COMPSCI 267 HW 2.3 - Parallelizing a Particle Simulation With GPU

Group 66: Sayan Das, Aditya Kumar, Varun Jadia

1. Introduction

In this report, we discuss how we implemented an efficient and accurate particle simulation model using GPUs. In previous homeworks 2.1 and 2.2, we used multiple CPU threads with shared memory (openMP) and message passing (MPI) respectively to speed up the simulation, but CPU threads are often limited and message passing requires significant communication across processes/machines. Here we explore the high computing power GPUs offer through their many cores, building mainly off of the shared memory approach of HW 2.1.

2. Previous Approach

In HW 2.1, we used binning (square bins of size cutoff in each dimension) and a DP approach to process the bins in the grid to get particle simulation to a nearly linear runtime. In the DP approach, for each bin, we apply forces between all cross products of particles: bin x bin, bin x right bin, bin x upper left bin, bin x upper bin, and bin x upper right bin, when the respective bins exist, and process all the bins in this manner. To parallelize the code, we split the grid into chunks of 2 consecutive full rows, then used OpenMP to statically assign these chunks to available CPU threads. Since all CPU threads resided on the same machine, we could store particle and grid data in shared memory, using an array of arrays of lists to handle the dynamic bin assignments of particles.

However, CUDA operations on GPU cannot mutate lists, so using the CPU to mutate the bins would be extremely inefficient. OpenMP also implicitly handles chunk assignment to CPU threads for us, but to use the GPU, we need to explicitly deal with the mapping of work to GPU threads.

3. CUDA Implementation

To get replace lists handling dynamic bin assignments of moving particles, we decided to represent the grid data structure using two arrays, one (gpu_sortedParts) of all the particle ids sorted by bin (within each bin they are unsorted) and stored in row-major order with respect to the bin, and the second (gpu_prefixSum) storing the position in gpu_sortedParts of the first particle belonging to that bin at the bin's respective index (also row-major order with respect to the bin). To help generate these arrays, a third array (gpu_binCounts) is used to count the number of particles within each bin.

Because of the abundance of GPU threads and to minimize computing time of each individual thread, we decided to implement monolithic kernels, so each thread was only responsible for the calculation/processing of just 1 bin/particle respectively, depending on the usage. We first assigned a GPU thread for each respective particle to calculate which bin they lay in, and atomically increment the respective bin's particle count in the binCounts array, to avoid data races on particle count between threads processing particles in the same bin. Then we used thrust::exclusive_scan to implement prefixSum based on the binCounts array, storing it in gpu_prefixSum. Using the prefixSum results, we assign a GPU thread for each respective particle to recalculate which bin they lay in, then store it in the gpu_sortedParts array in the region allocated for that bin (indices [gpu_prefixSum[bin], gpu_prefixSum[bin+1])). To ensure that each particle is assigned to a unique position in gpu_sortedParts, the GPU thread looks up the beginning position of the bin's particles in gpu_sortedParts by fetching gpu_prefixSum[bin], then atomically decrements the bin's binCount in gpu_binCounts and uses the prior value of the binCount to place the particle in gpu_sortedParts at index gpu_prefixSum[bin] + binCount's prior value - 1. This atomic decrementation allows every particle within a bin to get a unique prior value of the binCount, as well as resets the binCounts to zero for the next simulation step. Since each particle is added to a unique position in gpu_sortedParts, we don't need to synchronize this write. To calculate the forces applied on each particle, we assigned a new GPU thread to process each bin in the grid, using the DP approach to process the grid and applying a symmetric force between a particle and its neighbor from HW 2.1 to keep the total number of computations low. To avoid data races since different threads may be updating particles within the same bin, an atomicAdd updates a particle's acceleration during force application. Finally, we assign a GPU thread for each respective particle to calculate the particle's new position after the force has been applied. Since each thread processes its own particle, there is no need for additional synchronization.

Our implementation means that during bin processing for force application, we need a lot more GPU threads than particles since there are a lot more bins than particles. Most of these threads are wasted and do no work when their respective bin is empty, and the implementation may have poor work balancing if a bin and its neighboring bins have lots of particles, but the rest of the grid has few particles, resulting in a few threads doing a lot of work, but the rest doing no work. Since the density of the particle simulation is very low, we can assume that a bin will not have a very high number of particles in it/nearby it, so one thread may not be a major bottleneck, but some load imbalance exists. To address these concerns, another design was considered to remove the symmetric application of force and only allocate a GPU thread for each particle during force application. Instead of using the DP approach, the particle would check its bin and all neighboring bins for neighboring particles and apply the force only to the particle itself. This would mean no GPU threads allocated would be wasted, and

load imbalance would be low due to the density assumption discussed earlier. However, the total number of arithmetic operations and bin lookups performed would be higher since we do a lot of repeat calculations during force application (once for the particle, and again for the neighbor) and have to look at 9 bins instead of just 5 bins as in the DP approach, so the implementation was ultimately slower, taking 3.50 s for 1M particles while our final implementation took 2 s for 1M particles. We believe that this is due to the fact that non-empty bins have a relatively similar amount of workload. However, there could be certain use cases (such as a sparse grid with large clusters of particles in a few bins) where the alternate approach could be faster.

4. Benchmarking

4.1. Log-Log plot of performance and Scaling

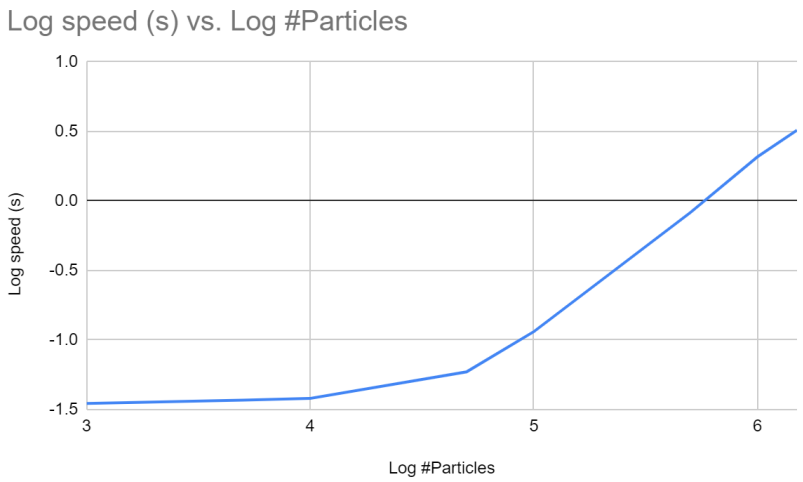

Fig 1: Log-log plot for GPU implementation (256 threads/block)

From the above plot, we can see that the relationship between speed and number of particles (in the log domain) seems to be almost exponential in nature due to the constant runtime initially, but the implementation seems to exhibit an almost linear trend starting from 100K particles. Specifically, when we go from 100K to 500K particles, execution time increases by a factor of 7x, which is relatively close to a linear increase in time of 5x.

4.2. Comparing GPU code performance to MPI, OMP and Serial Code

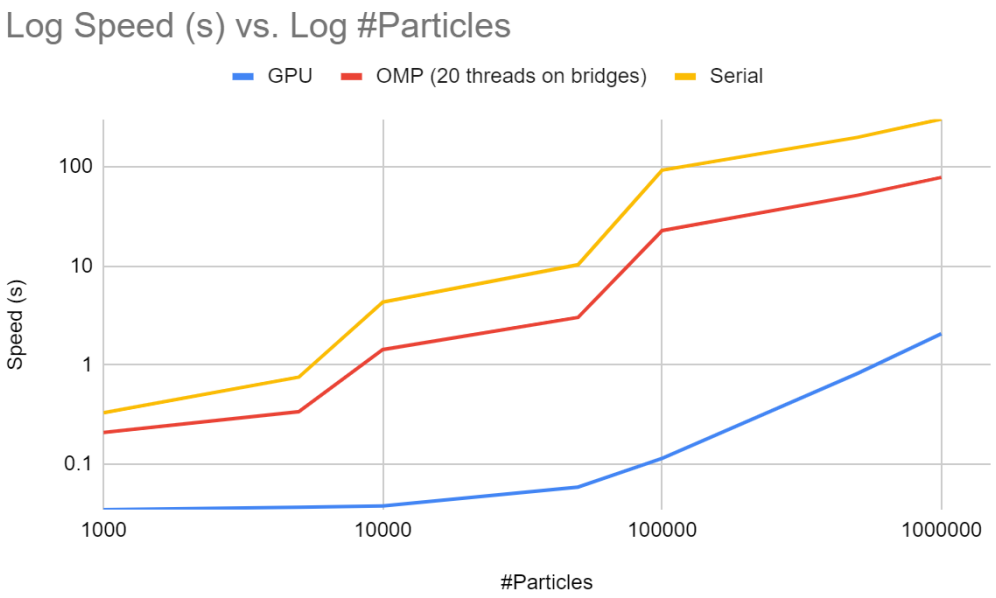

Fig 2: Comparison of different implementations

The above graph shows a comparison of GPU, OMP and Serial implementations of the problem. For the OMP implementation, we use 20, which is the maximum number of threads available on bridges as those settings give us the best performance. It is evident that the GPU implementation performs the best for all problem sizes, especially at ~1M particles where the difference is the most pronounced, i.e., we see most gains from GPU usage for larger problem sizes, which is what we expect. For smaller problem sizes, the overhead from processes such as memory allocation offset the speedup of GPU parallelism.

5. Code Profile

Using the nvprof profiler, we were able to look at a breakdown of execution time for each function in our implementation (i.e., moving particles, sorting, etc.), and further, which cuda API calls accounted for the largest chunk of execution time. For 100K particles, the split was as follows:

Function	**% of Time**
compute_forces_gpu	33.57
sort_parts	20.61

count_bins	18.98
move_gpu	15.85
Thrust operations	11.06

From the table above, we can see that the largest chunk of time is spent on computing forces between the particles, while moving the particles (move_gpu) takes about half the time as computing forces. The initial processing steps, i.e., counting the number of particles per bin (count_bins) and sorting the particles (sort_parts) using the prefix sum approach take about ~40% of the total time. In terms of API calls, around 65% of execution time was spent on cudaMalloc calls, and 25% on synchronize calls.

For 1M particles, the breakdown is as follows:

Function	**% of Time**
sort_parts	41.73
count_bins	32.12
compute_forces_gpu	15.15
move_gpu	7.58
Thrust operations	2.79

In this case, the preprocessing steps take the largest portion of time, accounting for about 73%, while computing forces and moving particles account for about 15% and 8% respectively. Moreover, synchronize calls now dominate, explaining about 71% of execution time, while cudaMalloc calls account for 19%.

6. Conclusion

With our GPU implementation, we see performance that is significantly better than both the OMP and the serial implementations. The GPU shines especially for large problem sizes as the initial setup overhead (particularly those related to allocating memory) partially offsets speedup gains when the number of particles is small.

6. Workload Distribution

AJ Kumar: GPU Bin-Parallel Implementation, Debugging, Report Writing
Sayan Das: GPU Bin/Particle-Parallel Implementation, Debugging, Report Writing
Varun Jadia: GPU Bin-Parallel Implementation, Profiling, Report Writing