Accelerating SRD Simulation on GPU

by

Zhilu Chen

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Electrical and Computer Engineering

by

April 2013

APPROVED:

Professor Xinming Huang, Major Thesis Advisor

Professor Erkan Tüzel

Professor Lifeng Lai

Abstract

Stochastic Rotation Dynamics (SRD) is a particle-based simulation method that can be used to model complex fluids either in two or three dimensions, which is very useful in biology and physics study. Although SRD is computationally efficient compared to other simulations, it still takes a long time to run the simulation when the size of the model is large, e.g. when using a large array of particles to simulate dense polymers. In some cases, the simulation could take months before getting the results. Thus, this research focuses on the acceleration of the SRD simulation by using GPU. GPU acceleration can reduce the simulation time by orders of magnitude. It is also cost-effective because a GPU costs significantly less than a computer cluster. Compute Unified Device Architecture (CUDA) programming makes it possible to parallelize the program to run on hundreds or thousands of thread processors on GPU. The program is divided into many concurrent threads. In addition, several kernel functions are used for data synchronization. The speedup of GPU acceleration is varied for different parameters of the simulation program, such as size of the model, density of the particles, formation of polymers, and above all the complexity of the algorithm itself. Compared to the CPU version, it is about 10 times speedup for the particle simulation and up to 50 times speedup for polymers. Further performance improvement can be achieved by using multiple GPUs and code optimization.

Acknowledgements

I would like to express my gratitude to my advisor, Professor Xinming Huang. He gave me the opportunity to do the research and guided me in my research.

Thanks to Professor Erkan Tüzel and James Kingsley for sharing the ideas of SRD simulation with me. This thesis would not exist without their help and insight of the physical model.

Thanks to all my friends and my family for giving me the courage and confidence to solve all the problems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Simulations are widely used in research and applications. In many cases, simulations are computationally expensive and thus require powerful computer clusters with expensive hardware. Even with these powerful computation resources, some simulations still take a long time like months to be completed. It is obvious that the optimized algorithm is vital to the simulations.

In this thesis we focus on modeling the particle-based fluid either in 2-dimension or 3-dimentsion, which is very useful in biology and physics area. In the real word, the particles in the fluid move randomly and affect each other based on Newton's law. Though it is possible to compute the interaction between every two particles and track the movement of each particle, we wouldn't want to do this because it requires too much computation and thus takes too long to run the simulations. SRD is a simulation method that can be used to handle such situations. It is computationally inexpensive compared to the traditional methods and therefore it saves time and money to run the simulation. Even though, it takes a long time to get the results in many cases depending on the simulation parameters. It would be great if we can accelerate the simulation.

Compute Unified Device Architecture (CUDA) programming makes it possible to parallelize the program to run on multiple cores on GPU. The program is divided into small parts and run in many threads independently and simultaneously. Thus it is possible to accelerate the simulation by parallelizing the computing tasks on a GPU instead of using expensive computer clusters.

Charpter 2 presents the idea of SRD method as well as polymers modeling. Charpter 3 introduces some basic concepts, procedures and techniques of CUDA programming using GPUs. Charpter 4 explains how we implement the SRD simulation on CUDA GPUs in details. Charpter 5 shows the evaluation of our code with different parameters and in different scenarios. Chapter 6 concludes our achievements and possible improvement in the work of future.

Chapter 2

SRD Method

2.1 SRD procedure

The SRD simulation method was originally proposed by Malevanets and Kapral in 1999 [1] and was further developed by several groups [2, 3, 4]. In this method, the fluid is divided into several square boxes with length l. With x boxes in one dimension, there are x^2 boxes in total, which is the grid size. There are n particles per box and thus there are nx^2 particles in the grid. Each particle has mass m, velocity $\vec{v}$ and coordinates (x, y) in the grid. We can calculate which box a particle belongs to using its coordinates. Fig. 2.1 is an example that illustrates how the grid is divided and how the particles are distributed randomly in the grid with random velocities.

For every time step, the particles within the boxes move and collide with each other. This method is very efficient compared to traditional methods. There is no need to track every particle to see if it collides with another one. Instead, we only compute the momenta of the particles within every box. The coordinates of the

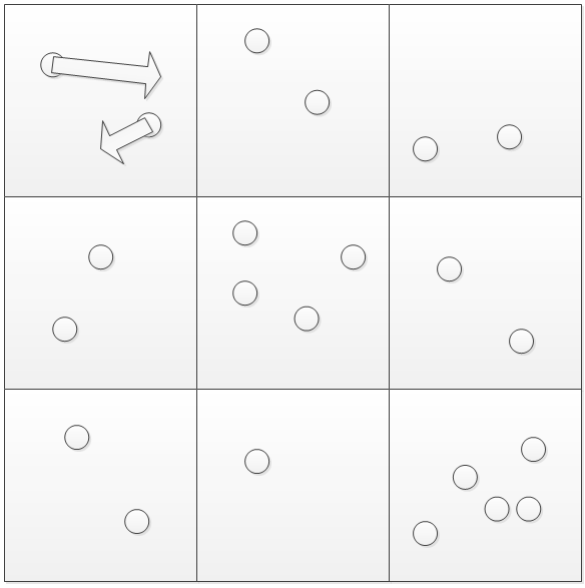

Figure 2.1: Particles in the grid with random velocities

particles are updated by

$$x(t + \triangle t) = x(t) + v_x \triangle t \tag{2.1}$$

and

$$y(t + \triangle t) = y(t) + v_y \triangle t \tag{2.2}$$

where v_x and v_y is the projection of $\vec{v}$ on x-axis and y-axis, respectively. For the collision step, we calculate the total momenta of every box and count how many particles are there in each box. The mean momentum will be

$$\vec{U} = \vec{I}/(nm) \tag{2.3}$$

where n is the number of particles in the box, m is the mass of a particle, $\vec{I}$ is the total momenta Then a rotation matrix and a rotation direction for each box are

generated for each box. The rotation matrix is

$$R = \begin{pmatrix} W_{xx} & W_{xy} \\ W_{yx} & W_{yy} \end{pmatrix} = \begin{pmatrix} \cos\alpha & \pm\sin\alpha \\ \mp\sin\alpha & \cos\alpha \end{pmatrix} \tag{2.4}$$

where α is the rotation angle. Thus the velocities of the particles can be updated using the equations

$$v' = v - \vec{U}/m \tag{2.5}$$

$$\triangle v = \begin{pmatrix} \triangle v_x \\ \triangle v_y \end{pmatrix} = \begin{pmatrix} W_{xx} & W_{xy} \\ W_{yx} & W_{yy} \end{pmatrix} \times \begin{pmatrix} v'_x \\ v'_y \end{pmatrix} \tag{2.6}$$

and

$$v_{new} = \vec{U}/m + \triangle v \tag{2.7}$$

where v'_x and v'_y are the x and y components of v'. The momenta and energy conserve in every box. The total energy in each box is

$$E = \frac{1}{2} m \sum v_i^2 \tag{2.8}$$

where v_i is the velocity of the ith particle in the box. The energy after collision is

$$E_{new} = \frac{1}{2} m \sum [\vec{U}/m + R(v_i - \vec{U}/m)]^2 \tag{2.9}$$

and we have $E = E_{new}$. In the simulation we also need to shift the grid at every time step. We do this by shifting the coordinates of all the particles so that the boundary conditions stay the same. In the shifting step as well as the moving step, it is possible that the particles are shifted or move out of the grid. We let the particles that are out of the grid enter from the opposite side using 2D wrap around.

2.2 Polymers

In the real word, polymers in the fluid cannot be modeled as particles. We model a polymer as an array of particles connected together. These points are nodes of the polymers and can be considered as particles mentioned before. Such particles have forces between each other so that they will not go too near nor too far away, like a set of springs. Thus the polymer stays in a whole. We also need to compute the bending force to adjust the velocities of the node particles. Due to the stretching energy and bending energy, the total kinetic energy no longer conserves. However, it is stay in a small range as the simulation runs and it is stable. Fig. 2.2 gives an example of how we model a polymer in the grid.

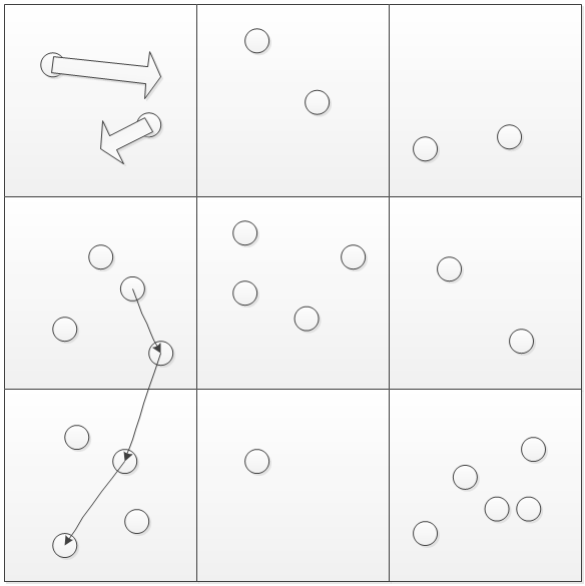

Figure 2.2: Modeling a polymer as an array of particles in the grid

The internal movements of the node particles caused by such forces are calculated using much smaller time steps. For example, for one of the iterations of the SRD, the movement of the particles of the polymer may be computed hundreds of times. The reason to do so is that the time step for SRD simulation is too large to simulate the movement of the node particles.

To make sure the polymers do not cross, we apply Lennard-Jones algorithm [5]. In addition to the forces applied to the node particles mentioned above, we compare every two node particles from different polymers. If they are close enough, we apply forces to both of them to make sure the polymers do not cross. Fig. 2.3 illustrates the forces between polymers.

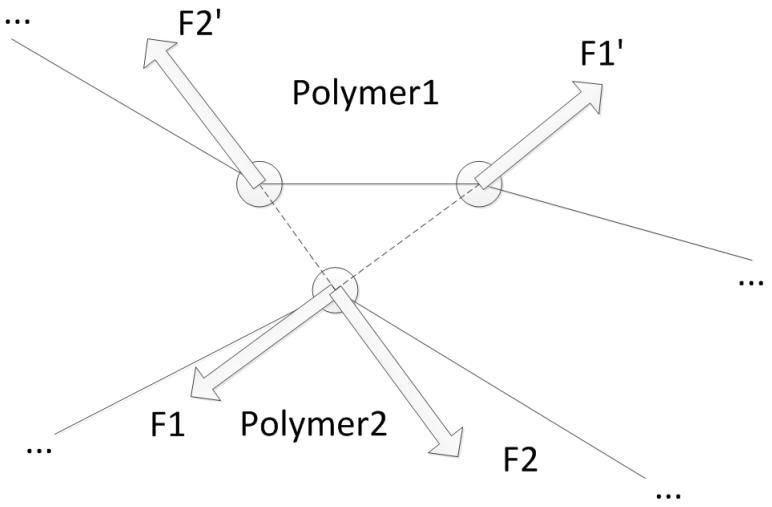

Figure 2.3: Example of forces between node particles from different polymers

If two node particles are close enough, the force can be expressed as:

$$f(r) = 24\epsilon(\frac{2\sigma^{12}}{r^{11}} - \frac{\sigma^6}{r^5}) \tag{2.10}$$

where r is the distance between two node particles and ϵ, σ are preseted physical parameters. The distance r is in unit of the box size a where $a = 1$. The force f is in unit of $\frac{ma}{\triangle t}$. If the value of r reaches a threshhold, then there are no forces applied to these node particles, which means $f(r) = 0$. Fig. 2.4 gives an example of the relationship between force f and distance r when $\epsilon = 50$ and $\sigma = 1$.

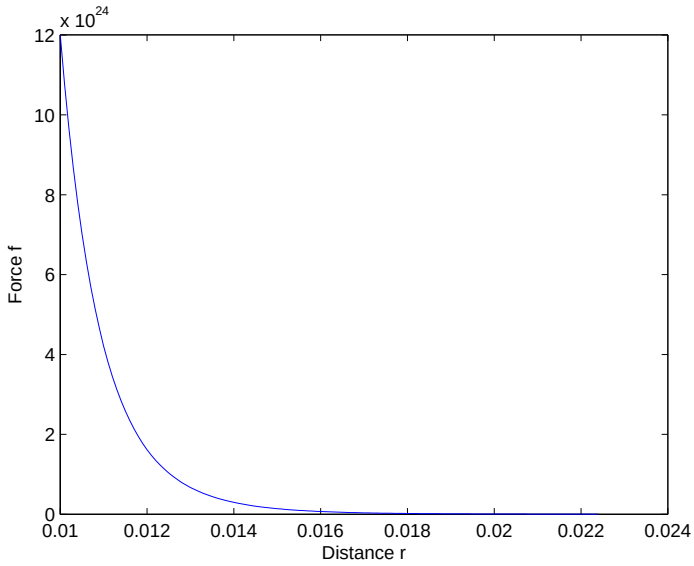

Figure 2.4: Example of relationship between force f and distance r when $\epsilon = 50$ and $\sigma = 1$

Chapter 3

CUDA

3.1 CUDA GPU overview

One of the most significant advancements in computer technologies of the past decade is General Purpose computing on GPUs (GPGPU). From a computer architecture prospective, GPUs have evolved into massively parallel, multithreaded, many-core processor system with tremendous computational power. Owing to the introduction of Compute Unified Device Architecture (CUDA) programming paradigm, a vast of computation problems outside of the graphics domain have benefited from the superior performance of GPUs. Among the examples of the GPGPU computing initiative are FFT, data mining, molecular dynamic simulation and many other science and engineering applications. Parallelism is accomplished through the use of threads on the GPU. CUDA has the ability to run the same kernel concurrently through many threads on a large number of cores. Picture a latest (as of March, 2013) GPU card, Tesla K20, which has 2496 cores. With the Compute Capability 3.5, each core can provide up to 1,024 threads per block and the maximum number of blocks is $2^{31} - 1$. Therefore, there is a possibility to run upwards of 2

trillion instances of a single kernel per GPU in a system. Spanning this number across multiple GPUs, it is easy to see how parallelism prevails and allows for faster processing. This type of parallelism is known as single instruction multiple data (SIMD). CUDA devices are also capable of running thousands of copies of small programs simultaneously as well.

3.2 Global memory and shared memory

Global memory is the memory used by GPU and it is also called device memory. The memory used by CPU is called host memory. Data can be copied between device memory and host memory. Thus we can make CPU and GPU work together to solve problems. For example, we do initialization by CPU and in host memory. Then the data is copied from host memory to global memory and the GPU does the computation in parallel. After the computation is done by GPU, we copy the updated data from global memory back to host memory. The CPU can then do the rest work such as displaying results. However, copying data between CPU and GPU is often expensive compared to the computation time. We must avoid copying data between them frequently in order to reduce the overall computing time. All threads in all blocks can access the global memory. However, the access time is hundreds of cycles which are too long compared to that of shared memory and registers. Thus, accessing data in global memory often becomes the bottleneck of a program, especially for the programs that are simple and don't have enough computation time to hide the delay of accessing global memory.

The advantage of the global memory is its abundant capacity. The size of global memory is often several GB, which is sufficient to store anything the program uses. But it is slow compared to the shared memory and registers.

Shared memory is the memory used by a block, but none out of that. It can be used by all threads within the block. Shared memory cannot communicate with host memory but can communicate with global memory. The shared memory is very small compared to global memory, but it is very fast: it only takes several cycles to access the data in shared memory. Shared memories in different stream multiprocessors cannot communicate directly. They can only communicate with each other through global memory, i.e., copying data between global memories. To maximize the performance, we store the data shared by the threads in the same block in shared memory to avoid accessing global memory. Fig. 3.1 illustrates the usage and communications of host memory, device memory and shared memory.

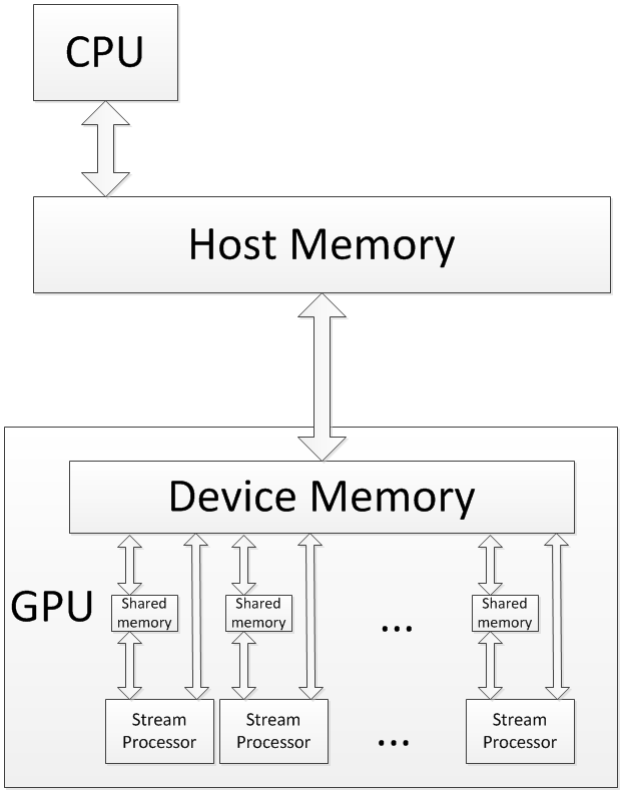

Figure 3.1: Memory usage and communications between CPU and GPU

3.3 Kernel functions

The code that runs on GPU is launched by kernel functions. The kernel function decides how the code is run in parallel. It decides how many blocks are there and how many threads per block. The code in each thread is the same, but uses different indexes to deal with different data. The local variables for each thread are stored in the thread memory which has a small size but high speed.

The blocks are distributed over the stream processors. Each stream multiprocessor processes a block at the same time and the threads in a block are run in parallel. The maximal number of active blocks and threads allowed in a stream multiprocessor are varying among different models of GPUs. Usually the more powerful the GPU is, the more threads and blocks can run in parallel. Fig. 3.2 shows the blocks and threads in a kernel function.

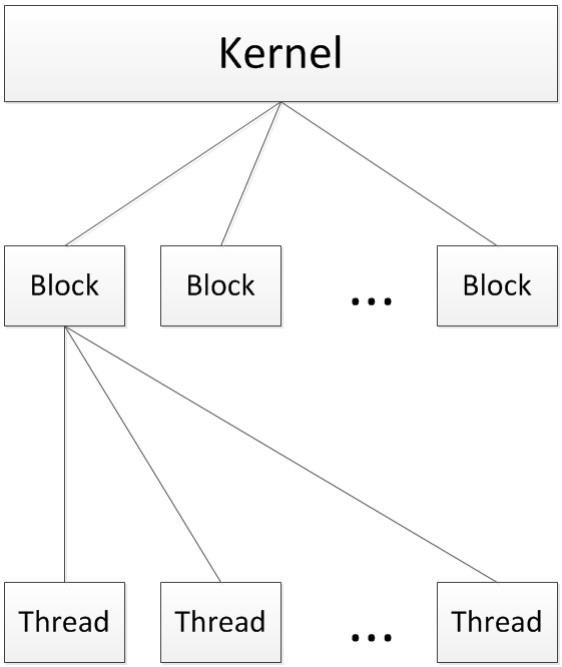

Figure 3.2: Blocks and threads in a kernel function

We must note that there is no automatic synchronization between host and

device when launch a kernel function on GPU. That means, the host launches a kernel function and continues to execute the next line of the code. It does not wait for the completion of the kernel function. If we need to use the results of that kernel function, we can use cudaThreadSynchronize( ) function to synchronize between host and device. If we use cudaMemcpy( ) function in the host program, it also synchronizes between host and device. In this case, there is no need to use cudaThreadSynchronize( ) function.

3.4 Avoiding hazards

Since the code runs in parallel, it may has problems if we don't deal with the data carefully. For example, if different threads want to update the same variable in global memory, problems may occur because we cannot ensure the order of the execution of the threads. Suppose we have two threads calculation a simply expression $n = n+1$ and update the value in global memory. It is easy to see that the final result is $n+2$. However, if one thread read the value n before the other thread finish updating the value, both threads will write the same value $n + 1$ to global memory, which is incorrect. Fig. 3.3 shows such situation with incorrect result.

To solve this problem, we use atomic operations of CUDA. The atomic operations ensure that the execution of one thread always start after another atomic operation is finished. In the case that we have two threads calculating $n = n + 1$ by atomic operations, the first thread reads the value of n and the other thread waits there until the first thread completes its calculation and update the value to $n + 1$. Then the second thread reads the updated value which is $n + 1$, and compute the final result $n + 2$, which is correct. However, it is obvious that using atomic operations affects the performance of the program because there are threads doing nothing but waiting

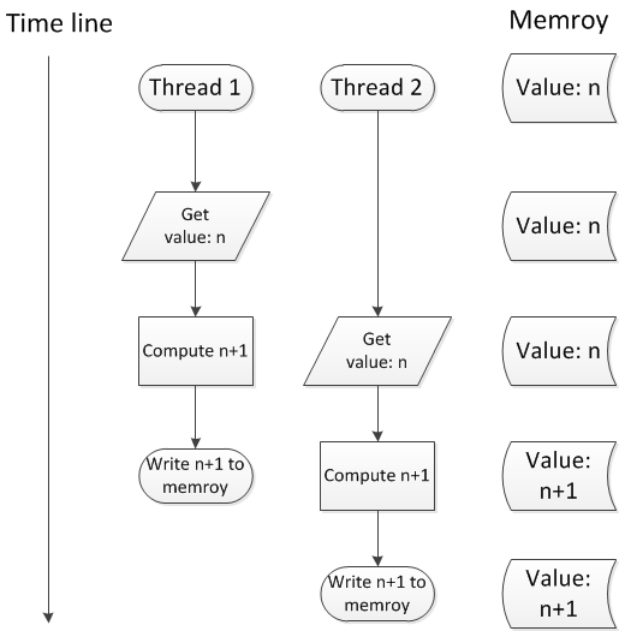

Figure 3.3: Multi-thread causes incorrect result

there and that is a waste of resources. Therefore we only use atomic operations when it is necessary. Atomic operations are only supported by the GPUs with computation capability of 1.2 or higher. In addition, the atomic operations only support floating-point precision but often our code needs double precision operations. We can use the function in [6] to build the double precision atomic-add function. Fig. 3.4 shows how the atomic operations get the correct result.

```
__device__ double atomicAdd(double* address, double val)
{
        unsigned long long int* address_as_ull = (unsigned long long
            int*)address;
        unsigned long long int old = *address_as_ull, assumed;
        do
        {
                assumed = old;
                old = atomicCAS(address_as_ull, assumed,
                    __double_as_longlong(val + __longlong_as_double(
```

```
                    assumed)));
        }
        while (assumed != old);
        return __longlong_as_double(old);
}
```

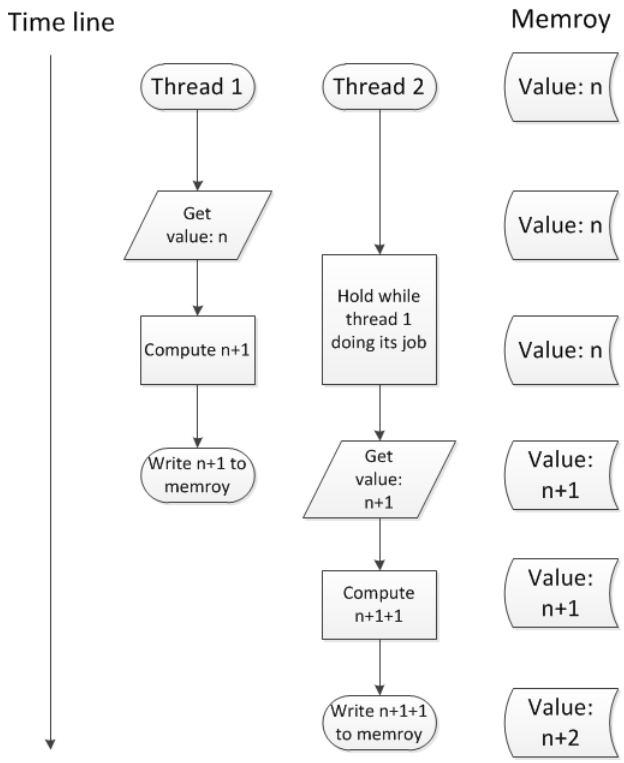

Figure 3.4: Using atomic operations to get correct result

Another type of hazard is due to the dependence of the data or program. For example, a block of threads run in parallel to compute a for-loop and the results require all the threads complete their tasks. Once it's done, all the threads use the results of that for loop to compute another loop. If we don't do anything, some threads could enter the second loop while other threads are still working on the first loop, which means the results of first loop have not been available. To ensure all the threads have complete their tasks before entering the second loop, we must synchronize all the threads before the second loop. Thus the threads that have

completed the computation will wait until all threads have finished the first loop and then all the threads enters the second loop together, with the results of the first loop ready to use. It is similar with the atomic operation, synchronization affect the performance as well. We would like to reduce the synchronization if possible. Moreover, threads can only be synchronized within the same block. There is no way to synchronize the threads in different blocks, and there is no way to synchronize multiple blocks. If synchronization among blocks is required, we have to terminate the kernel function and then launch a new one, which affects the performance.

Chapter 4

GPU Approach

4.1 Initializing the simulation

We use CPU host function for initialization, because the initial function is executed only once and its execution time is much shorter compared to the actual simulation time. The variables such as velocities that are vectors are in Euclidean space with x axis and y axis for two dimensional simulations and the additional z axis for three dimensional simulations. For simplicity purpose, we mainly discuss two dimensional cases here and it is easy to extend the approach to three dimensional simulations.

The field is divided into boxes. The box structure has the following properties: momenta along x axis direction, momenta along y axis direction, number of particles inside the box and the random rotation variables along x axis direction and y axis direction. Each box has multiple particles. The particle structure has the following properties: coordinates in the field, velocities along x axis direction and y axis direction, mass of the particle and variables that record the number of particles moved out of the boundaries. We use an array to store the particles and another array to store the polymers. Although each polymer is an array of particles, we

still use a one-dimensional array to store all the particles for all polymers for the convenience of GPU implementation:

$$i = i_{poly} \times n + i_{part} \tag{4.1}$$

where i is the index of the particle in the one-dimensional array, n is the number of node particles in a polymer, i_{poly} is the index of the polymer and i_{part} is its index in that polymer.

Upon initialization, the arrays of boxes, particles and polymers are stored in the host memory. Then we copy these arrays to global memory using cudaMemcpy( ) function. The data stays in the global memory until the simulation is done and it avoids copying between host memory and global memory which is time consuming.

4.2 SRD kernels

The simulation loop is outside the kernel because each time step, or iteration, depends on the results of previous iteration, such as velocities and coordinates. Thus the simulation loop are sequential, not in parallel.

The reason why we use 4 kernels instead of a single kernel is that we need to synchronize the code. A later step may require the results of the previous steps. For example, we must reset every box before we add the momenta to it. Such dependences must be satisfied, or the results would be incorrect. It has no problem on CPU because all the code is executed sequentially. However, we cannot guarantee that the code execution is in sequential order on the GPU because it is divided into many blocks and those blocks run in parallel. If one block completes resetting the box and move on to add momenta to a box that haven't been reset by another block, problems will occur. All momenta that are added to the box prior to the reset action

will be lost. In the other hand, CUDA only support threads synchronization within a block, it does not support synchronization among blocks. Since we have many blocks, we have to terminate a kernel, synchronize between host and device, then launch a new kernel. Only then can we make sure the dependences are satisfied, though it slows down the execution speed a little bit.

For each iteration of particle simulation, the following 4 kernel functions are launched. The first one is kernel_ResetBoxes( ). It runs as one box per thread. For each box, it resets the mean momentum to zero and counts the number of particles. The second one is kernel_SumMomenta( ). It runs as one particle per thread. For each particle, the thread function decides which box the particle belongs to, according to its position and the box-shift value. Then it adds the momenta to that box and the number of particles of that box is increased by 1 using double-precision atomic operation. The following one is kernel_DivideBoxes( ). It runs as one box per thread. For each box, it calculates the mean momentum of the ith box

$$\vec{u_i} = \vec{I_i}/c_i \tag{4.2}$$

where $\vec{I_i}$ is the total momenta and c_i is the counted number of particles of the ith box. It generates a random rotation direction r for the rotation matrix and its value is 1 or -1. Finally we have kernel_Collide_Move ( ) function. We combine the collision step and moving step into a single kernel function to save the synchronizing time and launching time. It is possible to do that for these two steps because one of the step is followed by another and they have no dependency. It also because that both of them loop by particles and that means they have the same number of threads if our kernel function runs as one particle per thread. For each particle, it decide which box it belongs to. Then it updates the velocities of the particles according

to the mean momentum and the rotation matrix of that box. The particles move at each time step. If they move across the boundary, we let them enter from the opposite side using 2D wrap around. Fig. 4.1 shows the procedure of our code to run SRD simulation with CUDA.

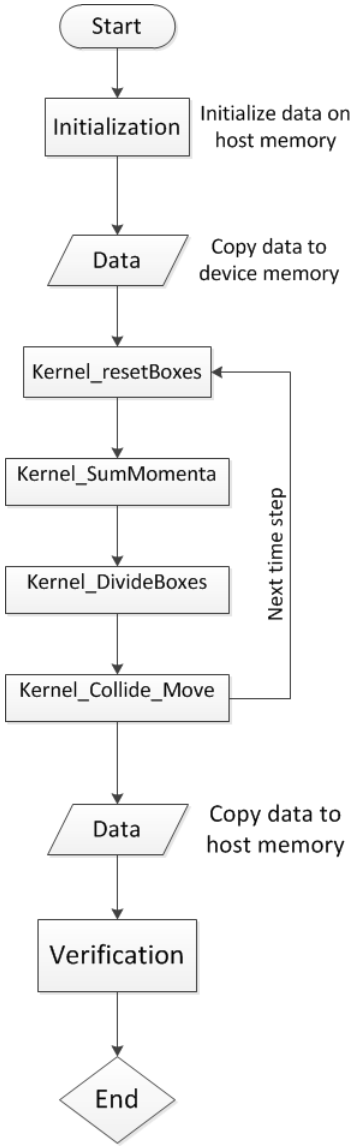

Figure 4.1: Flowchart of SRD

To compile our CUDA code on Linux, we use the following command:

```
nvcc fluidsim.cu -o fluidsim -arch compute_20 -code sm_20 -I /usr/local
    /cuda/include -L /usr/local/cuda/lib64
```

where compute_20 and sm_20 indicates that we are using computing capability 2.0, which support most of the important operations we are using in CUDA.

4.3 Polymer kernels

A polymer is considered an array of particles. Thus the polymers can use similar procedure of the particles. However, we need to make sure that they are linked together with several nodes. By calculating the forces between the particles in a polymer, we can adjust the velocities and thus the momenta and positions of them. Therefore, Kernel_Collide_Move ( ) is modified to Kernel_Collide( ) and Kernel_Move( ). Kernel_Collide( ) works for both particles and polymers. Kernel_Move( ) works only for particles. A new kernel function called Kernel_MolecularDynamics( ) is created for polymers to replace Kernel_Move( ). To speedup this kernel and achieve effective synchronization, we assign one block to deal with each polymer. So we can synchronize inside the block without terminating the kernel and that is more efficient. Shared memory is used for communication among threads within the same block. By using shared memories, the particles that are within the same polymer can communicate with each other. The node particles used are read from global memory and stored in shared memory for efficient execution.

As we mentioned before, the time step for calculating forces and simulate the movement of the node particles are much smaller than SRD time step. For every small time step, the forces of every two node particles are calculated using device function calcForce( ). Instead of writing the calcForce inside the kernel function, we use this device function to simplify the code because we need to call it several times within each time step. In order to calculate the forces, we first compute the distances between every two particles. Each thread has access to the node particles handled

by other threads because the node particles are stored in shared memory, and once we have the results we also store them in the shared memory so that all threads in the same block have access to them to compute forces. Then we use __syncthreads( ) to synchronize all the threads to make sure the all the distances have been calculated and ready to be used in shared memory. Each node has connections at both ends, except for the head node and tail node. Each thread computes the forces, including spring forces and bending forces, based on the distances between the node itself and connected nodes. The way of the spring forces calculated is similar to calculating the springe forces:

$$F = k \times (d - l) \tag{4.3}$$

where d is distance and l is the original length with zero forces. The bending forces are calculated using the distances between the two nodes on each side. Once the spring forces and bending forces are calculated, the device function returns the summary of the spring forces and the bending forces to the kernel function.

At the beginning of the small time step, we call the device function calcForce( ) to compute the forces to move the node particles slightly:

$$\triangle s = \tau v + \frac{1}{2}\frac{f}{m}\tau^2 \tag{4.4}$$

where m is the mass of the node particle and τ is the small time step. After synchronizing the threads in the same block with __syncthreads( ), we call the device function calcForce( ) again to calculate the updated forces for new positions of all the node particles. Once we have the updated forces, we also update the velocities of every node particle:

$$\triangle v = \frac{1}{2}\tau\frac{f_{new} + f_{old}}{m} \tag{4.5}$$

where m is the mass of the node particle and τ is the small time step. At last, update the polymers in global memory and that ends our kernel_molecularDynamics( ) function. Fig. 4.2 illustrates the polymer kernels in the simulation.

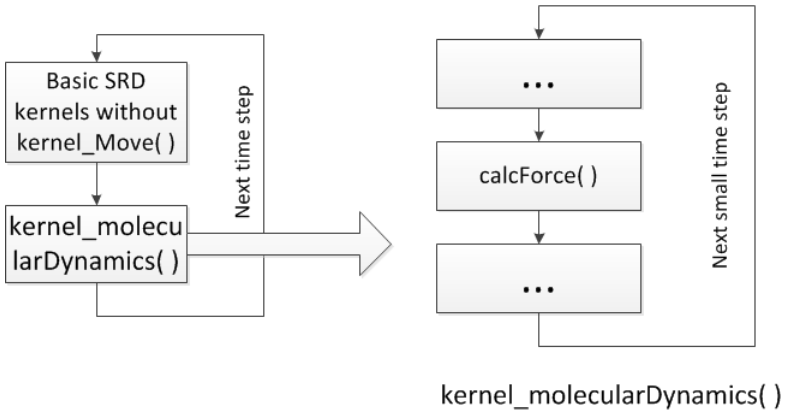

Figure 4.2: Flowchart of polymer simulation

4.4 Lennard-Jones algorithm

The kernel_molecularDynamics( ) function only concerns the interactions between the node particles in the same polymer. However, there are also interactions between different polymers. One problem we need to deal with is that the polymers should not cross because they are modeled as arrays of particles linked together but not independent particles. To make sure that the polymers do not cross, we need to check every two node particles in different polymers after calcForce( ) function, using the same small time step. Since we need the data of different polymers every small time step, the synchronization is required to avoid hazards. However, we use one block to handle one polymer in our kernel_MolecularDynamics( ) function, it is impossible to synchronize the blocks without terminating the kernel function. As a result, we convert the kernel_MolecularDynamics( ) into several kernel functions and use host loop to handle the small time step, and we use kernel_Lennard_Jones(

) function to make sure the polymers do not cross.

This function checks every two node particles from different polymers. In other to assign the computation task to each thread in each block, we need to decide which particle in which polymer is compared to another. We do not want to do redundant computation to compare the same pair of node particles more than once, we need to carefully decide which thread in which block to compare which pair of node particles. The following C++ code snippet shows general approach to determine the index of each particle in different polymers using loops, where POLY_COUNT is the number of node particles per polymer, n is the index of one node particles and m is the index of another.

```
for (n=0;n<POLY_COUNT; n++)
{
        for (m=n+1;m<POLY_COUNT;m++)
        {
                lennard_jones(m,n);
        }
}
```

To maximize the performance of our kernel function, we unroll the loops to one dimension as well as the thread id. The thread id is calculated like:

$$tid = blockIdx.x \times blockDim.x + threadIdx.x \tag{4.6}$$

where tid is the thread id, $blockIdx.x$ is the block id, $blockDim.x$ is the number of threads per block and $threadIdx.x$ is the index of the thread in the block. We name the index of the first polymer as $poly0$ and the other as $poly1$. The name of the index of the node particle in $poly0$ is $part0$, when $part1$ is the index of that in $poly1$. Then we compute them as the following code.

```
int poly_tid=tid/(POLY_PART_COUNT*POLY_PART_COUNT);
int part_tid=tid%(POLY_PART_COUNT*POLY_PART_COUNT);
int part0=part_tid/POLY_PART_COUNT;
int part1=part_tid%POLY_PART_COUNT;
int n=POLY_COUNT;
int poly0,poly1;
for(int i=n-1;i>0;i--)
{
        if(poly_tid<(n-1+i)*(n-i)/2)
        {
                poly0=n-i-1;
                poly1=poly0+poly_tid-(n+i)*poly0/2+1;
                break;
        }
}
```

Once the tasks are evenly distributed, each thread checks its pair of node particles to see if they are close enough to apply forces to them. If so, we calculate the forces and use double-precision atomic operations to update the forces because they may also be updated by other threads. Since kernel_Lennard_Jones( ) function is called at every small time step, we continue to run kernel_molecularDynamics( ) function for next small steps. Fig. 4.3 illustrates the polymer kernels with Lennard-Jones algorithm included in the simulation.

4.5 Finalizing the simulation

When the simulation is completed, we need to verify the results. Fisrt, we copy the data from device memory back to host memory using cudaMemcpy( ). Then we run a function to compute the total momenta and energy of all the particles as well

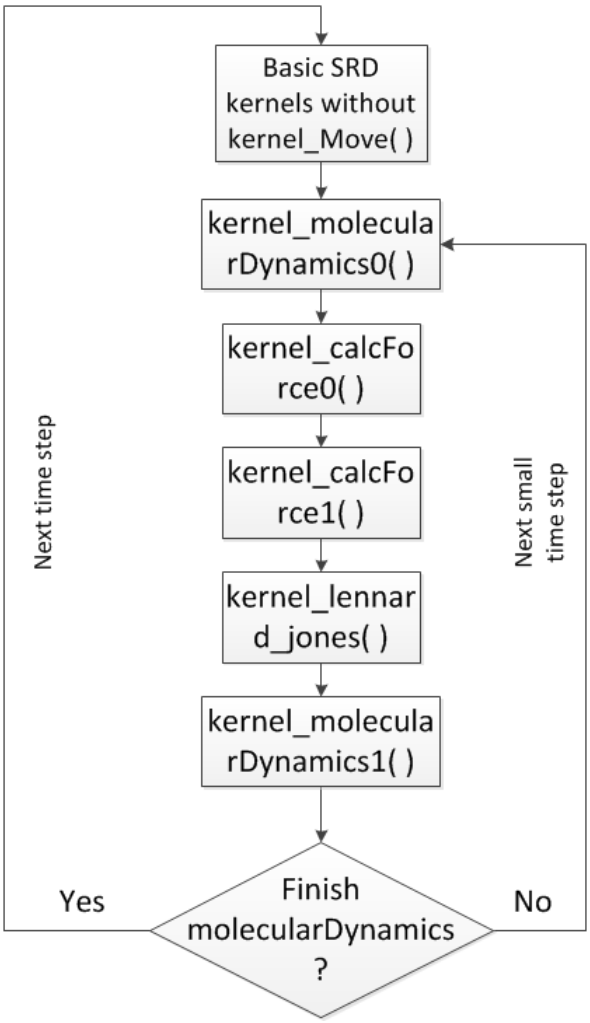

Figure 4.3: Flowchart of polymer simulation with Lennard-Jones algorithm included

as polymers to verify energy and momentum conservations. If there are polymers added, the energy may be a little different from the original energy because of the stretching energy in the polymers. Finally, we release the occupied memory on GPU using cudaFree( ) function.

4.6 Multi-GPU implementation

Our server contains 2 NVidia Tesla C2050 (1.15 GHz, 448 cores) GPUs with Compute Capability 2.0 and we have only used one of them so far. Then we are thinking

about using all of them to further accelerate our simulation. There are two types of approaches. One is to run two simulations on these two GPUs respectively but simultaneously. The other one is to run one simulation with two GPUs working together to share the burden.

We decide to do two simulations on these two GPUs respectively. There are three reasons. First, the simulation code has data dependency. In our single-GPU implementation, we are forced to split the kernels to do synchronization in order to satisfy these depending conditions. We terminate a kernel to make sure that all operations have been done in that kernel, and then launch a new kernel to do new operations. Such synchronization is inefficient but it has to be done for accuracy. In multiple-GPU implementation, synchronizing is even harder. Since we have multiple GPUs, we need to synchronize data among GPUs. This is inefficient just like the way we split the kernels. For example, a kernel function is done on GPU A but not done on GPU B, our code must wait there until it is done on GPU B, then both GPUs start to launch a new kernel function. Second, we have a vital problem that must be solved if we want to do that way: communication between GPUs. Each GPU has its own global memory and can only access its own global memory. How can the GPUs communicate with each other to know their status, and use the data or results computed by other GPUs? We can use OpenMP to keep track of the kernel functions on different GPUs and synchronize them [7]. For data communication, the most intuitive method is to copy the data from the global memories of all GPUs to host memory, and then copy the data that is useful back to global memories. Thus, the data from all the GPUs is exchanged. However, copying data between device memory and host memory is extremely slow. The speedup will be heavily degraded. Third, though we have so many threads and thus a lot of computational tasks to be parallelized to achieve good speedup, the computation task for each

thread is relatively small. That is to say, the computation is cheap for GPU and the advantage of our code is parallelism and our bottle neck is synchronization. Our code will probably even slower than the CPU version if we cannot hide the latency of communication between multiple GPUs. Thus this approach is not an option for us.

In the real world, we often need to run a simulation many times to obtain enough results for static analysis. Running multiple simulations on multiple GPUs respectively is a better choice. It is also much easier because there is no need to concern communications and synchronization between GPUs. What we do is to create multiple threads on CPU by OpenMP and each thread run a simulation on a GPU. In our case, we create two threads by using omp_set_num_threads( ) function in OpenMP. Each thread use one GPU by using omp_get_thread_num( ) in OpenMP and cudaSetDevice( ) in CUDA. Then each thread initializes the data on host memory and copy it to their device memory respectively.

To compile our CUDA code on Linux, we use the following command:

```
nvcc -Xcompiler -fopenmp -lgomp fluidsim.cu -o fluidsim -arch
    compute_20 -code sm_20 -I /usr/local/cuda/includes -L /usr/local/
    cuda/lib64
```

where "-Xcompiler -fopenmp -lgomp" is added because we are using OpenMP.

Chapter 5

Performance

5.1 Evaluation setup

During our experiment, we measure the computing performance on the server with two Intel Xeon X5650 CPUs (2.66GHZ, 12 cores, hyperthread) and two nVidia Tesla C2050 (1.15 GHz, 448 cores) GPUs with Compute Capability 2.0. Initializing and finalizing time is ignored in both CPU and GPU evaluations. We use clock( ) function without OpenMP and omp_get_wtime( ) with OpenMP. The omp_get_wtime( ) function is more accurate than clock( ) function, but both of them are capable of estimating the execution time.

5.2 SRD performance

The speedup of our CUDA code varies for different parameters in the simulation. The number of boxes in our 2D simulation and the number of particles in each box are significant. All computations use double-precision floating-point. One thousand iterations are performed. Fig. 5.1 shows the execution time of CPU over GPU, which is essentially the speedup factor of GPU over CPU, for the simulations with

different grid size and each grid initially contains 9 particles. So the largest case of 500-by-500 grid simulates the movement of a total of 2.25 million particles. Fig. 5.2 shows the same performance comparison for a fixed grid size of 128-by-128 while varying the number of particles in each grid. The largest case here is about 4.92 million particles in total. Since the GPU has a large, high-speed (GDDR5) memory, much larger simulation case can be executed on the GPU without much performance degradation. From both figures, it shows 10x speedup is readily achievable by using a single GPU when compared with two 6-core CPUs. Further speedup is achievable through using multiple GPUs and also by optimization of the simulation code.

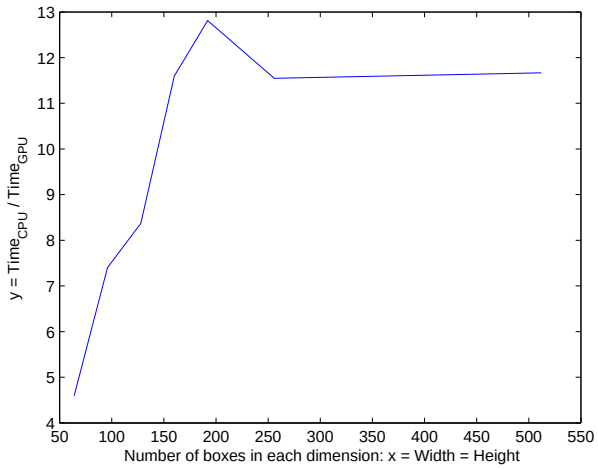

Figure 5.1: Speedup of GPU over CPU with different SRD grid size

We also tested our code in three dimensional cases. Table 5.1 shows the speedup in 3D grids. We can see that the performance is similar to that of 2D grids.

Table 5.1: Speedup of GPU over CPU in 3D grids

Particles per box \ Grid size	16^3	32^3	64^3
5	5.03	9.42	12.70
25	5.50	8.42	7.71

The total number of the particles increases rapidly with the grid size and thus

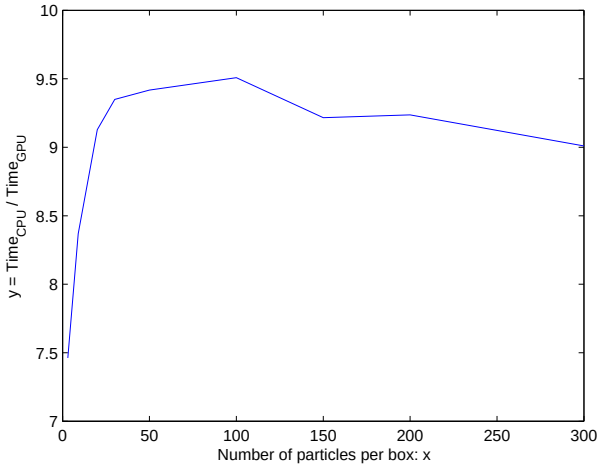

Figure 5.2: Speedup of GPU over CPU for SRD with different number of particles in each grid

we need to pay attention to the maximum number of blocks we can have in a kernel function. Since our Tesla C2050 has compute capability of 2.0, the maximum number of x-dimension of thread blocks is 65535, which is $2^{16}-1$. With 512 threads in one block, the maximum number of threads we can have is approximately 2^{25} in total. If the grid size is N^3 and we have x particles in one box, the total number of particles will be xN^3 and this number should be less than $2^9 \times (2^{16}-1)$. This is because the number of particles is much larger than that of boxes, and in some of the kernel functions we use one thread for each particle. In order to process larger grid with more particles in one box, we may let one thread to handle several particles with a loop inside the kernel. However, such loops themselves are not run in parallel, so there will be no contributions of these additional particles.

Some new GPUs have compute capability of 3.0 or higher, and the maximum number of x-dimension of thread blocks is $2^{32}-1$, which is incredibly large and thus we can simulate a larger grid with more particles per box.

5.3 Performance of polymer simulation

The speedup depends on several parameters in the simulation. Besides those we mentioned before, the number of polymers and number of particles per polymer are also important. We test it in 2D grid with different number of polymers added and with different number of node particles per polymer. The execution time is much longer on both CPU and GPU, but the speedup is much higher. Table 5.2 shows the speedup with 5 particles per box and grid size 128^2. Table 5.3 shows the speedup with 5 particles per box and grid size 256^2.We can see that the speedup increases with the number of polymers and number of node particles per polymer.

Table 5.2: Speedup of GPU over CPU with polymers added in grid with size 128^2

Polymers \ Particles per polymer	32	64	128	256	512
1	4.95	5.20	5.38	6.43	7.29
10	7.47	9.14	13.90	23.57	35.92
20	7.07	10.39	16.37	29.87	44.82
30	6.61	10.83	18.18	32.40	49.17
40	8.06	13.19	24.11	41.39	62.45
50	7.47	12.74	23.70	42.02	63.25

Table 5.3: Speedup of GPU over CPU with polymers added in grid with size 256^2

Polymers \ Particles per polymer	32	64	128	256	512
1	9.55	9.61	9.84	10.09	10.10
10	10.61	11.77	13.98	17.63	24.88
20	9.68	11.81	15.63	22.63	32.66
30	9.43	11.69	16.61	25.68	37.76
40	10.17	13.31	19.92	31.92	47.89
50	9.55	13.37	20.19	32.77	49.36

Since the computation tasks of molecularDynamics( ) are intensive and it is well parallelized on GPU, the speedup result is impressive.

5.4 Performance with Lennard-Jones algorithm included

Because of that we splitting a kernel function into several kernel functions to perform forced synchronization, the performance of our code is highly affected. It is easy to see this from Table 5.4. There is only one polymer in the grid, which means that there is no need to use Lennard-Jones algorithm. The grid size is 128^2 and there are 5 particles in one box.

Table 5.4: Speedup of GPU over CPU with or without Lennard-Jones algorithm

Splitting kernels \ Particles per polymer	32	64	128
No	9.55	9.61	9.84
Yes	0.64	0.64	0.70

On the other hand, there are too much computation in Kernel_Lennard_Jones( ) function because we need to compare every two node particles between polymers. With the number of polymers getting larger, the execution time of Kernel_Lennard_Jones( ) function grows exponentially. However, Lennard_Jones( ) function costs too much computation and it is extremely slow on CPU. It is responsible for over 90 percent of the computing time on both CPU and GPU. With our parallelized implementation of comparing all the pairs of node particles between polymers, we are able to achieve significant speedup.

Table 5.5 shows the speedup with 5 particles per box and grid size 128^2.We can see that the performance is increasing with the number of polymers and number of node particles per polymer.

Compared with Table 5.2 which has the same grid size and number of particles per box, we can see that the speedup is affected a little but still achieve satisfying

Table 5.5: Speedup of GPU over CPU with Lennard-Jones algorithm included in grid with size 128^2

Polymers \ Particles per polymer	32	64	128
1	0.64	0.64	0.70
10	5.11	15.51	36.78
20	15.94	34.86	59.00
30	25.69	43.09	55.00
40	32.80	47.86	48.31
50	36.54	43.75	43.75

results. With the speedup is from 5 up to 43, the simulation is highly accelerated.

5.5 Multi-GPU performance

We use OpenMP along with CUDA to implement two simultaneous simulations on two GPUs of our server.

First we test our code for the SRD simulation without polymers. Table 5.6 shows the execution time with different number of particles per box in a grid with size 128^2. We can see that the execution time of single GPU and two GPU is almost the same, which means that we are able to double the efficiency by using two GPUs.

Table 5.6: Execution time of SRD in milliseconds for 1000 time steps

Particles \ Single, multi-GPU or CPU	Single	GPU0	GPU1	CPU
5	971.143	976.86	983.418	7460
10	1763.92	1765.74	1779.98	11490
20	3353.16	3358.31	3371.52	29640
30	4953.29	4940.11	4952.81	44600
40	6532.66	6531.79	6542.54	59600
50	8132.45	8124.48	8130.08	74600

Then we test it with polymers added. Table 5.7 shows the execution time with 5

particles per box, 32 node particles per polymer and grid size 128^2.We can see that the execution time of single GPU and two GPU is almost the same, which means that we are able to double the efficiency by using two GPUs.

Table 5.7: Execution time of SRD with polymers in milliseconds for 1000 time steps

Single, multi-GPU or CPU / Polymers	Single	GPU0	GPU1	CPU
1	1994.81	1995.68	2008.99	9800
10	1987.36	2001.86	2013.95	14200
20	2794.95	2819.78	2812.36	19000
30	3613.38	3612.16	3615.01	23800
40	3620.11	3610	3625.51	28600
50	4424.84	4412.96	4428.87	33400

Finally we test it with Lennard-Jones algorithm included. Table 5.8 shows the execution time with 5 particles per box, 32 node particles per polymer and grid size 128^2.We can see that the execution time of single GPU and two GPU is almost the same if we have more than 30 polymers, which means that we are able to double the efficiency by using two GPUs.

Table 5.8: Execution time with Lennard-Jones in milliseconds for 10 time steps

Single, multi-GPU or CPU / Polymers	Single	GPU0	GPU1	CPU
10	189.176	285.09	284.945	1100
20	247.782	300.352	300.246	4220
30	352.506	368.438	366.795	9460
40	508.905	527.305	529.633	16810
50	760.064	763.512	764.573	26270

However, if we have less than 30 polymers, the execution time of multi-GPU is longer than that of single GPU, though it is still shorter than two times of the execution time of single GPU. Currently we are not clear why it is slow if we have small number of polymers.

Comparing the single GPU column in these three tables, we can see that the execution time increase much with polymers added. Even with only one polymer added, the execution time will be twice that of basic SRD simulation. Note that we only to see the row with 5 particles in table 5.6 because the number of particles is fixed to 5 in table 5.7 and table 5.8. The time steps we estimated are 1000 steps for table 5.6 and table 5.7, while 10 steps for table 5.8 due to the long execution time.

Chapter 6

Conclusions

In this thesis, we describe the particle-based SRD method along with modeling of polymers for biophysical simulations. We develop the program in CUDA to accelerate its simulations on GPU with different parameters such as grid size and dimension, particle density, polymer length and density and so on. We benchmark the simulations on a server and compare the performance of GPU with that of CPU.

However, the speedup may be further improved by optimizing our kernel functions, data structures and even the algorithm itself. For example, in the Lennard-Jones algorithm, the node particles are so far away from each other that we do not need to apply any additional forces on them. That means most of the threads in that kernel functions are idle while waiting for a few threads to perform the computation.

We also notice that copying data between host and device memory slows down the simulation significantly. We do the evaluation with all the outputs disabled and only measure the execution time of the computation. In a real simulation, we may need the results of every time step to do analysis or for other purposes. Alternatively, we may also implement the post-processing steps on CUDA GPUs as well to further accelerate the entire simulation.

We are glad to see that our GPU approach achieves the speedup of 10 times faster than the CPU alone for basic SRD simulation, while the speedup is up to 60 times with polymers added and up to 40 times with Lennard-Jones algorithm included.

In conclusion, GPU accelerations is a cost-effective, high-performance computing method that is well-fitted for simulations in scientific research, especially for large-scale simulations that usually take weeks or months to complete by using CPU-based programs.

Bibliography

[1] A. Malevanets and R. Kapral, "Mesoscopic model for solvent dynamics," *The Journal of Chemical Physics*, vol. 110, no. 17, pp. 8605–8613, 1999.

[2] T. Ihle and D. Kroll, "Stochastic rotation dynamics: A galilean-invariant mesoscopic model for fluid flow," *Physical Review E*, vol. 63, no. 2, 2001. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.63.020201

[3] E. Tüzel, T. Ihle, and D. Kroll, "Dynamic correlations in stochastic rotation dynamics," *Physical Review E*, vol. 74, no. 5, 2006. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.74.056702

[4] E. Tüzel, M. Strauss, T. Ihle, and D. Kroll, "Transport coefficients for stochastic rotation dynamics in three dimensions," *Physical Review E*, vol. 68, no. 3, 2003. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.68.036701

[5] J. E. Lennard-Jones, "On the Determination of Molecular Fields. II. From the Equation of State of a Gas," *Royal Society of London Proceedings Series A*, vol. 106, pp. 463–477, Oct. 1924.

[6] *CUDA C PROGRAMMING GUIDE*, 5th ed., nVIDIA, October 2012.

[7] S. Lee and R. Eigenmann, "Openmpc: Extended openmp programming and tuning for gpus," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[8] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010.

[9] J. Kingsley, "A particle-based method for the simulation of complex fluids and polymer solutions," 2011.

[10] J. E. Lennard-Jones, "Cohesion," *Proceedings of the Physical Society*, vol. 43, no. 5, pp. 461–482, Dec. 2002.