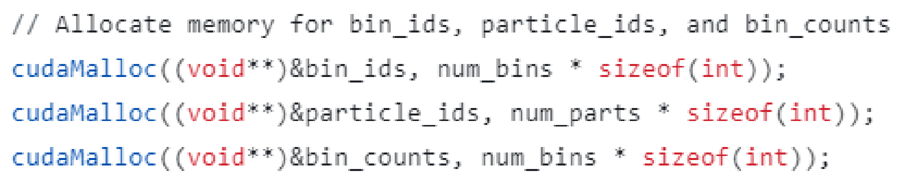
# Homework 2.3

CS 267
3/17/2024
Giovanni Battista Alteri, Jeffy Jeffy, Elizabeth Gilson
Group 36

**Introduction:**
The assignment is an extension of the particle simulation performed in Assignment 2.1. The previous assignment was a naive implementation of a looping algorithm which calculates the force on particles in a closed system. Based on the force and acceleration of each particle, the particle moves within the box, and interacts with each other. The number of particles is constant, and so is the volume of the system (defined by the variable “size”, given by the square root of the total number of particles). We have chosen a density sufficiently low so that we will have *O(n)* interactions with *n* particles. This means that we could achieve the same level of accuracy of the naive code even if we consider only the interactions among particles that are sufficiently close to each other.

**Parallel Implementation using CUDA**
For this homework we aim to parallelize the code by targeting Perlmutter’s A100 GPUs. First, we attempted to implement the OpenMP solution into CUDA, however the arrays that we used in our OpenMP solution were too large for the GPUs. After facing this limitation, we opted to refactor the code to leverage CUDA's parallel computing capabilities efficiently. We also tried utilizing linkedList as a data structure since they produce results. However, due to the ambiguity presented by using a linked list, it became much harder to implement and debug. Additionally, due to its dynamic nature, it is also harder to perform shared memory implementations of the program.

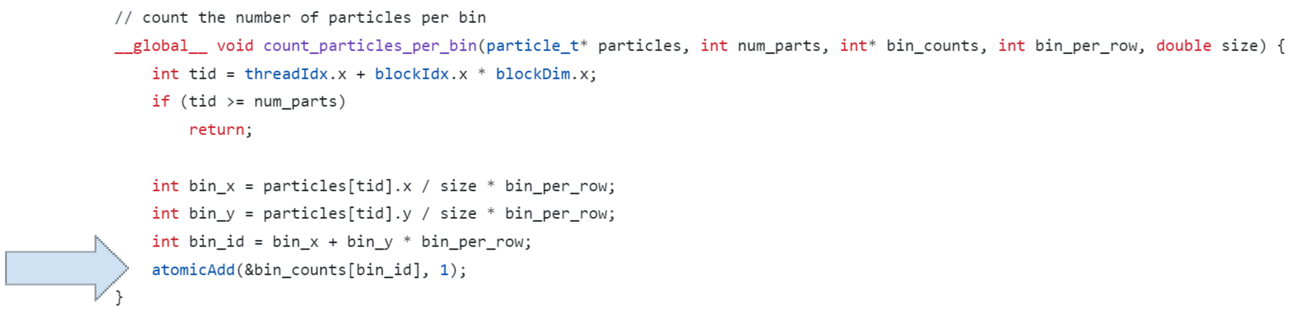
*Parallelization with CUDA Kernel*s: CUDA allows executing kernels (functions that run on the GPU) in parallel on multiple threads. In this code, both *compute_forces_gpu* and *move_gpu* kernels are invoked with multiple threads, where each thread handles a different particle or a group of particles. CUDA enables concurrent execution of kernels across multiple threads, which is essential for handling large numbers of particles efficiently. This parallel execution significantly speeds up the computation compared to sequential execution on a CPU.


```
// Allocate memory for bin_ids, particle_ids, and bin_counts
cudaMalloc((void**)&bin_ids, num_bins * sizeof(int));
cudaMalloc((void**)&particle_ids, num_parts * sizeof(int));
cudaMalloc((void**)&bin_counts, num_bins * sizeof(int));
```

*Memory Allocation Optimization*: Memory allocation for bin_ids, particle_ids, and bin_counts is done on the GPU using cudaMalloc. This ensures that memory is directly accessible by GPU kernels, eliminating the need for data transfers between the CPU and GPU during simulation execution.

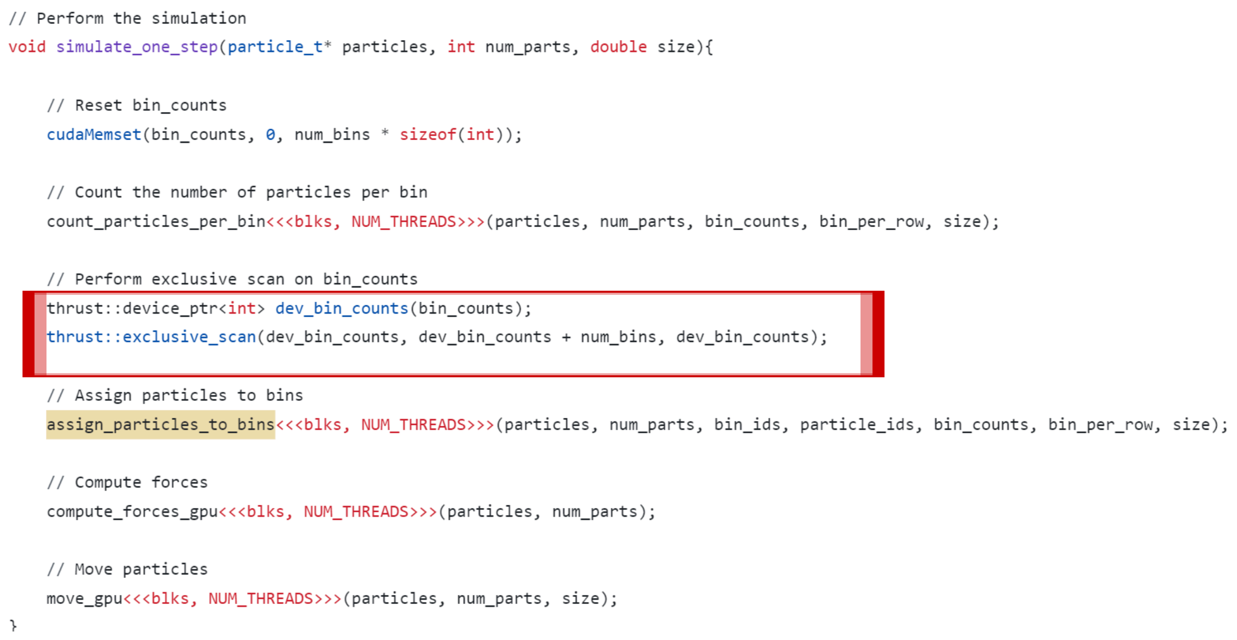
```
#define NUM_THREADS 256
```

*Thread and Block Management*: The simulation distributes the workload across multiple CUDA threads and blocks. The number of threads per block is defined as NUM_THREADS, and the number of blocks is calculated based on the total number of particles (num_parts). This optimization ensures that the GPU's resources are effectively utilized, and the workload is evenly distributed across available CUDA cores.



```
// count the number of particles per bin
__global__ void count_particles_per_bin(particle_t* particles, int num_parts, int* bin_counts, int bin_per_row, double size) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid >= num_parts)
        return;

    int bin_x = particles[tid].x / size * bin_per_row;
    int bin_y = particles[tid].y / size * bin_per_row;
    int bin_id = bin_x + bin_y * bin_per_row;
    atomicAdd(&bin_counts[bin_id], 1);
}
```

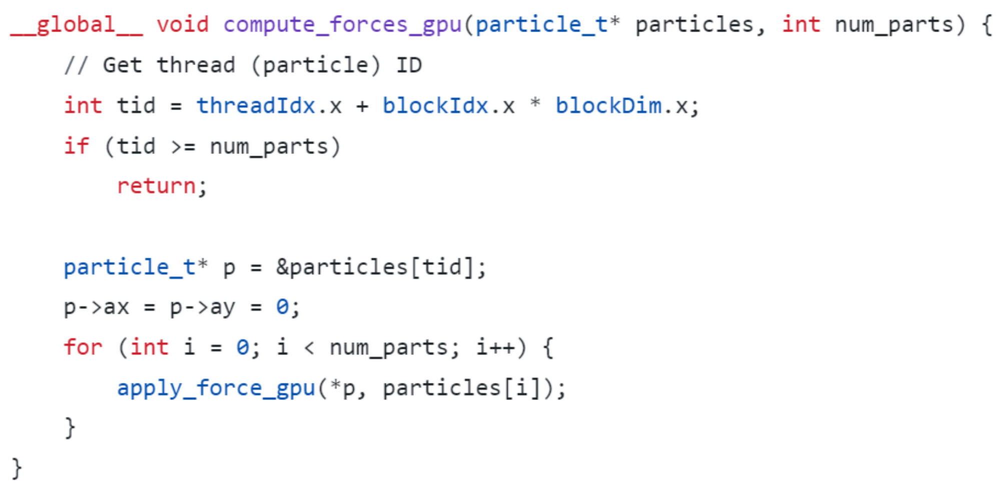
*Atomic Operations*: Atomic operations like atomicAdd are used to safely update shared memory locations accessed by multiple threads concurrently. In the assign_particles_to_bins kernel, atomic addition is employed to update bin_counts and assign particle indices to appropriate bins without race conditions.



```
// Perform the simulation
void simulate_one_step(particle_t* particles, int num_parts, double size){

    // Reset bin_counts
    cudaMemset(bin_counts, 0, num_bins * sizeof(int));

    // Count the number of particles per bin
    count_particles_per_bin<<<blks, NUM_THREADS>>>(particles, num_parts, bin_counts, bin_per_row, size);

    // Perform exclusive scan on bin_counts
    thrust::device_ptr<int> dev_bin_counts(bin_counts);
    thrust::exclusive_scan(dev_bin_counts, dev_bin_counts + num_bins, dev_bin_counts);

    // Assign particles to bins
    assign_particles_to_bins<<<blks, NUM_THREADS>>>(particles, num_parts, bin_ids, particle_ids, bin_counts, bin_per_row, size);

    // Compute forces
    compute_forces_gpu<<<blks, NUM_THREADS>>>(particles, num_parts);

    // Move particles
    move_gpu<<<blks, NUM_THREADS>>>(particles, num_parts, size);
}
```

*Thrust Library*: The Thrust library is utilized for performing exclusive scan operations (thrust::exclusive_scan). Exclusive scan is applied to bin_counts after counting the number of particles per bin. This operation is essential for calculating the starting index of particles within each bin, enabling efficient access during force computation.

*Kernel Fusion*: Kernel fusion is employed by combining multiple computational tasks into a single kernel invocation. In the simulate_one_step function, multiple CUDA kernels (count_particles_per_bin, thrust::exclusive_scan, and assign_particles_to_bins) are executed sequentially within the same CUDA kernel invocation. This reduces kernel launch overhead and improves computational efficiency.



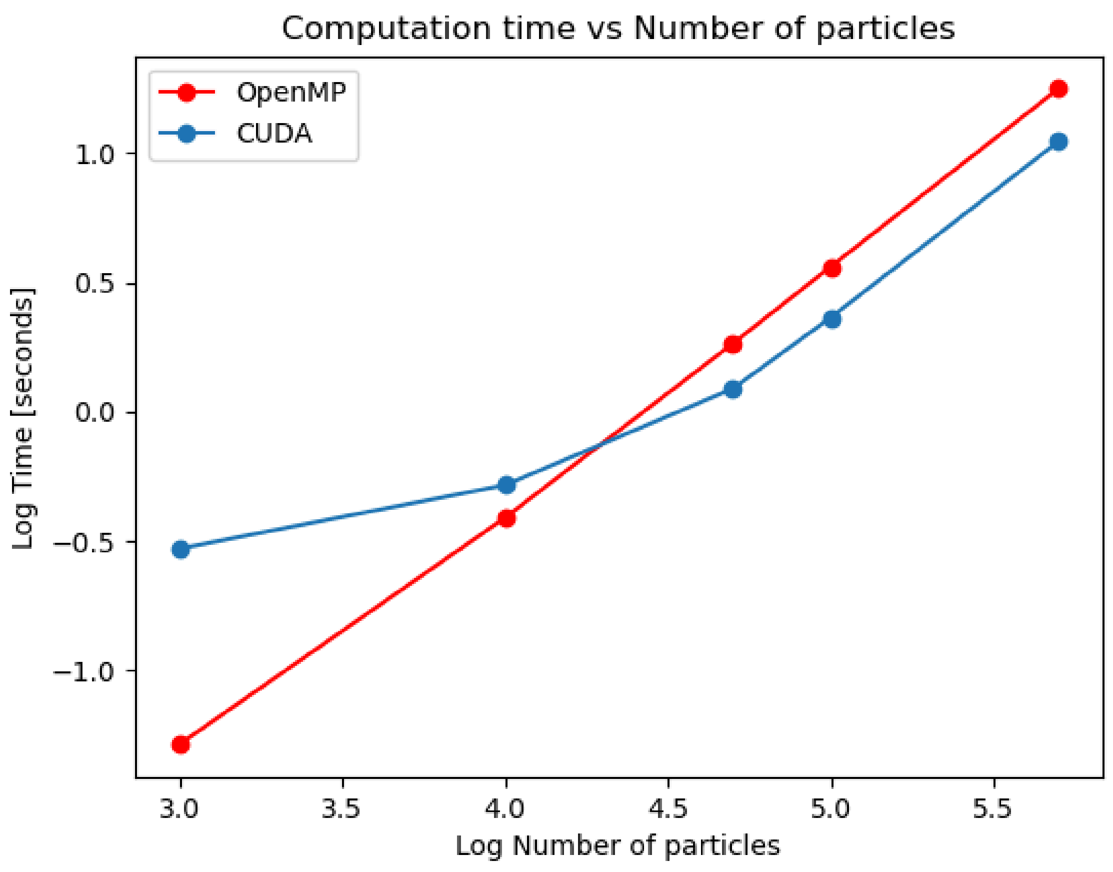
```
__global__ void compute_forces_gpu(particle_t* particles, int num_parts) {
    // Get thread (particle) ID
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid >= num_parts)
        return;

    particle_t* p = &particles[tid];
    p->ax = p->ay = 0;
    for (int i = 0; i < num_parts; i++) {
        apply_force_gpu(*p, particles[i]);
    }
}
```

*Parallelization of Particle Interactions*: In the compute_forces_gpu kernel, each thread is responsible for computing the forces acting on a single particle due to interactions with its neighboring particles. By assigning a unique thread to handle each particle, the computation of forces between particles is parallelized. This means that multiple particles can be processed simultaneously by different threads on the GPU. For example, if there are N particles in the simulation, N threads will be launched, each responsible for computing the forces acting on a specific particle. This parallel approach enables the GPU to exploit its massive parallelism efficiently, as thousands of threads can execute concurrently on the GPU cores. As a result, the computation of forces between particles can be completed much faster compared to a sequential implementation, especially when dealing with a large number of particles.

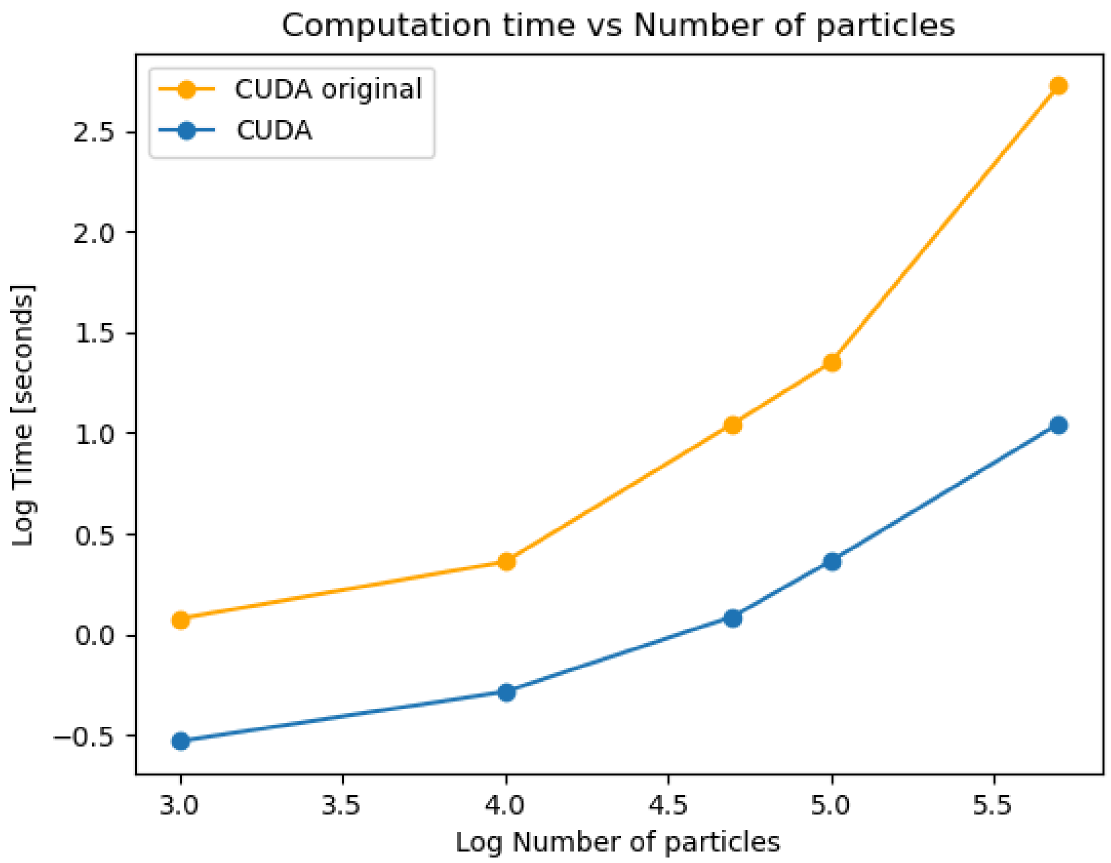
```
__global__ void move_gpu(particle_t* particles, int num_parts, double size) {

    // Get thread (particle) ID
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid >= num_parts)
        return;

    particle_t* p = &particles[tid];
    //
    //  slightly simplified Velocity Verlet integration
    //  conserves energy better than explicit Euler method
    //
    p->vx += p->ax * dt;
    p->vy += p->ay * dt;
    p->x += p->vx * dt;
    p->y += p->vy * dt;

    //
    //  bounce from walls
    //
    while (p->x < 0 || p->x > size) {
        p->x = p->x < 0 ? -(p->x) : 2 * size - p->x;
        p->vx = -(p->vx);
    }
    while (p->y < 0 || p->y > size) {
        p->y = p->y < 0 ? -(p->y) : 2 * size - p->y;
        p->vy = -(p->vy);
    }
}
```

*Parallelization of Particle Movement:* Similarly, in the move_gpu kernel, each thread is assigned to update the position of a single particle based on its velocity and acceleration. This process is also parallelized, allowing for efficient computation of particle movement across multiple particles simultaneously. Just like with particle interactions, if there are N particles in the simulation, N threads will be launched in the move_gpu kernel. Each thread will independently update the position of its assigned particle based on the particle's velocity and acceleration, without relying on the computation of other particles. This parallel approach enables the movement computation of multiple particles to be performed concurrently, leveraging the GPU's parallel architecture to achieve significant speedup. By parallelizing both particle interactions and movement, the simulation can fully utilize the computational power of the GPU, resulting in faster and more efficient simulations, especially when dealing with large-scale particle systems.

**Results**



Above is a graph showing the log of the time in seconds vs the log of the number of particles for our CUDA and OpenMP implementations of the particle simulation. As you can see for smaller numbers of particles the OpenMP implementation is faster, this is likely due to the slight overhead in parallelization using CUDA. CUDA is effective for large workloads, but there is a performance penalty for smaller-scale computations. This is confirmed because for larger numbers of particles, around 20,000 and higher, the CUDA solution is faster than the OpenMP implementation.



In this graph we have the log of time in seconds vs the log in the number of particles for both the original naive CUDA implementation and our implementation. As you can see, for any number of particles, our implementation is faster and this difference only increases with the number of particles.

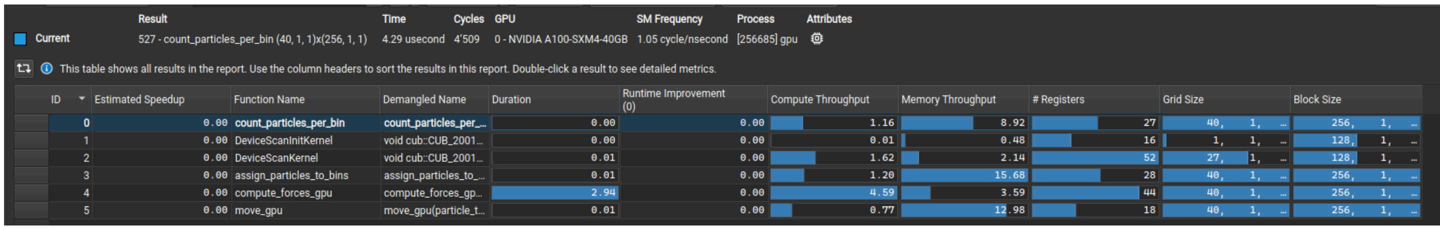
**Time Components**

To study the different components of the total computational time, we used a modified version of the job-gpu file that employs NVIDIA Nsight (from the CUDA Toolkits). This is the code used:

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -C gpu
#SBATCH -A mp309
#SBATCH -t 00:30:00
#SBATCH --gres=gpu:1
```

```
srun --ntasks-per-node 1 dcgmi profile --pause
size=(2000)
output_file="profile${size}.ncu-rep"
srun ncu --kernel-id :::1 -o "${output_file}" --call-stack ./gpu -s 1 -n
"${size}"
srun --ntasks-per-node 1 dcgmi profile --resume
```

The toolkit generates an output file that can be opened using a Desktop version of NVIDIA NSight Compute; this is a typical result:

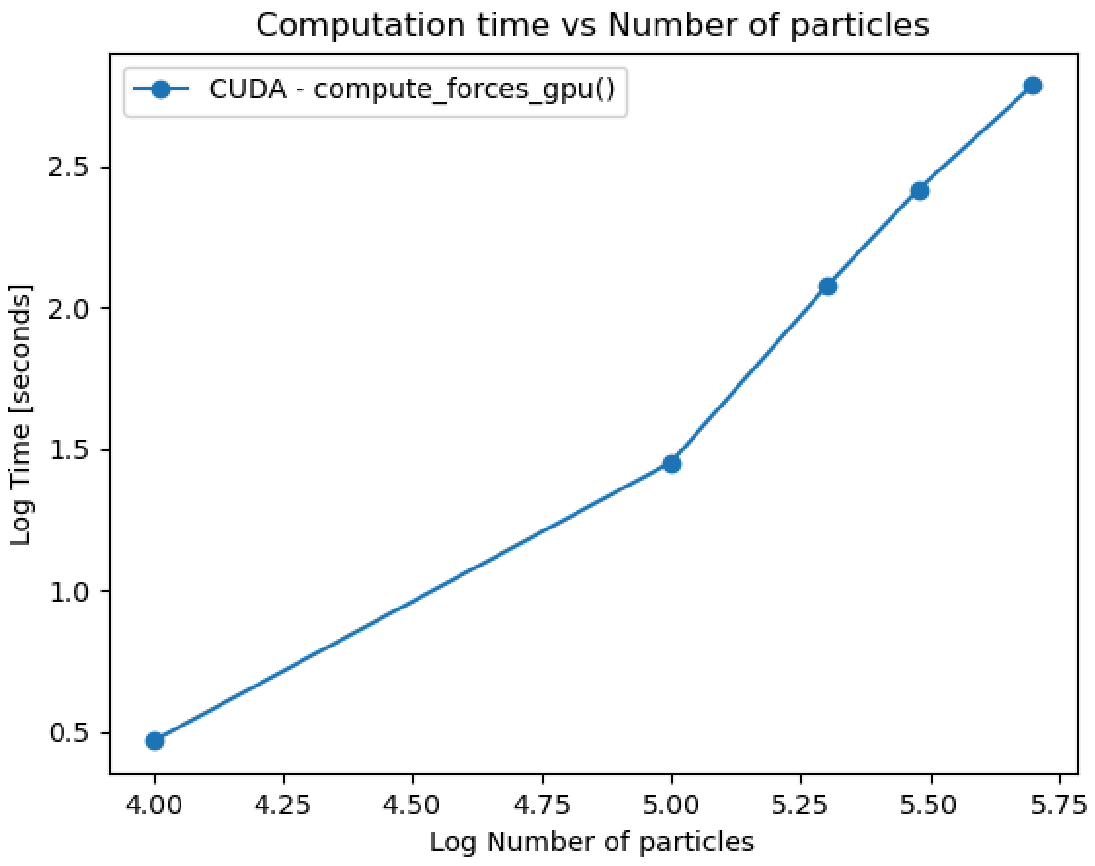


| | Result | Time | Cycles | GPU | SM Frequency | Process | Attributes |
|---|---|---|---|---|---|---|---|
| Current | 527 - count_particles_per_bin (40, 1, 1)x(256, 1, 1) | 4.29 usecond | 4'509 | 0 - NVIDIA A100-SXM4-40GB | 1.05 cycle/nsecond | [256685] gpu | |

This table shows all results in the report. Use the column headers to sort the results in this report. Double-click a result to see detailed metrics.

| ID | Estimated Speedup | Function Name | Demangled Name | Duration | Runtime Improvement (0) | Compute Throughput | Memory Throughput | # Registers | Grid Size | Block Size |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.00 | count_particles_per_bin | count_particles_per_... | 0.00 | 0.00 | 1.16 | 8.92 | 27 | 40, 1, ... | 256, 1, ... |
| 1 | 0.00 | DeviceScanInitKernel | void cub::CUB_2001... | 0.00 | 0.00 | 0.01 | 0.48 | 16 | 1, 1, ... | 128, 1, ... |
| 2 | 0.00 | DeviceScanKernel | void cub::CUB_2001... | 0.01 | 0.00 | 1.62 | 2.14 | 52 | 27, 1, ... | 128, 1, ... |
| 3 | 0.00 | assign_particles_to_bins | assign_particles_to_... | 0.01 | 0.00 | 1.20 | 15.68 | 28 | 40, 1, ... | 256, 1, ... |
| 4 | 0.00 | compute_forces_gpu | compute_forces_gp... | 2.94 | 0.00 | 4.59 | 3.59 | 44 | 40, 1, ... | 256, 1, ... |
| 5 | 0.00 | move_gpu | move_gpu(particle_t... | 0.01 | 0.00 | 0.77 | 12.98 | 18 | 40, 1, ... | 256, 1, ... |

We can visualize the total computational time spent in the CUDA section of the code, and we can highlight the time required by each different function speeded-up using CUDA. For example, in this result we can see that the compute_forces_gpu method took the longest amount of time and memory. This trend is repeated for all the experiments we tried, using different numbers of particles (10000, 100000, 200000, 300000, 500000).



**Can we improve this code?**
There is a lot of room for improvement within this code. Some of the considerations that we wanted to include but were not able to successfully implement are as follows:

*Implementation of linked list:* We tried creating a ListNode struct where each node holds the index of a particle in the 'particles_t* particles array. The general function of the linked list is very similar to the function of the array in the code provided above. In this implementation of the code, once the linked list are initialized, the kernel inserts particles indices into the appropriate linked list based on their bin ids

*Updating and modifying compute_forces_gpu:* In order for the compute_forces_gpu function to be truly optimized, the function must traverse through the data structure of choice to find the neighboring bins and compute the forces between the particles. In our current code. In our code, this functionality can be improved and can provide faster and better results.

*Addition of padding:* In a grid-based approach for spatial partitioning, padding is appropriate to handle particles that are near the boundary of a bin. This is important because particles near the edge of a bin may actually overlap with neighboring bins. Without padding, these particles would be assigned to incorrect neighboring bins.

We believe implementing these improvements can strongly improve our code and the speed of the particle simulation.

**Contributions**
Jeffy- helped work on the code, optimize the code, helped with data collection and helped with the write-up
Giovanni- helped work with the timings, on editing and testing the code, creating the graphs and helped with the write-up
Elizabeth- helped with the write-up, on editing the code, and with the data for timings