Parallelizing a Particle Simulation - CUDA

Tongge Wu[1], Debbie Liang[2], Zhuoming Zhang[2]

[1] *Department of Mechanical Engineering, University of California, Berkeley*
[2] *Department of EECS, University of California, Berkeley*
March 27th, 2020

1 Introduction

Molecular dynamics simulation, a well-studied problem in scientific computing, is widely used in biochemistry, mechanics, and astrophysics. The basic problem setting is as follows:

- Given a domain (i.e., square) and boundary conditions (i.e., fully elastic recover).
- Initialize positions and velocities of a group of particles with prescribed masses.
- The force between any pair of particles is a prescribed function of the distance between the two particles.
- All particles move within the domain according to Newton's second law in a finite time interval.

In this assignment, we optimized a naive molecular dynamic simulation on an Nvidia P100 Graphic Process Unit (GPU). Here is our contribution:

- Tongge Wu works on optimizing the code on GPU, testing scalability, providing mathematical support to our algorithms and writing the report
- Debbie Liang works on optimizing the code on GPU, testing scalability, visualizing data, and writing the report
- Zhuoming Zhang works on optimizing the code on GPU, testing scalability, testing correctness, and working on timing analysis, and writing the report

2 General Framework

A naive particle simulation costs FLOPS of $\mathcal{O}(n^2)$, where n is the number of particles. Because for each particle, we need a loop to scan all other particles to get the total force. In general, *Cell Lists* and *Verlet Neighbor Lists* are two common approaches to optimized the naive particle simulation [1].

In this project, we used Cell Lists to achieve $\mathcal{O}(n)$ performance. When all particles' position was given, the cell ID was calculated for each particle ID. This step was embarrassingly parallel and we use one thread to process one particle. By the end of calculation, there was a map between particle IDs (`particles_index`) to cell IDs (`particles_cell_index`).

Using the map between particle ID to cell ID, for each particle, we could find the neighbor cell of its cell. Looping through all particles in the neighbor cells and in the center cell yielded the total force applied on this particle, since our cell length was designed to be slightly larger than the cut-off distance in the force function. Moreover, in the actual implementation, we looped through all cells (non-empty cells), since it is easy to find the neighbor cell for the center cell.

With all particles' forces, hence, accelerations were calculated, we would move the particle for a given time step using the Verlet ODE integrator. This step was also embarrassingly parallel and we use one thread to move one particle.

In order to synchronize the all threads in GPU, we added `cudaDeviceSynchronize` after mapping, sorting, applying force, and moving, because these kernels depended on the data provided by the previous kernel.

3 Optimizations

3.1 Mapping and Sorting

When the map between particle ID and cell ID was constructed, the `particles_index` was in order and the `particles_cell_index` was out of order, due to the nature of thread numbering. Then the `particles_cell_index` was sorted (`thrust::sort()`) as key and `particles_index` was paired sorted as value. This step was implemented since we need to loop through the cell ID for total force calculation.

In the force calculation function, each thread was assigned to a `particles_cell_index`, physically it meant a thread processed a particle, but logically a thread processed a cell. That is, there might be multiple thread processing multiple particles in one cell, and no thread processing the empty cell. Each thread would find the neighbor cells of its cell, and get the positions of all particles in the neighbor cells to calculate the force applied on this particle. Meanwhile, the thread would also get the positions of other particles within its cell (if there is any) to yield a total force.

To achieve this loop, each thread would need to know the range of particle ID of each physical cell, thus, we had a `cell_start` variable, whose content was index indicating the range in `particles_index` of each physical cell.

Sorting and looping over cell strategy, compared to loop over particle (in order) directly, gave us an exciting 52% perforamce increasing.

3.2 Shared Memory using L1 Cache

Within a GPU Stream Multiprocessor (SM), all the Streaming Processors (SPs) (thread in code) shared the same L1 caches. To take advantage of the fast memory, we explicitly declared `__shared__` memory for `particles_cell_index` when we constructed the `cell_start`. Since in kernel `fill_cell_start`, each thread would access the `particles_cell_index` array three times contiguously at index tid, tid - 1 and tid + 1. Explicitly reserving a piece of cache among all the threads on the same block ensured that each thread can make use of the value in the `particles_cell_index` whenever it needed and get it fast.

Another kernel used explicitly shared cache is `move_gpu`. In this kernel, the `particles` array containing particles was accessed three times by each thread. Declaring a piece of cache ensured the array staying in the cache.

Shared memory in cache improved the performance of our code by about 2 %.

3.3 Symmetric Force Matrix

If we have a force matrix, where each row and column represents a particle, and each nonzero entry represents the magnitude of the force between the two particles. It is trivial to argue that the force matrix would be sparse and symmetric due to Newton's third law. Therefore, we can half our FLOPS using this fact that if we calculate f_{ij}, there is no need to calculate $f_{ji} = -f_{ij}$. However, to use the symmetric update strategy, two threads might be updating the same particle at the same time. Thus, atomic operations were needed, especially `atomicAdd` was needed for acceleration terms.

Since CUDA 6.0, CUDA compiler supports `atomicAdd` for double. However, the compiler requires a flag `-arch=sm_60` in the cmake file to enable it. Alternatively, we could implement the `atomicAdd` using `atomicCAS` invented before the CUDA 6.0. If we used the in-house version `atomicAdd`, we observed 10% performance increase, and if we used the native `atomicAdd`, we observed additional 13% performance increase[1].

[1]Implement the native `atomicAdd` required modifying the cmake file, which is not permitted as the writting time

3.4 Minimize Branching

GPU runs true Single Instruction Multiple Data (SIMD) within one SM, that is, one block in our code. Which means when there is a branch (`if` or `while` statement), any two threads branching into different directions would have to wait for each other, thus, branching increases the potential thread idle time, negatively impacts the performance.

To minimize branching, we padded zeros to the boundary of the cells in `cell_start`. So when we check the neighbors of a cell, there were no `if` conditions to avoid accessing cells outside the boundary. In kernel `fill_cell_start`, we prefill the boundary with -1 as padding when sharing the cache so that there was no `if` condition to check the boundary. Since we implemented these branch-avoiding strategies in the beginning, there was no measurement to show the performance increase.

3.5 Unroll

To iterate through the 9 cells surrounding the current cell, we first created a vector that holds the 9 cells and used a for loop to iterate. In the process of optimizing our code, we discovered that unrolling the `for` loop to be 9 similar blocks of code improves the run time. The improvement may due to minimizing the branch prediction. In conclusion, unrolling sped up our code by up to a factor of 3.

4 Performance

The results were checked with the naive particle simulation ($\mathcal{O}(n^2)$) solution.

4.1 Complexity with simulation scale

When we utilize hardware SM and SP, that is 56 SMs with 64 SPs per SM, 3549 CUDA cores in total, then progressively scaled up the simulation, the computation time was collected and listed in Table.2 (see. Sec. 6). A linear regression analysis showed that the complexity was $\mathcal{O}(n)$ (Eq.1). The data was also shown in Fig.1.

$$\log(T) = 0.98\log(n) - 13.04 \quad \text{with} \quad R^2 = 0.996 \tag{1}$$

4.2 Weak scaling

To obtain the weak scaling characteristic, we assigned each SM with 256 particles, then progressively increased the number of SM calls (that is blocks in our code). The computation time was in Table.3 (see. Sec. 6).

If we define the Efficiency as

$$\text{Efficiency} = \frac{\text{Time(1 SM)}}{\text{Time(p SMs)}}$$

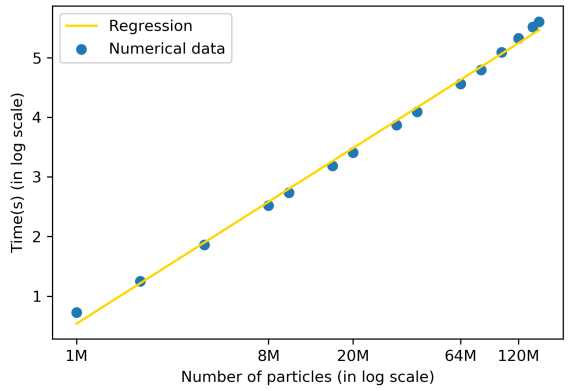

Figure 1: Computational time as a function of simulation scale. Blue dots are numerical experiment results, while the yellow line is the linear regression of these data points. All data points are log-log transformed from Table.2

we can plot the efficiency as a function of number of SMs, see Fig.2.

We observed that when the requested number of SMs was less than 17, we had almost perfect weak scaling, that is, each SM spent the same time processing each group of particles simultaneously. However, when the number SMs was greater than 25, the data fell onto a new plateau, which was less than half of the initial efficiency. It suggested that the communication between each Graphic Processor Cluster (GPC) and the cache would affect the weak scaling performance, since each GPC control 10 SPs, when we have more than 2 GPC (25 SPs cases), each SM would spend some time to wait or to communicate with each other. Increasing the number of particles also required a larger cache size, otherwise, some data were located in slower memory instead of cache.

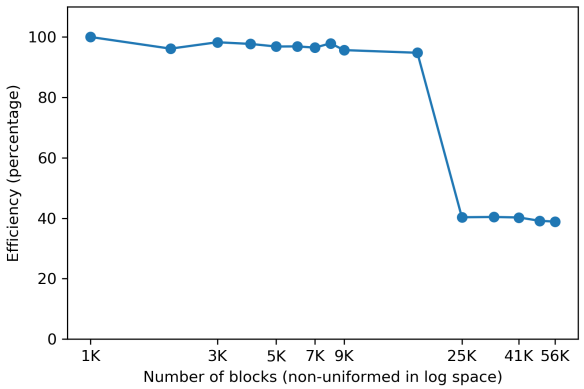

Figure 2: Weak scaling: efficiency as a function of number of SMs.

4.3 Performance Compared with Naive Implement

To compare our implement with the naive GPU implement, we collected data of both implements with varying number of particles, see Table.4 (see. Sec. 6). As shown in Fig.3, we observed that our implement had a linear scalability ($R^2 = 0.74$), this linear scalability was not as good as the one in Eq.1. Because for a small number of particles, the communication overhead was not negligible, and it affected the linear scalablity. While the naive implement had a quadratic scalability ($R^2 = 0.985$), which met our expectation.

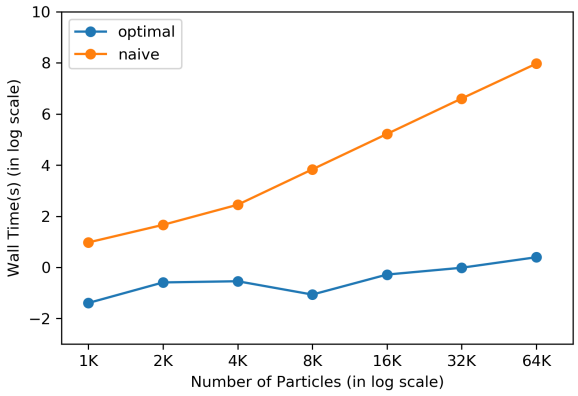

Figure 3: Computational time as a function of simulation scale. Blue dots are optimal implement, orange dots are naive implement. All data points are log-log transformed from Table.4

4.4 Performance Compared with OpenMP and MPI Implement

To compare our implement with the optimal implement using share-memory (OpenMP) and distributed-memory (MPI) architecture, we also collected data of all implements with varying number of particles, see Table.5 (see. Sec. 6). As shown in Fig.4, we found out that all implementations possessed linear scalability ($R^2 = 0.999$ for OpenMP case, $R^2 = 0.998$ for MPI case, $R^2 = 0.983$ for GPU case). We observed that GPU implementation had a slope of 0.768, which was also less than the Eq.1 due to relatively small number of particles. While MPI had a slope of 1.010, and OpenMP had a slope of 1.007, which indicated almost prefect linear scalability. Meanwhile, we observed that the run time using GPU was roughly $1/10$ of OpenMP and $1/8$ of MPI, which suggested that the our implement of GPU code was much efficient than our optimal CPU code, especially on a large simulation scale.

4.5 Runtime Breakdown

To obtain the timing analysis of our GPU code, we used Nvidia `nvprof` comes with CUDA 10.1 installed on Bridge.

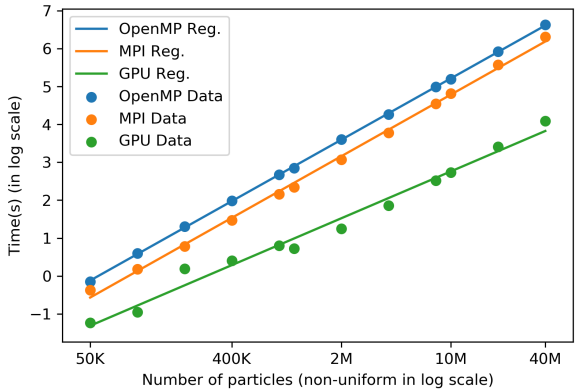

Figure 4: Computational time as a function of simulation scale. Dots are data of each implement, lines are linear regression of each implement. All data points are log-log transformed from Table.5. Data were obtained by OpenMP with 68 threads and MPI with total 68 ranks on 2 nodes

The computing time was the time of GPU working on calculation excluding memory allocation and kernel launch time. The computing time divided by total run time was Compute Utilization, which was provided by Nvidia Visual Profiler. Basically, when the number of particles was increased, we observed that Compute Utilization increased gradually, see Fig.5. It indicated that the percentage overhead of launching a kernel and/or cudaMalloc decreased when the simulation scale increased.

We also found that different kernels showed different memory bandwidth utilization, as shown in Table.1. Compare with the compute utilization (over 90%), our code was bounded by the compute utilization. We could keep optimizing the memory and apply more shared memory in our code to further increase the simulation speed.

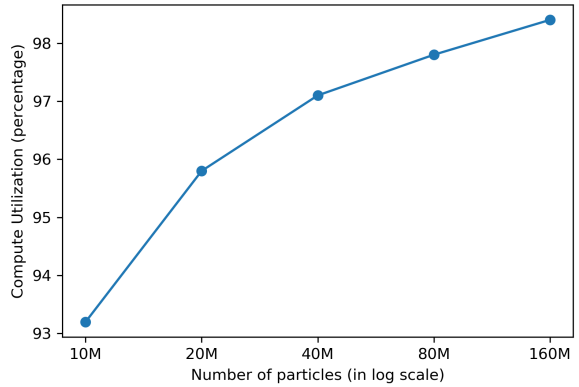

Figure 5: Runtime analysis: Computation utilization as a function number of particles.

Kernel Descrip.	Kernel Time	BW Util.
Compute Force	39.2%	75%
Move Particle	19.4%	62.5%
Sort By Key	21.8%	22.27%
Construct `Cell_Start`	7.8%	100%
Construct Map	6.3%	100%
Clear `Cell_Start`	5.4%	100%

Table 1: Memory bandwidth utilization and executing time of each kernel

5 Conclusion

In the scope of this project, we optimized the serial molecular dynamic simulation, designed and optimized parallel molecular dynamic simulation using a single GPU. We achieved $O(n)$ complexity, which was better than naive $O(n^2)$ complexity. The weak scaling test showed our efficiency stays on two plateaus when the number of blocks was smaller than 17 or was larger than 25, which is optimal on a GPU. The comparison between OpenMP, MPI, and GPU demonstrated the strong linear scalability in all cases, and an efficient GPU implementation. Moreover, we found our code was bounded by computation utilization, not memory bandwidth. In conclusion, we achieved a reasonable parallel version of the basic molecular dynamic simulation on GPU.

References

[1] Victor Eijkhout with Robert van de Geijn and Edmond Chow. *Introduction to High Performance Scientific Computing*. http://www.tacc.utexas.edu/~eijkhout/istc/istc.html. lulu.com, 2011. ISBN: 978-1-257-99254-6.

6 Appendix

Wall-time (s)	# of particles
2.07123	1M
3.49473	2M
6.4183	4M
12.4431	8M
15.3946	10M
24.2205	16M
30.2132	20M
47.8294	32M
59.7026	40M
95.445	64M
121.034	80M
162.12	100M
205.123	120M
248.194	140M
269.91	150M

Table 2: Relation between simulation scale (M: $\times 10^6$) and computation time

Wall-time (s)	# of particles	# of SMs
0.0999432	256	1
0.103951	512	2
0.101706	768	3
0.102289	1024	4
0.103198	1280	5
0.103147	1536	6
0.103592	1792	7
0.102074	2048	8
0.104475	2304	9
0.105451	4352	17
0.247912	6400	25
0.247349	8448	33
0.248362	10496	41
0.255306	12544	49
0.257109	14336	56

Table 3: Relation between simulation scale and computation time for fixed number particle per SM (256)

Optimal (s)	Naive (s)	# of particles
0.248102	2.65163	10K
0.554248	5.27649	20K
0.58051	11.5907	40K
0.344501	46.2721	80K
0.75386	185.052	160K
0.984039	739.82	320K
1.48573	2895.58	640K

Table 4: Relation between simulation scale (K: $\times 10^3$) and computation time for optimal and naive implement

# of particles	OpenMP (s)	MPI (s)	GPU (s)
50K	0.866901	0.68855	0.292241
100K	1.82691	1.20377	0.384641
200K	3.70489	2.19192	1.21542
400K	7.30698	4.3707	1.49869
800K	14.474	8.69291	2.2238
1M	17.2298	10.3756	2.07123
2M	36.6943	21.5452	3.49473
4M	70.8856	43.7266	6.4183
8M	147.385	94.1803	12.4431
10M	180.926	123.164	15.3946
20M	373.195	262.55	30.2132
40M	759.171	549.158	59.7026

Table 5: Relation between simulation scale (K: $\times 10^3$; M: $\times 10^6$) and computation time for OpenMP with 68 threads, MPI with 68 ranks on 2 nodes, and P100 GPU