# CS267 Homework 2.3: Parallelizing a Particle Simulation

Yifan Zheng, Brenda Damian, Ian Kolaja

March 18th, 2022

## 1 Introduction

The purpose of this assignment is to parallelize the same particle problem as in HW2 part 1 and 2 yet this time the parallelization is done with GPUs rather than with OpenMP or MPI. The goal of doing this is to obtain an $O(n)$ runtime. The GPUs implementation of this code aims to implement a solution that allows to maximize the features provided by GPUs, which are designed to maximize throughput and bandwidth.

## 2 Data structure

The main challenge associated with using CUDA is efficiently using the very limited memory of GPUs. Our previous implementation that used 2D vectors to store the set of particles associated with each bin would be too costly from a memory standpoint. Thus, we heavily leveraged the ideas provided in discussion to implement a lighter weight system for binning the particles [1]. This was done with several arrays. An integer array *bin_cnt_gpu* with a length num_bins was initialized, with each entry corresponding to the number of particles in that bin. This structure of the particles is done in a flat array, which is sorted in a grid square (Number of bins * Number of bins) in column-major order. A separate array is used to store the pointer to the index of our first appearance of a particle in a given bin in our grid square. This structure allows us to use the Thrust functions in order to rebuild our grid. An additional array with length num_bins+1 is initialized to store the prefix sum of *bin_cnt_gpu*.

### 2.2 Algorithm

When using CUDA, essentially each bin belongs to a thread. The algorithm leverages the massive parallel power of the GPU to rebuild the grid at every step rather than maintaining it. The ability to rebin each particle in each step would not have been as efficient when using OpenMP or MPI, since both solutions only recurred to synchronization between threads when it was absolutely necessary

(when a particle moved to another owner) to avoid the high costs of communication.
Between each of the following steps, cudaDeviceSynchronize function is called, which blocks the program flow until all of the GPU threads have finished executing the current function. For each bin, one GPU thread is used.

1. The *bin_cnt_gpu* array is initialized with all zeros.
2. The number of particles in each bin is determined and stored in *bin_cnt_gpu*. of where particles belong is done. The ID of the thread is calculated, the bin ID of the particle is determined, and then atomic addition is used to increase the count of particles in that bin.
3. A separate array stores the index where the first particle of each bin appeared in our grid.
4. By using the *exclusive_scan* function from thrust CUDA library on a copy of *bin_cnt_gpu*, an array containing the prefix sum of the bin count is computed and stored separately. In other words, the cumulative sum of the number of particles in a given bin and all bins before it is stored.

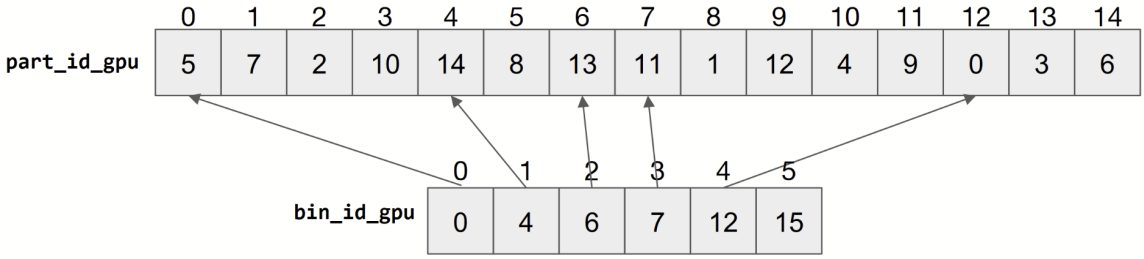


Figure 1. Diagram of arrays used for calculation. The array *part_id_gpu* stores particle IDs in the same bin in a contiguous manner within the array, while *bin_id_gpu* indicates the index of that array where each bin starts [1].

After initialization, the main loop of the code begins for computing forces and moving particles.

1. The forces are calculated in GPU as well as the forces of the neighboring bins. In order to do this calculation, given a bin id, the starting and ending point of a bin's particles in our grid is computed.

2. The particles are moved, and it is determined if they've entered a new bin. If so, the corresponding initial bin's particle count is reduced by one, and the new bin's count is increased. This is done using atomic operations to prevent the overlapping of threads.
3. The particles are rebinned in the same way they are during initialization, utilizing the *exclusive_scan* function and filling bin_id_gpu

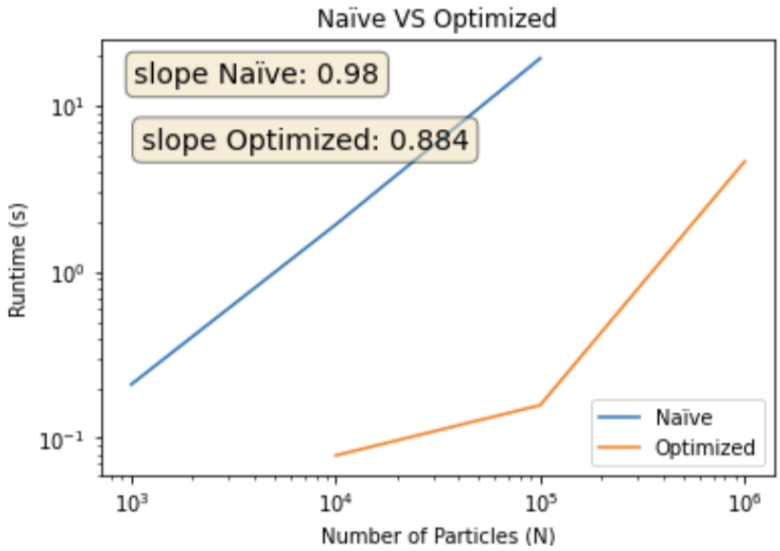


Figure 2. Naive implementation vs our optimized implementation.

Figure 2 represents a side by side comparison of the code given versus the implementation our team achieved. The naïve implementation performed better than our OpenMP code, but the GPU needs to be used wisely to exploit its full potential. The naïve code could not handle more than $10^6$ particles with the same resources our optimized code used. Since our kernels did not exploit data sharing between threads, we did not expect to have better performance when the threads per bin were increased. Therefore, the number of threads in each block was kept at 256 for both solutions.

The slope reveals that it is very close to a linear representation. The results display a constant scale almost achieving a $O(n)$ scaling. From this graph we can

observe that it is still not completely optimized for all the number of particles it as its still showing deeps specially in $10^5$ number of particles.

## 2.3 Design choices

The design choices allowed us to make the bin count in place and allowed us to remain having the calculation of the forces parallelized. The design choices also allowed us to do the calculations with constant time. It is important to ask why should we insert a function that requires $O(n \log n)$ in a solution that requires only $O(n)$ but the answer is simple, the alternative would be more costly. The design decision to keep the particles stored sorted was extremely important to reduce the search complexity of the problem, if the particle storage would not have been sorted, each time we would have to search each with a time complexity of $O(\log n)$, while the sort requires only $O(n \log n)$ twice.

Additionally, we changed our function for finding adjacent bins by removing the conditional statements previously used. A possible pitfall with GPU programming lies in wasting computation time waiting for conditionals to evaluate in other threads while being blocked by synchronization.

## 2.4 Time

The time spent in performing the implementation is not divided equally. When n is small there is more time spent communicating and synchronizing than it is computing. However when n is large, the nonlinearity in computation dominates.

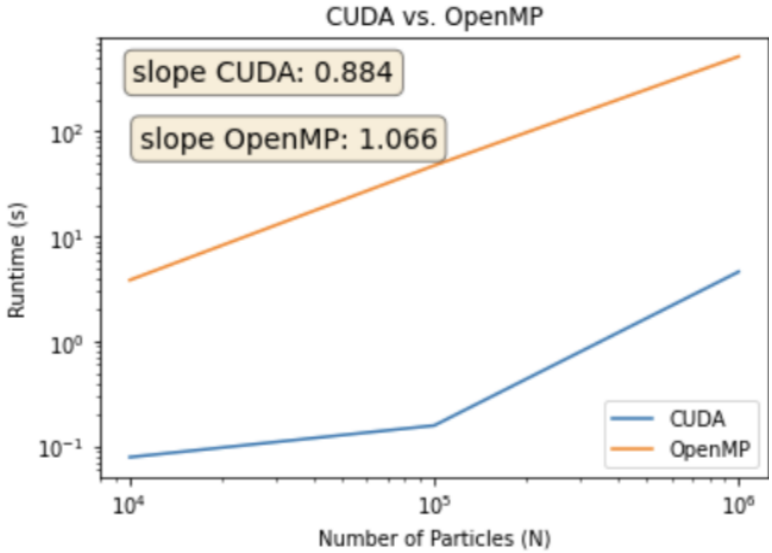

Figure 3. CUDA implementation vs OpenMP.

# 3 Comparison with other parallel computing techniques

Figure 3 shows a comparison of our solution using OpenMP and the CUDA implementation. The results, as shown in the graph, show a significant speedup. The speedup that is obtained with CUDA represents a single point at 1 task, which is noticeably higher than the solution implemented with OpenMP, while the implementation of OpenMP achieves a slope of $O(n + \varepsilon)$.

# 4 Conclusions

Our CUDA implementation achieved the best performance of all of our particle simulations thus far. The CUDA model is a SPMT (Single Program, Multiple Threads) that requires the data to have a grid structure [2] and has strengths in handling instructions in threads. OpenMP and MPI are both SIMD models (Single Instruction, Multiple Data) which can allow the multiple autonomous processors to execute simultaneously.

Even though CUDA achieved a better performance in the optimized implementation, it exceeded the time limit in the naïve implementation at $10^7$. The same results were obtained in the optimized implementation of MPI, meaning that an optimized version of the code that aimed to solve the same problem had the same results of the naïve implementation in CUDA.

Each component of homework 2 gave us an understanding of different parallelization methods and helped us identify each of their weaknesses and strengths. Subjectively, CUDA proved to be the easiest for our group to implement because there is no need to consider what combination of serial and parallel execution is best. However, certain problems are less suitable for GPU programming. Programs that rely on many conditionals risk wasting substantial computation time due to the nature of CUDA synchronization. Problems that require a lot of communication between the CPU and GPU can also discourage the use of CUDA, as MPI communication could end up being cheaper. CUDA also faces the challenge of having a vendor lock with NVIDIA. Meanwhile, OpenMP might be preferred to MPI for parallelizable problems that don't require leveraging the sheer scalability of clusters.

There is great potential for utilizing both MPI and CUDA together; MPI allows for communication between processors (notably in cluster environments), where CUDA can then be used to leverage NVIDIA GPUs when available to accelerate calculations further. For future work, we can explore the different possibilities of mixing methods.

# 6 Division of Work

The code conceptualization process was highly collaborative for our team. Each of us attempted to implement our own version of the serial and parallel code, but ultimately Yifan's version of the code served as the basis of our report. We all worked together on brainstorming and implementing solutions to specific bugs, as well as finding ways of improving the performance together in group calls and by attending office hours. Each of us wrote some of our own sections of the report and had unique roles in putting it together.

# 7 References

[1] G. Wang, "CS 267 Lab 2.3 CUDA/GPU Particle Simulation", March 2022.
[2] "Cuda Refresher: The Cuda Programming Model." *NVIDIA Technical Blog*, 25 Aug. 2020,
https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/.