CS 267: Parallel Computing **(Due: 04/02/20)**

Homework Write-Up #2.3

Instructor: A.Buluc, J.Dummel, K.Yelick *Name:* Bo Li, Philippe Ferreira De Sousa, Jingran Zhou

1 Members & contribution

- Philippe (non EECS/CS major): Wrote the base code adapting the design from the OpenMP version adding communication between the CPU and the shared memory of the GPU and splitting the work in blocks and threads. Wrote about the design decisions.
- Bo Li: Debugged code and ran the experiments to test the performance and analyze the runtime. Wrote about performance and run-time profiling.
- Jingran Zhou: Debugged and optimized CUDA code; Drew performance plots; Benchmarked our CUDA code against the starter code and the OpenMP code; Wrote about performance, data structures, synchronization, and benchmark results.

2 Performance & data structures

Figure 1 shows how our code performs with varying numbers of particles. The specific values in this plot are presented in Table 1.

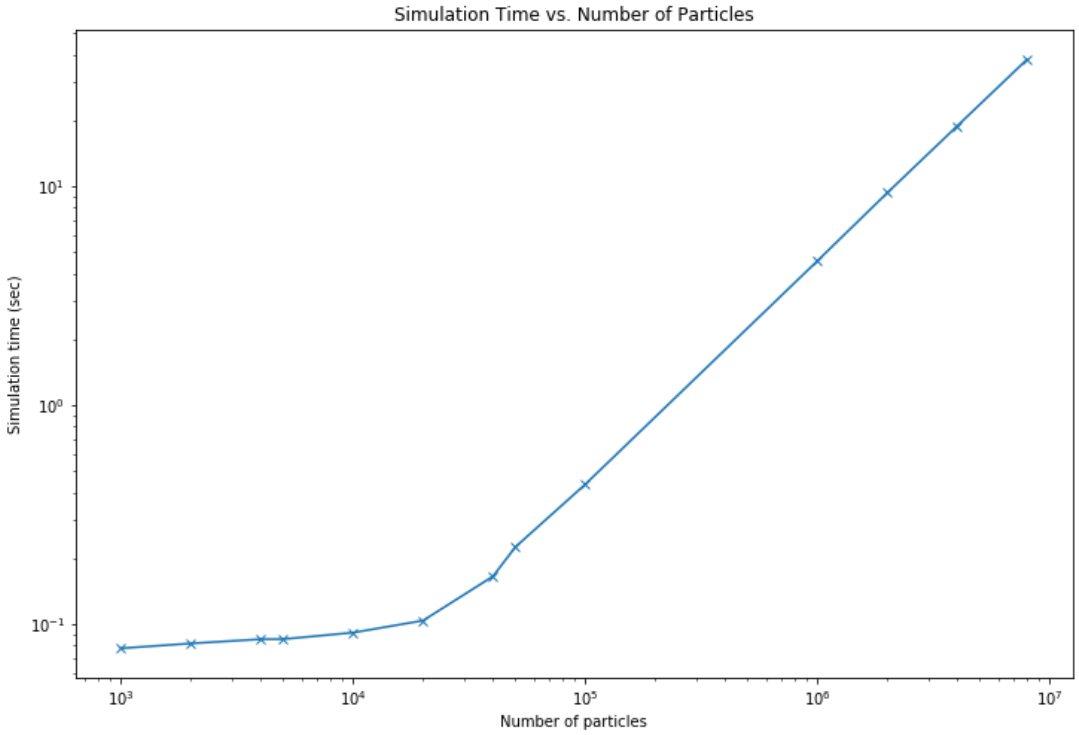

Figure 1: A plot in log-log scale showing our parallel code performance.

Observe that when the number of particles is roughly less than 50,000, the simulation time of our code is approximately constant. After this turning point, however, the run-time scales linearly with the number of particles. This inconsistency can be explained by the capacity of the GPU. In our implementation, each particle is handled by one thread. The simulation time is relatively constant before the turning point probably because the number of particles is less than the total number of available threads in the GPU. To test this hypothesis, we need to look at the specifications of the GPU.

The GPU on which our code runs is NVIDIA Tesla P100, which has 3,584 FP32 cores and 1,792 FP64 cores. Since the fields of a particle are `doubles` in this homework, we assume that our code runs on the 1,792 FP64 cores. Each CUDA core can execute 32 threads. Therefore, with a single GPU, our code should be able to sustain $1792 \times 32 = 57,344$ threads theoretically. This supports our hypothesis, as the turning point indeed lies between 10^4 and 10^5.

Number of Particles	Time (s)
1000	0.048168
4000	0.050735
16000	0.07102
64000	0.19994
256000	0.77587
1024000	3.14693
4096000	12.6584
16384000	50.8511

Table 1: How the simulation time of our code varies with the number of particles.

The data structure we used to achieve this is **dynamic arrays**. Each bin is an integer array with a fixed capacity and a varying size, with each entry denoting the ID of a particle. The bins, as a whole, are represented as a 2D integer array. At each simulation step, we clear all the bins, then add each particle to its corresponding bin. For parallelism, each particle is handled by one thread. That is, each thread is responsible for adding a particle's ID to the corresponding bin.

However, as in Homework #2.1, this setup might result in race conditions when multiple threads try to add particles to the *same* bin. Our solution to this concurrency issue is discussed in the next section.

3 Synchronization techniques

In our OpenMP implementation, we used `omp_lock_t` for synchronization. The equivalent of locks in CUDA is **atomic operations**. Specifically, we used `atomicAdd()` in our code. Concretely, whenever a thread is adding a particle ID to a bin, it invokes `atomidAdd()` to achieve two goals: 1) to increment the size of the bin *atomically*, which ensures there is no race condition; and 2) to obtain the old size of the bin before the atomic operation, which is precisely the index where we insert the particle ID.

4 Design decisions

Number of Particles	Our CUDA Time (s)	Double bin size (s)	With neighbor array (s)
1,000	0.0481681	0.0610244	0.0662612
4,000	0.0507354	0.0666899	0.0710739
16,000	0.071024	0.0822761	0.0826435
64,000	0.199941	0.251351	0.329485
256,000	0.775865	1.10196	1.54356
1,024,000	3.14693	4.66362	6.78338
4,096,000	12.6584	19.3349	28.1992
16,384,000	50.8511	77.7015	113.675

Table 2: Benchmark results of our CUDA code against other design attempts

We have tried to double the size of the bins as it worked well with OpenMP, but we see a net degradation of the performance.

Then we tried to generate the array of neighbor of each bin in advance in the init_simiulation function in order not to have to handle the edge cases when iterating on the neighbors to compute the forces, testing each time whether we are on the border or not instead of doing it once at the beginning. Nonetheless we see an even bigger degradation of the performance while doing that. We understand that those arrays of neighbors need to be fetched form cache each time, as a result rechecking whether we are on the border or not is quicker.

We also tried to save pointers of the particles in the arrays of bins instead of the index of the particle to then fetch in from parts. But it did not give a correct simulation of the interactions without certainty on the cause.

5 Benchmark results

We benchmark our CUDA code against the naive starter version and the best-performing OpenMP version (with 272 threads), varying the number of particles. The results are shown in Figure 2, while the corresponding numbers are recorded in Table 3.

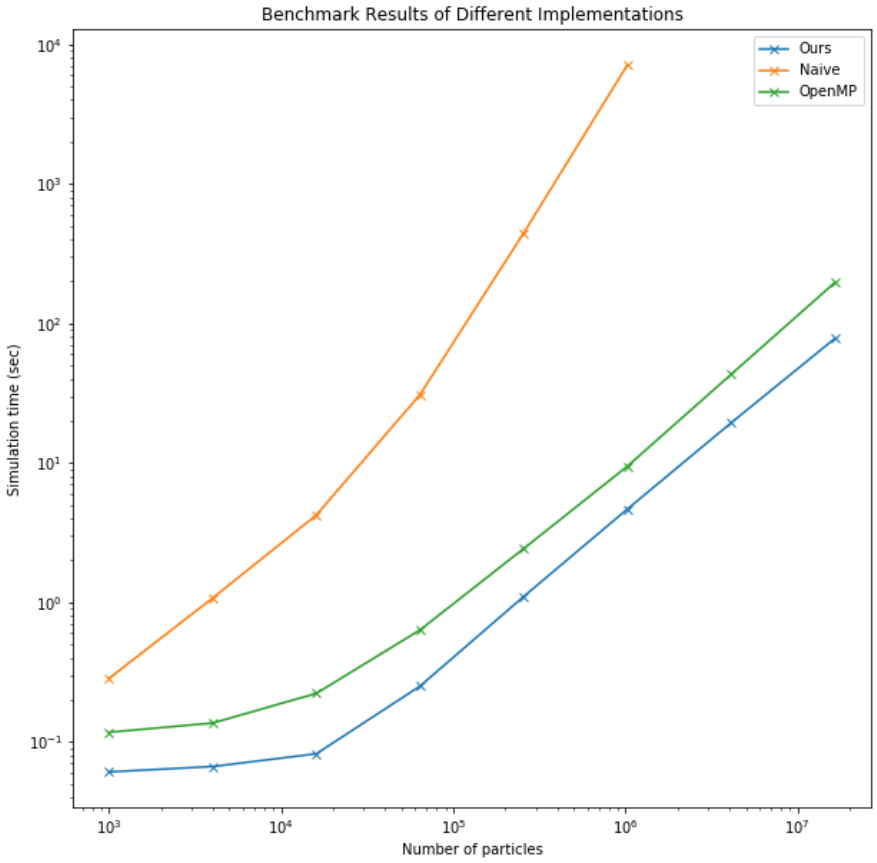

Figure 2: A log-log plot comparing three implementations: our CUDA code, the naive starter code, and our best OpenMP code.

From the plot, we clearly observe that the starter naive version runs in *quadratic* time, which is unsurprisingly inefficient. It is so slow that the job cannot even finish for large particle counts. By contrast, our CUDA and OpenMP implementation are $O(N)$ after the "elbow point." A possible explanation for the sub-linear performance before the "elbow point" is already presented in Section 2.

Furthermore, note that our CUDA implementation is noticeably faster than the optimal OpenMP code. For instance, with about 16 million particles, the CUDA version takes about 78 seconds, more than twice as fast as OpenMP's 196 seconds.

Number of Particles	Our CUDA Time (s)	Starter Time (s)	OpenMP Time (s)
1,000	0.0481681	0.285551	0.134949
4,000	0.0.0507354	1.07478	0.152778
16,000	0.071024	4.23024	0.302797
64,000	0.199941	30.8576	0.734635
256,000	0.775865	444.094	2.47878
1,024,000	3.14693	7105.45	10.1955
4,096,000	12.6584	TLE	41.544
16,384,000	50.8511	TLE	166.265

Table 3: Benchmark results of our CUDA code, the naive starter code, and our best OpenMP code.

6 Run-time profiling

We use `nvprof` command to test the runtime of our simulation. Our program mainly consists of four parts: 'reset bin sizes', 'rebin particles', 'compute forces' and 'move particles'. According to the data from `nvprof` result on the test of one million particles, the 'reset bin sizes' takes 0.25% of the total time, 'rebin particles' takes 19.05%, 'compute forces' takes 74.85% and 'move particles' takes 5.72%. We can see that most of time is spent on the force computation and rebinning particles. In the force computation section, we need to get particle information from its neighbor bins and apply the force on the particles bins. And in the rebinning particles section, we use the function `atomicAdd`, which is an atomic operation. It only allows one thread to change the shared variable, which makes it slower. For the synchronization, in the main function, `cudaDeviceSynchronize` is called after each simulation steps and at the end of all the simulation. Therefore, it is almost the same as the total runtime and takes 94.94% time of all API calls.