CS 267 Homework 2 Part 3

Xingyou Song [*], Yao Yuan [†], Jingbo Wu[‡]

March 9, 2018

1 Old Algorithm (Serial)

Recall from our previous part 1 submission that we used the following method to achieve $O(n)$ time: Our technique was to separate the space (formed by the bounding box of the entire set of points), into a grid of $G \times G$ blocks, where G is the partition number of each of the x or y dimensions. Let n be the number of particles in the simulation. For our serial implementation, $G = n$, while in our parallel implementation, $G = \sqrt{n}$. (For these two cases, they are also set as the particle interaction minimum distance, in case the grids are too small). Then for any neighboring blocks, we brute-forced the force computations over the union of the particles.

The order then follows:

- For each block, compute inner forces within the block.
- For each block, compute the forces of its particles affected by the block's 8 neighbors.
- Move all of the particles as needed.

1.1 Proof of Concept

In the serial setting, in every stride interval (1-dimensional) of other the x or y axis, there is an average of 1 point. The idea here is to shortcut the force-checking process by only brute-force calculating forces in between neighboring blocks. More rigorously, suppose we assume that each of the n particles has uniform probability to be placed in one of the $n \times n$ blocks. Our total calculation will then be

$$\sum_{neighboringblocks A,B} brute\_force(A, B)$$

[*]xsong@berkeley.edu - Xingyou wrote the report, presented techniques and strategies for optimization, and coded some parts

[†]yao_yuan@berkeley.edu - Yao coded a significant part of the code in C and tested using various compiler settings

[‡]wu622@berkeley.edu - Jingbo mainly contributed to the report and coding.

where the runtime of $brute\_force(A, B)$ is $|A||B|$ to brute force all of the points' forces. Note that by linearity of expectation, we may then consider the expected total calculation as

$$O(n^2)\mathbb{E}\left[brute\_force(A, B)\right]$$

After some calculation, we find that $\mathbb{E}\left[brute\_force(A, B)\right] \leq O(1/n)$ which implies the linearity result.

2 GPU/CUDA Modifications

2.1 Ordinary Implementation

In the global memory updating solution, we allow each block in the particle space (renamed "bins" in the code) of the grid to correspond to each thread on the GPU. Every physical GPU block corresponds to a $1 \times k$ tile on the grid, where k is the blockDimension.x (i.e max number of threads on a block). Every thread would simulate/compute in its bin. However, for location updating, we then updated a global memory (called redundantBins in the code) so that particles from bin A go to bin B from time t to $t + 1$. The next time step would load the global memory back into the GPU again. Thus, every loop iteration consists of the following:

- Load from redundantBins (global memory) into corresponding GPU blocks.
- Each thread computes the particles' forces and new locations in the thread's corresponding bin.
- (Synchronize all threads first) Send particles to the corresponding positions in the redundantBins for next iteration.
- Repeat.

2.2 Possible Improvements

Note that there are some flaws with this approach that could or cannot be improved:

- The global memory updating scheme is a large overhead, as we're making 2 reads of the entire grid memory at every time step.
- Rather than each GPU block representing a $1 \times k$ tile, A block might represent a $\sqrt{k} \times \sqrt{k}$ tile instead, because this requires fewer memory operations for each GPU block. (i.e. the perimeter of this square is $O(\sqrt{k})$, rather than the $O(k)$ from the $1 \times k$ tile. However, this method is very coding intensive (we attempted to do so, but there are an incredibly large number of edge cases related to computing indices), because CUDA memory copying is inherently sequential.

- Also, this square-tiling approach does not actually remove the global memory updating scheme cleanly (but can improve a few memory updates), because it doesn't help with the case when a particle has an extremely large speed that forces it to jump between very far bins on the grid. Since CUDA doesn't allow inter-block communication, (e.g. we can't just tell thread A corresponding to a high speed particle in bin A, to communicate with thread B to collect that particle into bin B in the next time step, using their corresponding blocks), global updating seems to be needed.

- We are not utilizing shared memory for threads on the same block - this means that for a $1 \times k$ block, every thread accesses its 8 neighbors, with some accesses shared between threads (for a total of $8k$ global reads) In order to fix this, we propose preloading all of the neighbors (and itself) for a shared memory for the $3 \times (k+2)$ "perimeter tile" covering the original tile. This would give us $3(k+2)$ global reads, and $8k$ shared memory reads instead. Since shared memory is on the order of 100 times faster than global memory, theoretically we should achieve $8 * 100/(3 * 100 + 8) = 2.59$x speedup for memory reads. For $\sqrt{k} \times \sqrt{k}$ blocking, the naive memory approach would give us $8k$ global reads again, and the local memory optimization would give us $(\sqrt{k}+2)^2$ global memory reads with $8k$ shared memory reads instead, which should achieve $8 * 100/(100 + 8) = 7.4$ times memory speedup.

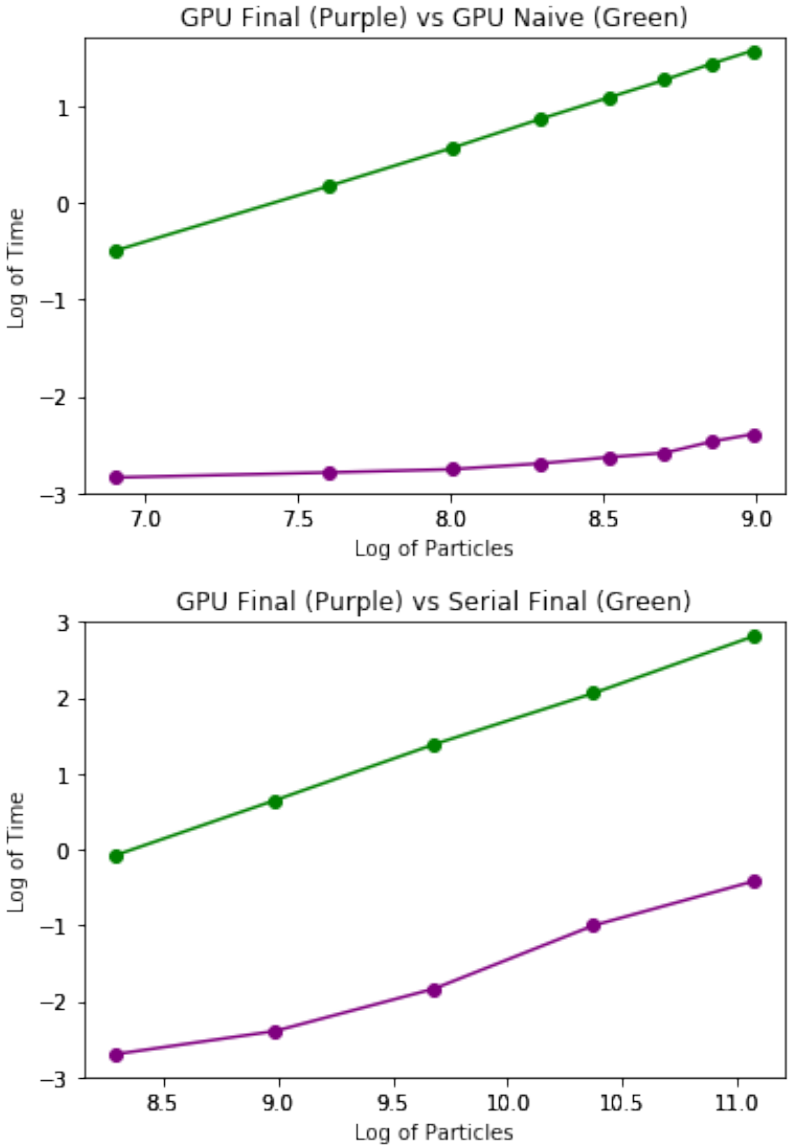
2.3 Attempted Improvements

- We coded a 2d version of the blocking procedure, using $\sqrt{k} \times \sqrt{k}$ GPU blocks. However, this actually resulted in a slightly slower performance. This is possibly due to lack of cache use in this implementation, as well as more overheads from computing correct indices.

- We coded a version of shared memory for the $\sqrt{k} \times \sqrt{k}$ block case, but found that it this was even impossible due to hardware limitations even for $k = 64$ threads. This is because of the small size of the shared memory in the GPU. Furthermore, we suspect that while theoretically if this were allowed, the overhead from the if-statements needed from computing correct indices between global memory and shared local memory would drastically affect performance as well.

2.4 Results

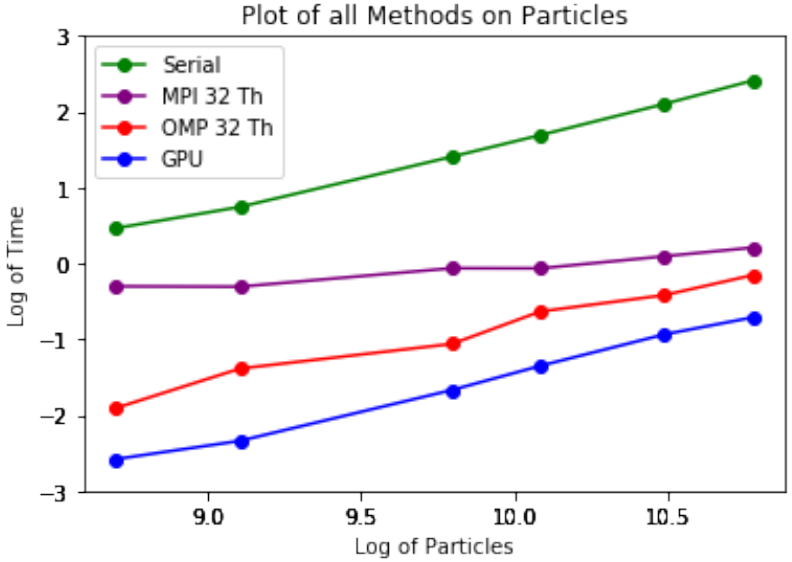
Figure 1: GPU Final Code vs GPU naive (Avg Speed up 35.29, GPU final log-log slope avg is 0.20



Note that as we have more particles, we have slightly lower slope for our GPU-final code, while serial and GPU-naive codes' slopes are very consistent.

2.4.1 Comparison to Other Parallelism

Figure 2: Performance for Serial, MPI 32 Threads, OpenMP 32 Threads, GPU



We found that ultimately, the GPU still worked better than the 32-threaded MPI and OpenMP versions, for particle sizes (6000-48000). It is likely that from viewing the slopes of the curves, the GPU will remain the fastest, although MPI may overtake OpenMP for larger particle sizes. [1]

2.5 Strengths and Weaknesses of CUDA

2.5.1 Strengths

CUDA allows for very large parallel memory reads, which makes it suitable for when there are a massive amount of particles (we're able to simulate orders of magnitude more particles than other frameworks), unlike MPI, which is limited by a memory reading bandwidth. It also allows for a significant number of threads to run, which also increases parallelism.

2.5.2 Weaknesses

The problem of communication in the GPU is one of the greatest limitations - unlike MPI, there is no good method of allowing different GPU blocks to communicate with each other than merely copying back and forth between global memory, which makes it ineffective for computations that require shared resources. This also means that topology in a problem can't be exploited (i.e. the $\sqrt{k} \times \sqrt{k}$ doesn't actually get much speedup even if we've reduced the order of the perimeter sums). Lastly, as we've seen from the hardware limitation, we realized that the cache size for each block is actually quite small and doesn't scale as well as the number of threads allowed.

[1] We did not simulate more due to overloading the XSEDE allocation - our wait time went to 80 hours for a batch job.