help me eliminate most of these global synchronizations.

In our group's earlier work, which was on the Fermi architecture, the data had to be copied back to the host after each finished kernel, and it was also necessary to transfer the data to the device before the next kernel function was called. Otherwise, it was impossible for the latter kernel functions to get the updated results. Conversely, it is different on the GK110 architecture. All these data transactions can be removed if Dynamic Parallelism is used. Because all the kernels can be called by a specific kernel with only one thread, the data need to be transferred to this parent kernel at the beginning of the program, and they will be kept on the device through out the whole progress of the kernel, which is also the execution of all child kernels as well. Therefore, once the initialization of the data is done and the initial data have been copied to the device, all the kernel functions can obtain the data directly from the device, and the data can also be updated directly in the global memory. The process is shown in Figure 4.7.

This improvement is a benefit of the new attribute that the spawned kernel functions can retrieve the formal parameters of the parent kernel function so that they could process these variables and update them directly to the global memory of the device. The old Fermi architecture, without Dynamic Parallelism, cannot support the spawned

kernel functions, and all the kernel functions are treated individually right from the host function so that they all use their own parameters which are separate from each other.

```
Host function{

    for moves <-- 1 to 1000000
        1. copy data from cpu to gpu
           Particle_Displacement<<<>>>();
           copy data from gpu to cpu

        2. copy data from cpu to gpu
           Particle_Transfer<<<>>>();
           copy data from gpu to cpu

        3. copy data from cpu to gpu
           Volume_Swap<<<>>>();
           copy data from gpu to cpu
}
```

(1) Implementation under Fermi

```
Host function{
    copy data from cpu to gpu
    parent_function<<<>>>();
}

parent_function{
    for moves <-- 1 to 1000000
        1.  synchronization
            set # of threads used in the function
            Particle_Displacement<<<>>>();
        2.  synchronization
            set # of threads used in the function
            Particle_Transfer<<<>>>();
        3.  synchronization
            set # of threads used in the function
            Volume_Swap<<<>>>();
}
```

(2) Implementation under Kepler

Figure 4.7: The implementations of data transfers on the (1) Fermi and (2) Kepler GK110

The implementation in Figure 4.7 (1) is the method used in the previous work. The data has to be copied back to the host after each of the kernel functions (device_function). The other one, shown in Figure 4.7 (2), uses Dynamic Parallelism. The special_device_function is the only kernel function that is called from the host, and all other kernel functions are called from the GPU. There is no need to copy data, but it might be necessary to add a synchronization after each inner kernel function.

Getting rid of the intermediate data transfers can absolutely improve performance; however, there is also a side effect that this implementation may require global synchronizations after each of the spawned kernel functions. In Figure 4.7 (2), one global synchronization per kernel function is inserted to insure that the results of the function have been updated to the global memory before they are required by the next function. What is worse is the high overhead of global synchronization can significantly lower the performance that was optimized by removing data transfers. Therefore, in my research, the main challenge of Dynamic Parallelism minimizes the use of the global synchroniza-

tion while retaining the speedup saved from data transfers.

4.2.2 Global Synchronization

Global synchronizations are needed after each of the kernel functions because both of the kernel's formal parameters, the number of threads per block and the number of blocks per grid specified in the <<< ... >>> syntax, depend on the results of the previous kernel call. As in Figure 4.8, it is the initialization of the block size and the grid size of the Particle Displacement function that introduces a synchronization problem. The variable Box1Params.NumberParticlesInBox is stored in the global memory and can be retrieved by all the kernels on the device. Therefore, the block size here may be incorrect if it is used for the latter kernel functions before the new calculated Box1Params.NumberParticlesInBox is updated.

```
if ((Box1Params.NumberParticlesInBox) < MAXTHREADSPERBLOCK)
        ThreadsPerBlock = Box1Params.NumberParticlesInBox;
else
        ThreadsPerBlock = MAXTHREADSPERBLOCK;
BlocksPerGrid = (Box1Params.NumberParticlesInBox + ThreadsPerBlock - 1) / ThreadsPerBlock;
```

Figure 4.8: Example of initialization of block size and grid size.

To avoid this kind of race condition, I found that there are two different prototypes which could help. By looking at Figure 4.8, one is assigning a constant value to the block size to keep it unchangeable. Another much different solution is I still keep the parameter changeable; however, I don't change it before the invocation of the kernel function. Instead, I change the block size at the end of the last kernel, which means the block size and the grid size are updated to the global memory inside the running kernel so that when the next kernel is called, all these parameters are guaranteed to be updated earlier, and no race condition is possible.

It is obvious that assigning a constant value to the block size could help. The value of the variable Box1Params.NumberParticlesInBox will change only when a Particle Transfer move is accepted because when it is accepted, there is a particle that

is transferred to another box and the number of particles in the boxes will change. However, with the constant block size and grid size, no matter how much the variable Box1NumberParticlesInBox changes, will not affect the value of these two parameters. In a small Gibbs ensemble system, if the size of a system is smaller than 1024, which is the upper bound of the block size, the size is also the reasonable constant value for the block size because it is fixed and always larger than the number of particles in either of the two boxes. For the large system, the block size can be simply set to 1024.

Assigning a constant value to the parameters is effective, but it is not efficient. The constant value used for the parameters should always be greater than or equal to what it needed. This restriction causes the drawback that the threads with an ID greater than the actual block size are idle throughout the entire function. This wastes resources, which can degrade performance.

The second option allows the kernel call to specify the actual block size and grid size of the kernel functions. It is based on a different strategy. Comparing the original realization in Figure 4.8 and the first option mentioned above, it is clear that there is one point in common that both block size and grid size are initialized before the functions are called. In this case, programmers should always watch for potential asynchronous errors, and the asynchronous errors can only be hidden but cannot be removed completely. As a result, to remove the asynchronous errors completely, the block size and the grid size can no longer be initialized before the kernels are called.

From the definition of *streams* [6], all the kernel functions in the same stream are definitely executed in a sequence. The point here is that the next kernel function will not be called until the current function is finished completely. This same requirement applies to the data processing inside the kernel. If there are any data processing operations, they must be done before the next function is called. Therefore, the data must be saved to the global memory before they are used by other kernel functions. In the Gibbs ensemble

system, when simulating the three movements, the only change to the block size and the grid size happens when a Particle Transfer movement is accepted. Therefore, I only need to recalculate block size and grid size inside this function with the updated parameters if the movement is accepted. The limitation here is that the kernel functions should be in the same stream, but there is also good news that all the spawned kernel functions which have the same parent are in the same stream. In my research, the main functions to model those three movements are all spawned by the same parent. Figure 4.9 gives the pseudo-code of my implementation.

```
__global__ parent(int parameter){
    for moves <-- 1 to 1000000
        if(move type 1)
            Particle_Displacement <<<GridSize, BlockSize>>>(parameter);

        else if(move type 2)
            Volume_Swap <<<GridSize, BlockSize>>>(parameter);

        else (move type 3)
            Particle_Transfer <<<GridSize, BlockSize>>>(parameter);
        .
        .
        .
}

__global__ Particle_Transfer(int parameter){
    .
    .
    .
    if (accepted)
        //change the value of parameter and update it to the global
        //recalculate BlockSize and GridSize from the new parameter
}
```

Figure 4.9: Recalculate the parameters inside the kernel function

In Figure 4.9, the three displacement functions are called from their parent function, and they are all initialized with a block size and a grid size at the beginning, which depend on the number of particles in each box. When the Particle Transfer move is accepted, the global parameter of the number of particles in each box will be changed,

and this will also causes the changes on both the block size and the grid size. They will also be updated in the global memory before the next kernel functions are called. In this way, it is not necessary to add any global synchronizations after each of the functions, and there is no need to spend any time on circumventing the asynchronous errors either.

4.3 Performance Trade-Off

As discussed earlier, thread occupancy and the memory usage of each thread have noticeable effects on the performance. It is similar to the comparison between thin threads and fat threads [12]. Either of these extreme cases, like highest occupancy or redundant memory per thread, may potentially ruin the performance. It is also important that the number of threads allocated to a function should be an integral multiple of 32, which is the size of a warp.

With the GK110 architecture, the NVIDIA Kepler K20c card supports 65536 registers and 2048 threads in each multiprocessor. Therefore, when the occupancy is 100 percent, in other words, all 2048 threads are activated, there are 32 registers allocated to each thread. The limitation of this condition is from threads and registers because they are all fully loaded. Making the resources fully loaded is ideal to optimize concurrency, but it is not the only factor. Therefore, it is not strange that the performance of the Particle Transfer function in this condition is not the best. The result of comparing the use of 32 registers with other numbers of registers shows that a thread with 32 or even fewer registers is much less efficient, and it takes longer to finish one of the parallel tasks.

To find the ideal number of registers, first of all, it is recommended to compare all the possibilities with the same amount of threads. The compiler makes the comparison easier since it could adjust the number of registers for each thread to the most appropriate one even if the programmer gives a bad value. The bad value refers to the inappropriate number of registers set to each thread, from which the program could gain little improvement for each individual thread but lose a lot on occupancy. For exam-

ple, in the condition of 5 warps per block and 45 registers per thread, the compiler is able to invoke 9 blocks. However, if the number of registers is set to 46, the compiler will only be able to invoke 8 blocks. To give the compiler more opportunities to adjust the appropriate number of registers, it is advised to use *__launch_bounds__(block size, maximum grid size)*. With this instruction, a programmer could set the total number of threads in each SMX, and the number of registers in each thread will be calculated by the compiler. Visual Studio provides another option: a maximum number can be set in the property of the project. However, it makes the compiler inflexible and forfeits the ability to adjust the bad parameter value. The second restriction on register allocation is from the shared memory, which makes the trade off considerations more complex. In the Kepler GK 110 architecture, the maximum use of the shared memory is 48K and can be achieved with the instruction, *cudaFuncCachePreferShared*. In my research, a 256 bytes memory is set to each kernel primitively, and each thread needs 32 bytes for the execution. Therefore, the shared memory limits the number of threads in each block and the number of blocks in each SMX. Consequently, when the quantitative restriction on the number of threads is ignored because full use of the threads results in low efficiency, the performance is limited mainly by the use of registers and shared memory, and the best performance can be gained when both are optimized at the same time so that all the registers and all the shared memory are both used. Optimizing either register or shared memory separately will result in the inefficient use of the other one. Theoretically, the use of registers per thread and the number of threads per block can be restricted by the following two formulas.

$$32 \times n \times \sigma \times \epsilon \leqslant 65536 \tag{4.1}$$

$$256 \times \sigma + 32 \times 32 \times n \times \sigma \leqslant 49152 \tag{4.2}$$

where n equals the number of warps in each block, σ is the number of blocks per SMX,

and ϵ is the number of registers in each thread. The first 32 in the second formula is the usage of the shared memory by each thread, and both the second 32 and the 32 in the first formula are the size of a warp.

From these two formulas, it is possible to treat different sizes of warps separately and find out the most appropriate use of registers respectively. The theoretical result is shown in Table 4.2.

Table 4.2: The value of σ and ϵ have been rounded down to the nearest integer.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
σ	2	3	3	3	3	4	4	5	5	6	7	9	11	14	21	38
ϵ	64	45	48	52	56	46	51	45	51	48	48	45	46	48	48	53

The practical performance may not completely follow the theoretical value because of some unknown preliminary occupancy on registers. Sometimes, the compiler cannot allocate the exact number of registers, listed in Table 4.2, to each thread. It may slightly reduce the value to find the most appropriate setting. However, the practical result is still the most efficient as will be discussed in the next section.

CHAPTER 5

EXPERIMENTAL RESULTS

Since all the changes are made on the existing CUDA code, the results are stated in terms of the speed-up which compares the Kepler architecture implementation to the existing one.

Due to the lack of performance analysis tools on Linux, the experiment is implemented on Windows 7. A free tool called Nsight, which is provided by NVIDIA, can be embedded to Microsoft Visual Studio 2010. With the ability of nested analysis, Nsight simplifies the performance diagnostics, since all the performance restrictions of each kernel function, including the spawned functions, are visible to programmers. Hence, programmers can tradeoff the performance accordingly.

5.1 The Performance of Original CUDA Code

To make the comparison more precise and objective, it is necessary to trade off the performance of the original CUDA code to reach the best. Because it is mainly restricted by the usage of registers, the best performance can be easily identified by setting different block sizes. As illustrated in Table 5.1, the experiments are made on a system with the size of 32768 particles. A system size of 32768 particles is big enough to utilize all the resources of the K20c, and it is not so big that the execution time is unreasonable for running many different configurations. Therefore, running the experiments with 32768 particles is time saving and accurate. The best performance of the original CUDA code with the size of 32768 particles is achieved when the maximum block size is 128.

Table 5.1: The performance comparison of the original CUDA code.

Block Size (particles)	512	480	448	416	384	352	320	288
CBMC with size of 32768 particles (seconds)	299.413	299.959	303.224	345.404	295.716	297.265	318.741	294.808
Block Size (particles)	256	224	192	160	128	96	64	32
CBMC with size of 32768 particles (seconds)	291.349	294.949	291.199	288.64	286.994	288.246	361.942	602.059

5.2 The performance of Warp Shuffle Function

As it is described above, Warp Shuffle, as a new method to transfer data, is a widely applicable operation. As a result, the Warp Shuffle functions can also be added to the original code and contribute to the enhancing the performance. The performance of the Warp Shuffle function is compared to the performance of the original CUDA code in Table 5.2. The comparison between the original CBMC code and the Warp Shuffle code allows one conclusion: the Warp Shuffle function can contribute performance to all systems, and the larger the system is, the more significantly the speedup that can be achieved.

On the other hand, the self performance trade-off on the Warp Shuffle code also mainly focuses on the use of registers because it uses limited shared memory and the code structure is almost the same as the CBMC code. Therefore, the best performance (listed in Table 5.3) of the Warp Shuffle code is also achieved with the block size of 128. In Table 5.3, the performances of the Warp Shuffle code with block size of 512 is much slower for the larger system sizes because it allows fewer registers per thread, which reduces the performance of each thread.

Table 5.2: Comparison between the best performance of CBMC code and Warp Shuffle code.

System Size (particles)	CBMC (seconds)	Warp Shuffle (seconds)
1024	57.181	54.946
2048	61.739	59.153
4096	72.244	67.569
8192	87.85	80.844
16384	141.06	125.836
32768	286.944	248.891
65536	808.161	665.288
131072	2868.72	2251.97
262144	10613.8	8033.6

5.3 The Performance of Dynamic Parallelism

Same as the Warp Shuffle code, several experiments are made to determine the best performance of the Dynamic Parallelism code. It is different from the CBMC code and Warp Shuffle code because the change made to eliminate the global synchronizations also makes the Particle Transfer function require more shared memory than the Particle Displacement function and Volume Swap function. Hence, the shared memory is another restriction to the Particle Transfer function in the Dynamic Parallelism code. Since there are no other changes, a block size of 128 is still the best size for the other functions. In addition, shown by the comparison in Table 5.4, a block size of 160 is the best for the Particle Transfer function.

The Dynamic Parallelism code is also more competitive at the performances which is shown in Table 5.5. Comparing the Dynamic Parallelism code to the original code, there is a decent speedup.

Since the performance improvement of eliminating data transfers is limited and con-

Table 5.3: The performance of the Warp Shuffle code with two block sizes.

System Size (particles)	block size = 128 (seconds)	block size = 512 (seconds)
1024	54.946	55.12
2048	59.153	59.85
4096	67.569	65.59
8192	80.844	86.68
16384	125.836	152.89
32768	248.891	280.312
65536	665.288	959.17
131072	2251.97	3202.03
262144	8033.6	11039.4

Table 5.4: The performance comparison of the Dynamic Parallelism code.

Block Size (particles)	512	480	448	416	384	352	320	288
Dynamic Parallelism with system size of 32768 particles (seconds)	255.884	250.672	250.242	255.125	255.457	250.422	251.81	250.372
Block Size (particles)	256	224	192	160	128	96	64	32
Dynamic Parallelism with system size of 32768 particles (seconds)	251.897	250.212	250.528	250.201	250.773	250.913	255.196	264.358

Table 5.5: The best performance of the CBMC code and the Dynamic Parallelism code.

System Size (particles)	CBMC (seconds)	Dynamic Parallelism (seconds)
1024	57.181	42.444
2048	61.739	47.349
4096	72.244	58.576
8192	87.85	73.731
16384	141.06	121.094
32768	286.944	251.588
65536	808.161	670.104
131072	2868.72	2263.95
262144	10613.8	7999.83

stant, the elimination of global synchronizations is the key factor that can make a big difference in the performance. Figure 5.1 displays the comparison between the Dynamic Parallelism code before eliminating global synchronizations and the code after the change.

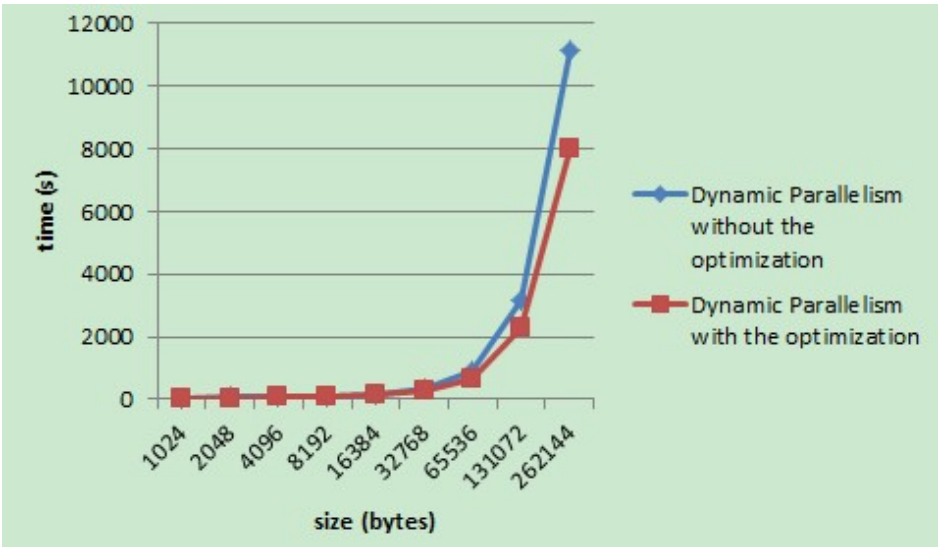

Figure 5.1: The Dynamic Parallelism code after the elimination of the global synchronization achieves decent performance improvements.

5.4 The Performance of the Combined Code

The comparison of the Warp Shuffle code and the Dynamic Parallelism code is shown in Figure 5.2. The line chart shows that the performances of these two functions are pretty close to each other.

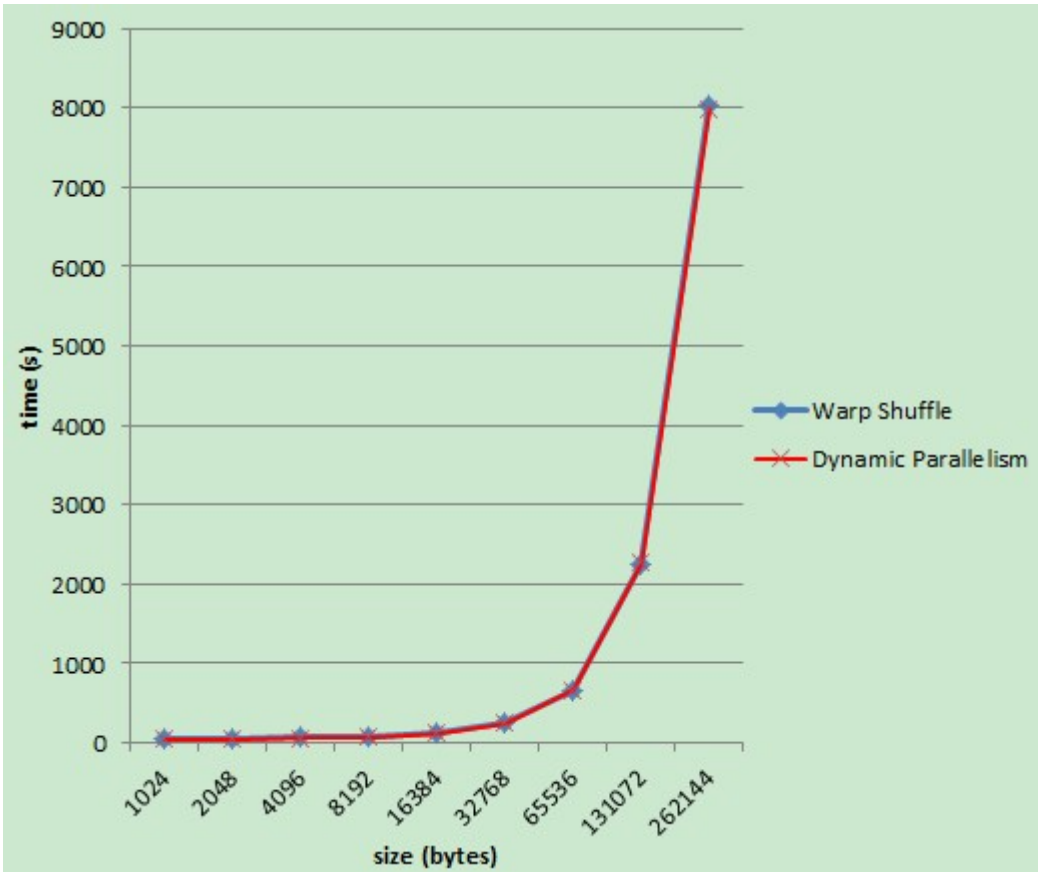

Figure 5.2: The two lines overlap each other.

The performance of the combined code, which includes both Warp Shuffle and Dynamic Parallelism, is shown in Table 5.6 and achieves a slight improvement over either of these two respective codes.

The execution time of the combined code gives a good but not a surprisingly good result. It does not save twice what Warp Shuffle and Dynamic Parallelism separately do as what is expected, but instead performs only modestly better on the optimization. This performance degradation may be caused by the nonideal block size of the Particle Transfer

Table 5.6: The comparison between the performances of the combined code and the CBMC code.

System Size (particles)	CBMC (seconds)	Warp Shuffle (seconds)	Dynamic Parallelism (seconds)	Combined Code (seconds)
1024	57.181	54.946	42.444	38.837
2048	61.739	59.153	47.349	43.638
4096	72.244	67.569	58.576	54.543
8192	87.85	80.844	73.731	69.504
16384	141.06	125.836	121.094	115.927
32768	286.944	248.891	251.588	248.872
65536	808.161	665.288	670.104	652.693
131072	2868.72	2251.97	2263.95	2206.48
262144	10613.8	8033.6	7999.83	7793.19

function. All three kernel functions can achieve improvements from Warp Shuffle, so to get the best performance from the Warp Shuffle, the block size of the Particle Transfer function should be 128. However, to make full use of Dynamic Parallelism, the block size has to be increased to 160, which could reduce the performance. There may also be some other unknown restrictions or conflicts happening during the execution, which requires further research.

Assume that the running times of the CBMC code are the basis numbers, which are set to 1. After comparing the optimized codes to the CBMC code, the ratio of the execution times for these optimized codes are shown in Figure 5.3. The histograms show a range of improvements from approximate 17% to 33%. For large systems, which are over 64K in size, the combined code achieves an average improvement of 20%.

The ratios in Figure 5.3, reflect the efficiency of the optimized code. It is interesting that the accelerated codes get good results from the smallest system; however, the performance keeps decreasing until the system size is increased to 32768 particles; after the

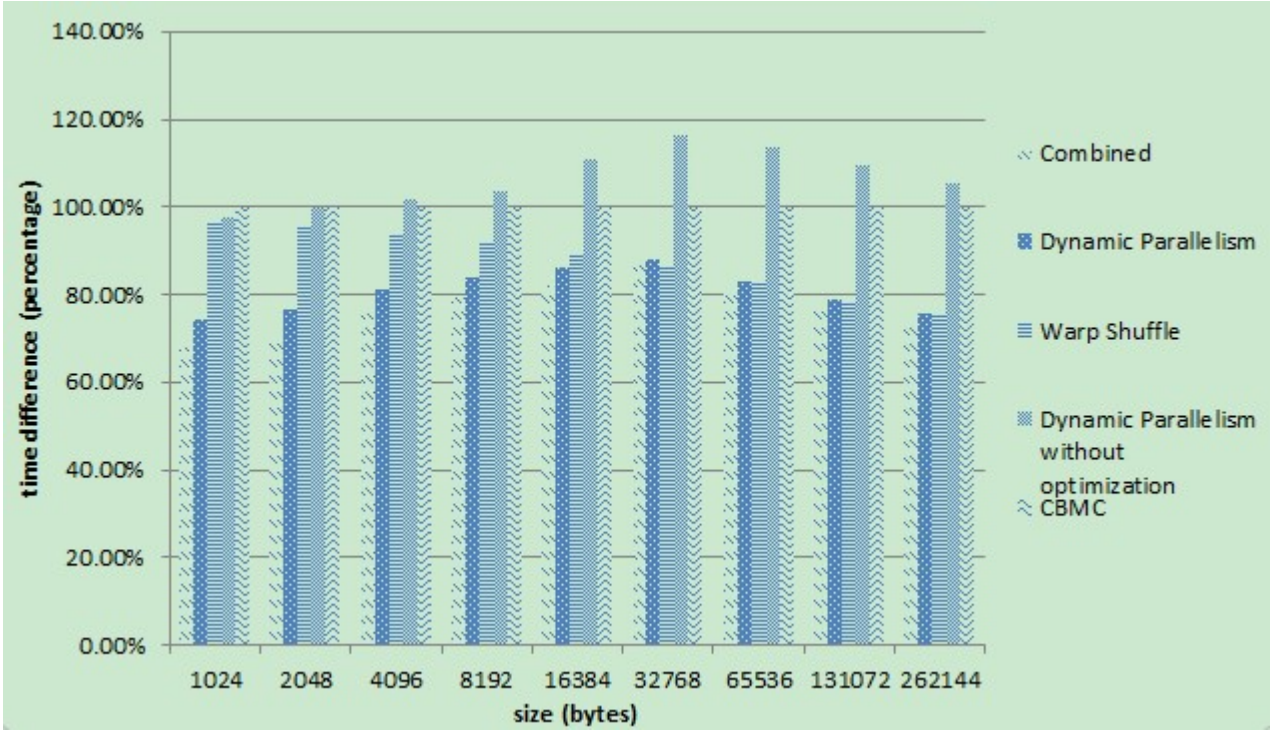

Figure 5.3: Comparison of the execution time of four different codes.

valley, the performance improves and remains stable. In addition, Dynamic Parallelism is more efficient than Warp Shuffle in the systems where the sizes are smaller than 32768 particles. However, they achieve very similar performance when the system sizes are larger than that.

CHAPTER 6

CONCLUSION AND FUTURE WORK

I utilized some of the new features of Kepler GK110 architecture that can contribute to the code optimization. The result I got about Warp Shuffle functions proves the power of using registers; it conforms to the memory hierarchy of the GPU [6]. However, NVIDIA positions the feature of Dynamic Parallelism unilaterally. It doesn't claim the benefits that Dynamic Parallelism could make to code optimization. The Dynamic Parallelism cannot speed up the code directly, but it can still make contributions to simplifying and optimizing the construction. Except for the main features that are mentioned above, I also utilized some other techniques that can make some small improvements as well. Loop unrolling on the simple loops with no divergence inside can release the compiler from unrolling the loops and improve the performance. Expanding the workload that is assigned to each of the threads in the function should also be considered. The workload extension gives the light threads a reasonable amount of extra work to make the thread calling worthwhile.

In this thesis, I evaluated some extreme configurations to trade off the performance. It is kind of inefficient and not that accurate. To achieve better performance, this job could be done by using auto-tuning [13]. Programmers can simply design a template that can change the value of all the parameters automatically to find the best configuration. From the final result, it still does not seem to achieve the best performance. The time that the combined code saves does not equal the total time that is saved by both Warp Shuffle and Dynamic Parallelism alone. Intuitively, it seems likely that further research will achieve additional speedups.

BIBLIOGRAPHY

[1] K. Binder, editor. *Monte Carlo and Molecular Dynamics Simulations in Polymer Science*. Oxford University Press, 1995.

[2] A. Brodtkorb, T. Hagen, and M. Stra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4 – 13, 2013.

[3] Nvidia Corporation. NVIDIA Fermi Compute Architecture Whitepaper, 2009.

[4] Nvidia Corporation. Nvidia Tesla Kepler GPU Computing Accelerators, 2012.

[5] Nvidia Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, 2012.

[6] Nvidia Corporation. CUDA C Programming Guide, 2013.

[7] Nvidia Corporation. Kepler Tuning Guide, 2013.

[8] E. Hailat. *Advanced optimization techniques for Monte Carlo simulation on graphics processing units*. PhD thesis, Wayne State University, 2013.

[9] E. Hailat, K. Rushaidat, L. Schwiebert, J. Mick, and J. Potoff. GPU-based Monte Carlo Simulation for the Gibbs Ensemble. In *Proceedings of the High Performance Computing Symposium*, HPC '13, pages 10:1–10:8, San Diego, CA, USA, 2013. Society for Computer Simulation International.

[10] E. Hailat, V. Russo, K. Rushaidat, J. Mick, L. Schwiebert, and J. Potoff. Parallel Monte Carlo simulation in the canonical ensemble on the graphics processing unit. *International Journal of Parallel, Emergent and Distributed Systems*, pages 1–22, 2013.

[11] J. E. Jones. On the Determination of Molecular Fields. II. From the Equation of State of a Gas. *Royal Society of London Proceedings Series A*, 106:463–477, October 1924.

[12] G. Klingbeil, R. Erban, M. Giles, and P. Maini. Fat versus Thin Threading Approach on GPUs: Application to Stochastic Simulation of Chemical Reactions. *Parallel and Distributed Systems, IEEE Transactions on*, 23(2):280–287, 2012.

[13] Y. Li, J. Dongarra, and S. Tomov. A Note on Auto-tuning GEMM for GPUs. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[15] J. Mick, E. Hailat, V. Russo, K. Rushaidat, L. Schwiebert, and J. Potoff. GPU-accelerated Gibbs ensemble Monte Carlo simulations of Lennard-Jonesium. *Computer Physics Communications*, 184(12):2662 – 2669, 2013.

[16] A. Panagiotopoulos. Direct determination of phase coexistence properties of fluids by Monte Carlo simulation in a new ensemble. *Molecular Physics*, 61(4):813–826, 1987.

[17] J. Phillips, R. Braun, W. Wang, et al. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[18] Wikipedia. Lennard-Jones potential, 2013.

ABSTRACT

GPU Optimizing and Accelerating Of Gibbs Ensemble On the CUDA Kepler Architecture

by

Yuanzhe Li

Dec 2014

Advisor: Dr. Loren Schwiebert

Major: Computer Science

Degree: Master of Science

The main purpose of implementing the code on Kepler architecture is to speed up the GPU code, which is from the previous work done by our group, by using the new functions of NVIDIA CUDA's Kepler architecture. Therefore, this thesis specifically focuses on the latest architecture.

To get benefits from the Kepler architecture, the primary work is to convert the code and make it adapt to the new features: Warp Shuffle and Dynamic Parallelism. The new code changes the way to transfer data and generate new kernel functions. In addition, another challenge is to trade off the use of resources on each thread to get the best performance.

The new code has different performance with different work sizes. Generally, the speedup is between 17% and 33%, and better performance is achieved in larger systems. This is a reasonable performance for the improvement with only two new features. The main contribution of this thesis is that the detailed evaluation of these two Kepler architectural features provide guidance to other researchers on the potential performance benefits of modifying their code. Therefore, they can make appropriate modifications and achieve reasonable speedup according to the structure of their codes.

AUTOBIOGRAPHICAL STATEMENT

Yuanzhe Li

5200 Anthony Wayne Dr

Detroit, MI 48201

(313)686-1060

Education:

Ph.D.	Computer Science
	Wayne State University, 2013-Present
	Advisor: Loren Schwiebert
M.S.	Computer Science
	Wayne State University, 2011-2013
	Advisor: Loren Schwiebert
B.S.	Computer Science
	XiDian University, China