CS C267 Homework 2.3 Report

Maya Lemmon-Kishi, Zach Van Hyfte, and Minan Wu

Homework Group 63

1 Overview

In this report, we discuss the strategies we used to optimize a C/C++ particle simulation program for execution on a GPU using CUDA. In Section 2, we discuss the design choices behind our implementation of the parallel particle simulation. When run on one of Bridges-2's shared NVIDIA Tesla V100 GPU nodes, our CUDA implementation completes a $1,000,000$–particle simulation in ~ 2.6 seconds. In Section 3, we analyze the performance of our CUDA implementation and compare it to that of our previous OpenMP particle simulation implementation.

2 Parallel Algorithm Implementation with CUDA

We chose to begin our CUDA implementation of the particle simulation program by modeling our OpenMP implementation from Homework 2.1 as closely as possible, as the underlying architecture and parallelism model resembles OpenMP more so than MPI. However, to adapt our implementation to the strengths and limitations of the GPU architecture, we made significant changes to the data structures used to track the contents of each particle bin, inspired largely by what was suggested in the recitation. We then implemented and evaluated further optimizations tailored specifically for the GPU architecture.

2.1 Simulation Grid Space and Symmetric Forces

Following in the footsteps of our earlier OpenMP implementation, our optimized CUDA implementation divides the simulation space into a single-layer grid of square bins, with the length and width of each bin equal to the cutoff radius (r), such that `bin_width` $= r$. As before, we initially selected r as our bin width because it minimizes the area of the neighboring bins that must be examined to find all of the particles that could be exerting a force on a particle within a given bin, such that each thread has to examine fewer nearby particles.

Let d be the particle density of the simulation, and let n be equal to the number of particles in the simulation. The width and height of the simulation space, s, is thus equal to $\sqrt{dn}$. In a square grid with area s, comprised of blocks of width r, each row l of the grid contains $\lceil \frac{s}{r} \rceil$ blocks, for a total of l^2 blocks in the entire grid. The block number in which a particle represented by a `particle_t` object named `part` is located is equal $(\lfloor \frac{\texttt{part.x}}{r} \rfloor * l) + \lfloor \frac{\texttt{part.y}}{r} \rfloor$.

We modified the function `init_simulation` to first calculate the size of each block and the number of blocks needed to cover the whole simulation space, based on the size and density of the simulation. However, unlike the `init_simulation` function in our OpenMP implementation, our CUDA implementation's `init_simulation` function it does not calculate and store the indices of each bin's "neighbor bins" for later use — we explain this design choice in Section 2.4.1.

Again modeling after our OpenMP implementation, we modified the provided `apply_force_gpu` function to update the acceleration values of both of the input particles at a time, rather than just the first of the input particles. This can be done efficiently because, for any pair of particles (i, j), the repulsive force particles (i, j) exerted on particle j by particle i has the same magnitude as the repulsive force exerted on particle i by particle j, but acts in the opposite direction. Thus, for every pair of particles, the repulsive force is calculated just once instead of twice, and applied (with different signs) to both particles within a single call

to `apply_force_gpu`. We made a corresponding modification to our implementation of `simulate_one_step` such that it calls `apply_force` once for every *combination* of two particles, rather than once for every *permutation* of two particles. As in our OpenMP implementation, this optimization introduces a potential data race that requires additional synchronization logic, since two threads executing `apply_force` can potentially both be trying to update the same particle's acceleration values. We addressed this potential data race in the synchronization logic that we added to our CUDA implementation, as described in Section 2.3.

In our optimized implementation of `compute_forces_gpu`, each thread is still mapped to a single particle (such that the thread with thread ID `tid` is mapped to `parts[tid]`). This implicitly exposes the thousand-plus-way parallelism needed to take full advantage of modern manycore GPUs. Each thread first examines the X– and Y-position of `parts[tid]` to calculate the index b of the bin in which it is located. Then, referencing the `part_ids` data structure described in Section 2.2, it iterates through all of the other particles in bin b that have a lower index in `part_ids` than `parts[tid]` does, and calls `apply_force_gpu` on `parts[tid]` and each of those other particles. This practice of only iterating through particles with a lower index in `part_ids` ensures that `apply_force_gpu` is called on every pair of particles in bin b only once, as is necessary for correctness when our symmetric force optimization is used.

Next, our `compute_forces_gpu` implementation then calculates (on the fly) the indices of all of the "neighbor" bins of bin b. For each such neighbor bin, it again uses the `part_ids` data structure to iterate through every particle p in that neighbor bin, calling `apply_force_gpu`.

As in our OpenMP implementation, consolidating the acceleration updates for each pair of nearby particles into a single call to `apply_force_gpu` also allowed us to reduce the number of neighbor bins that need to be examined. Instead of examining all 8 of the bins adjacent to its particle's bin b, each thread only needs to examine the bottom center, top right, middle right, and bottom right neighbor bins — 4 neighbor bins in total. We able able to compute all possible force interactions considering just four neighboring blocks since forces are calculated symmetrically. Figure 1 shows how this works. If our implementation were to have each thread examine the full set of 8 bins, additional computation would be required to maintain a data structure tracking which pairs of particles have already had `apply_force_gpu` called on them — this data structure would have to be queried or updated by each thread upon examining each permutation of two particles.

2.2 Particle Re-Binning and Bin Data Structure

Our OpenMP implementation used a two-dimensional vector — a data structure of type `std::vector<std::vector<int>>` — to keep track of the particles currently located in each bin. The outer vector contained l^2 vectors, one for each block. Each of the l^2 inner vectors stored the IDs of the particles currently located within that bin. This allowed threads to quickly find and iterate through all of the particles located within a specific bin. Re-binning was performed by

However, as explained during the recitation, `std::vector` objects don't work in the device-side code that runs on the GPU, and other data structures that are similar in functionality (e.g. `thrust::device_vector` objects) require frequent (and computationally expensive) data copying. So, following the main suggestions from the recitation, we modified our bin data structure and re-binning procedure to use a standard one-dimensional `int*` array. This array, called `part_ids`, is of length `num_parts` and is located in the GPU's device memory. As recommended in the recitation, `part_ids` contains the IDs of all of the particles in the simulation, sorted by their bin ID. As an example, assume that at a given simulation step t, there are n_0 particles in the bin with ID 0, n_1 particles in the bin with ID 1, and so on. At simulation step t, then, `parts[0]` through `parts[`$n_0 - 1$`]` will contain all of the particles in bin 0, `parts[`n_0`]` through `parts[`$n_0 + n_1 - 1$`]` will contain all of the particles in bin 1, and so on.

In addition to `part_ids`, our CUDA implementation maintains three "supporting" arrays that are used to assemble, and locate specific bins within, `part_ids`. Each of these supporting arrays is of type `int*`, is located in the GPU's device memory, and contains `num_bins` entries:

- The array `bin_counts` keeps track of how many particles are located in each bin; `bin_counts[i]` is equal to the number of particles located in the bin with ID i during the current simulation step. (To continue the example from above, `bin_counts[0]` would be n_0, `bin_counts[1]` would be n_1, and so on).

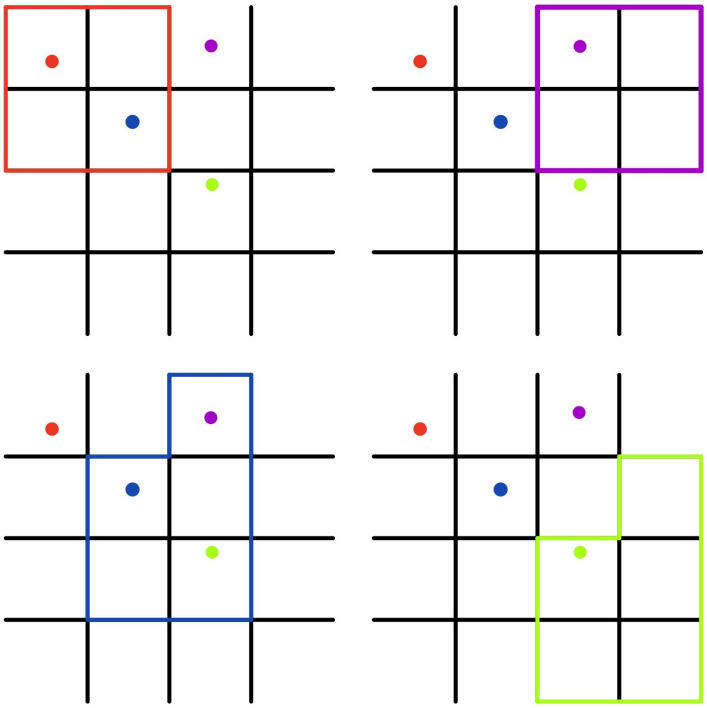

Figure 1: When symmetric forces on two particles are applied during a single call to `apply_force`, a maximum of five blocks must be searched to find all other particles that may be exerting a force on any given particle. The colored blocks represent the blocks searched to find all other particles that may be exerting a force on the particle with the matching color. This implementation saves the assertion of whether a certain pair of particles are already computed.

- The array `bin_start_indices` keeps track of where the contiguous region of `part_ids` corresponding to each bin begins; `bin_start_indices[i]` is equal to the index in `part_ids` of the first particle ID belonging to bin i. (To continue the example from above, `bin_start_indices[0]` would be 0, `bin_start_indices[1]` would be n_0, `bin_start_indices[2]` would be $n_0 + n_1$, and so on).
- The array `bin_next_indices` is used during re-binning, when the bin start and end indices are recalculated and `part_ids` is re-filled from scratch; `bin_next_indices[i]` contains the "bin tail pointer" for bin i — the index in `part_ids` of the first slot in bin i's region that hasn't yet been filled by a particle.

Although we store `part_ids` and its three supporting vectors on the GPU, we keep the pointers to those memory regions on the CPU side and pass those pointers as parameters to the CUDA kernels that use them. We chose to pass the pointers as parameters rather than maintain both CPU-side and GPU-side pointers for each array because it keeps our implementation simple and easy to debug if necessary, and (according to our GSIs) the parameter-passing doesn't incur significant overhead. We employed a similar parameter-passing strategy for frequently used constants like `num_bins` (which is equal to the total number of bins into which the simulation is divided).

At the end of each simulation step — once the forces on all of the particles have been calculated and applied by `compute_forces_gpu`, and all of the particles have thereafter been moved to their new locations by `move_gpu` — the beginning and end points of each bin's region of `part_ids` are recalculated based on the current bin sizes, and `part_ids` is re-filled from scratch; this constitutes the particle re-binning process. The re-filling of `part_ids` happens in five steps:

1. A call to `cudaMemset` zeroes out the `bin_counts` array, resetting the current number of particles in

each bin to 0.

2. A kernel called `count_particles_gpu` is called. As with `compute_forces_gpu`, each thread executing `count_particle_gpu` is mapped to just one particle. All that each thread does in this kernel is calculate the index i of the bin that its assigned particle is currently located in, and then increment `bin_counts[`i`]`. At the end of this step, `bin_counts` contains up-to-date information about the number of particles located in each bin that (i.e. information that reflects the particle movements that occurred during the current simulation step). During the execution of this kernel, it is possible for two threads to simultaneously try to update the same entry of `bin_counts` if their particles are both located in the same bin — we address this potential data race in Section 2.3.

3. An exclusive prefix scan is performed on `bin_counts` to generate an up-to-date copy of `bin_start_indices`. Our implementation uses the Thrust library — specifically, the function `thrust::exclusive_scan` — to perform this prefix scan.

4. A call to `cudaMemcpy` overwrites `bin_next_indices`, such that it is identical to `bin_start_indices` (because our implementation is re-filling all of the individual regions of `part_ids` from the start again).

5. Another kernel, `arrange_part_ids_gpu`, is called. Again, each thread executing this kernel is responsible for a single particle. Each thread: (a) calculates the index of the current bin of its assigned particle, retrieves the current value t of the `bin_next_indices` entry ("tail pointer") for that bin's region of `part_ids`, increments the tail pointer by 1, and then sets `part_ids[`t`]` to `tid` (which is the ID of its assigned particle). Very much like `count_particles_gpu`, it is possible for for two threads whose particles are located in the same bin to simultaneously update the same value — in this case, an entry of `bin_next_indices` — we address this potential data race in Section 2.3.

In the next simulation step, each thread executing the function `compute_forces_gpu` uses the `bin_start_indices` array to locate both the beginning and end points of the region of `part_ids` corresponding to its particle's bin b, and both the beginning and end points of the regions corresponding to bin b's neighbor bins.

2.3 Synchronization

As referenced above, there are three scenarios under which data races can potentially occur:

- Two threads executing `apply_force` can potentially both try to update the same particle's acceleration values at the same time, such that one update may erase and overwrite the other.
- Two threads executing `compute_forces_gpu` — whose particles are located in the same bin b — can potentially both try to increment `bin_counts[`b`]` at the same time, such that one update may erase and overwrite the other, such that the final value of `bin_counts[`b`]` is too low.
- Two threads executing `arrange_part_ids_gpu` — whose particles are located in the same bin b — can potentially both try to increment `bin_next_indices[`b`]` at the same time, such that one update may erase and overwrite the other, such they both try to write their respective particles' IDs to the same slot in `part_ids`.

Each of these data races involves two additions or subtractions potentially taking place at the same time, so we were able to completely eliminate all of these potential data races by simply replacing a total of 6 standard `+` and `-` statements with calls to CUDA's `atomicAdd` function. Using this function ensures that all of the relevant addition and subtraction operations take place atomically — the operations comprising one thread's addition or subtraction are not interleaved with those of other threads' additions or subtractions, eliminating the potential for concurrent increments or decrements that could potentially overwrite one another.

2.4 Additional Optimizations

2.4.1 Bin Index Caching

In the basic version of our implementation described above, during every simulation step, the bin index for each particle is calculated three separate times in three separate kernels — first by `count_particles_gpu` when calculating the number of particles in each bin, then again by `arrange_part_ids_gpu` when determining where in `parts_id` to place each particle, and finally by `compute_forces_gpu` at the start of the next simulation step. In the final version of our code, each thread executing `count_particles_gpu` stores the bin index that it calculates in a new global array `part_bin_indices` (where `part_bin_indices[`i`]` is equal to the index of the bin in which the particle with ID i is currently located). The threads executing `arrange_part_ids_gpu` and `compute_forces_gpu` that later use this same bin index fetch it from `part_bin_indices` instead of manually recalculating it themselves.

Given that the number of FLOPs that can be performed in the same time as a memory access is generally very high on a GPU, this optimization didn't make a lot of sense theoretically; we had previously found that calculating the indices of the neighbors of each bin on-the-fly in `compute_forces_gpu` — rather than precalculating all bins' neighbor indices in `init_simulation`, storing them in global arrays in the GPU's memory, and fetching them from global arrays, and then fetching them from those arrays in `compute_forces_gpu` — increased our implementation's total run time by 1–2 seconds.

But, since it could be implemented and tested in just a few minutes, we gave this optimization a shot to see how it would affect the overall runtime of our implementation. While the performance improvement yielded by this optimization was relatively small — the runtime of our implementation on a 1,000,000–particle simulation decreased by less than 0.2 seconds — it was an improvement nonetheless, and we incorporated it into the final version of our implementation.

2.4.2 `bin_counts` Array Re-Use

In the basic version of our implementation described above, separate arrays (`bin_start_indices` and `bin_next_indices`, respectively) are used to store pointers to the beginning and the current "tail" of each bin's region of `part_ids`. The `cudaMemcpy` call that sets the contents of `bin_next_indices` to an exact copy of the contents of `bin_start_indices` consumes time during every single simulation step. An array separate from `bin_start_indices` is indeed indeed necessary to hold the "tail pointers" used to refill `part_ids` — `bin_start_indices` needs to be kept intact so that `compute_forces_gpu` can locate the beginnings and endpoints of different regions of `part_ids`. However, this second array doesn't need to be a copy of `bin_start_indices`; it's possible to instead use the array `bin_counts` — which is no longer needed after the call to `thrust::exclusive_scan` — to implicitly store the "tail pointers" for all of the different bins' regions of `part_ids`.

Thus, `bin_next_indices` is not present in the final version of our implementation. Consider a thread executing `arrange_part_ids_gpu` whose assigned particle is located in bin b. In the original version of our implemention, this thread would set `int index = atomicAdd(bin_next_indices[`b`], 1)` (e.g. atomically increment the `bin_next_indices` entry and set `index` equal to the entry's previous value), and then set `part_ids[index] = tid`. In the final version of our implementation, this thread sets
`int subindex = atomicAdd(bin_counts[`b`], -1)` (e.g. atomically *decrement* the `bin_counts` entry and set `index` equal to the entry's previous value), and then sets
`part_ids[(bin_start_indices[`b`] + subindex - 1)] = tid`. Implementing this optimization (and thus eliminating a `cudaMemcpy` call from every simulation step). from reduced the runtime of our implementation on a 1,000,000-particle simulation by 0.5–0.6 seconds.

2.5 Unsuccessful Optimizations

In this section, we briefly describe some optimizations that we attempted, but that failed to increase the performance of our CUDA implementation.

2.5.1 Bin Size Increase

While the optimal bin size for the serial shared-memory implementation of our particle simulation (from Homework 2.1) was r, we found through experimentation that a larger bin size of $4r$ yielded greater performance. This is because, since each thread was assigned one bin at a time rather than one particle at a time to work with, smaller bin sizes increased the number of bins without any particles, leaving many threads idle. However, for our CUDA implementation — in which threads are assigned work in units of single known particles rather than whole bins at a time — we experimentally determined that a bin size of $2r$ yielded the best performance.

2.5.2 Custom Exclusive Prefix Sum

We attempted to implement our own exclusive prefix scan operation (in essence, our own version of `thrust::exclusive_scan`) using the work– and space-efficient non-recursive exclusive scan algorithm described in Lecture 8. However, our implementation was not as fast as `thrust::exclusive_scan`, and was not used in the final version of our implementation.

3 Performance Analysis

Our CUDA implementation is substantially faster than our previous OpenMP and MPI implementations of the particle simulation program. Using 1 of Bridges-2's shared NVIDIA Tesla V100 GPUs, it completes a $1,000,000$-particle simulation in just 2.6 seconds. Since the OpenMP and MPI implementations were originally benchmarked on Cori's Knights Landing nodes, we ran scaling benchmarks on our OpenMP implementation on Bridges-2's Regular Memory AMD nodes; below, we compare those OpenMP scaling benchmarks to compare our CUDA implementation's scaling benchmarks.

3.1 Scaling and Comparison to Starter Code

We tested our CUDA implementation (represented by the blue squares in Figure 2) on various particle counts. It completed a $1,000$-particle simulation in 0.038 seconds and a $1,000,000$-particle simulation in 2.6 seconds. Interestingly, if we assume linear scaling behavior, the runtime of our CUDA implementation on $10,000$-particle and $100,000$-particle simulations does not fall on the line between the $1,000$-particle simulation runtime and the $1,000,000$-particle simulation runtime; it can be seen in Figure 2 that they would fall below the line. This makes the scaling behavior look exponential. Given that we'd expect the CUDA implementation's performance to increase at high particle counts, we initally thought this may be an artefact of using the GPU-shared nodes in Bridges2. However, we determined that this increase may be due to the increasing wait times in `cudaDeviceSynchronize` (which our code never calls explicitly), as seen in Section 3.3. It's possible that our implementation's performance is being impacted by contention for resources like memory and GPU cores that are shared with others using the same Bridges-2 node.

We also compared our optimized CUDA implementation with the starter code (represented by the orange diamonds in Figure 2). Across all of the particle counts we tested, our implementation performs substantially better. The gulf in performance between the two also increases as the particle count increases. The runtime of the starter code for a $1,000,000$-particle simulation is not shown in the figure, as it timed out a 30-minute interactive node; this indicates that the starter code's performance also degrades at higher particle counts.

3.2 Comparison to OpenMP Implementation

To properly compare our CUDA implementation to our other particle simulation implementations, we benchmarked our OpenMP algorithm on Bridges-2's Regular Memory nodes. Cori's Knights Landing nodes consist of a single socket containing an Intel Xeon Phi 7250 processor with 68 cores, while Bridges-2's Regular Memory nodes consist of two sockets each containing an AMD EPYC 7742 processor with 64 cores (for a total of 128 cores). Due to the long queue times for `sbatch` jobs, we only tested thread counts that were powers of two. We found that 64 was the optimal thread count. (We also found 64 to be the optimal thread count on Bridges-2's shared Regular Memory nodes.) The performance discrepancy between Bridges-2's RM and

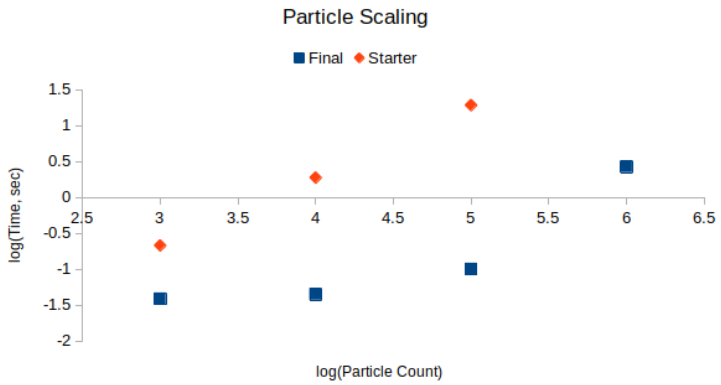

Figure 2:

RM-shared nodes may be due to the fact that RM-shared nodes are shared resources whose performance can be affected by other users' jobs.

Figure 3 compares the performance of the our OpenMP and CUDA implementations, with the OpenMP implementation using the optimal thread count of 64. At the lowest particle count of $1,000$, there is not much of a performance difference between the two implementations. However, as the number of particles increases, the CUDA implementation outclasses the OpenMP algorithm, with the delta between the two reaching its peak at $100,000$ particles. It is interesting to note that the performance gap decreases at $1,000,000$ particles, as we would expect the CUDA implementation's performance gains to be larger for larger numbers of particles. This discrepancy may be explained by the shared resource contention issue described above, the `cudaDeviceSynchronize` wait time issue described in Section 3.3, or some combination of the two.

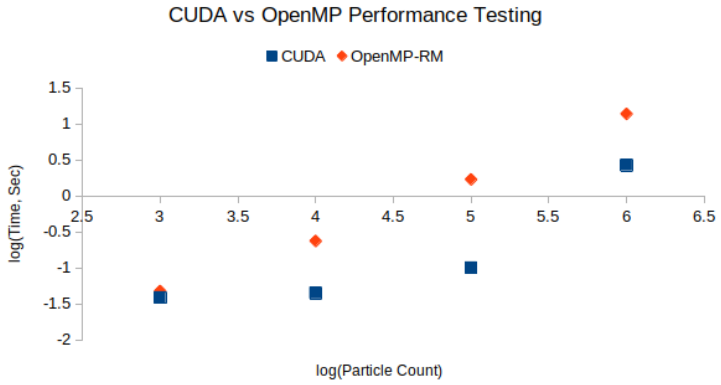

Figure 3:

3.3 Where Did the Time Go?

We used `nvprof` to analyze the runtime of the kernels and API calls in our CUDA implementation. Figures 4 and 5 show the percentages of the total runtime consumed by various GPU activities and API calls as the particle count is increased. Legends showing the most time-consuming activities are included in both figures. As shown in Figure 4, our implementation's runtime is consumed mostly by `compute_forces_gpu`, and that the portion of the overall runtime consumed by `compute_forces_gpu`'s increases significantly between the $100,000$– and $1,000,000$-particle simulations. This substantial increase in `compute_forces_gpu`'s percentage

of the total runtime mirrors a similar increase in `cudaDeviceSynchronize`'s percentage of the total runtime. This latter runtime percentage increase may explain why the runtime of our implementation does not follow a linear trend; the process needs to wait longer for all of the GPU threads to finish before moving onto the next kernel.

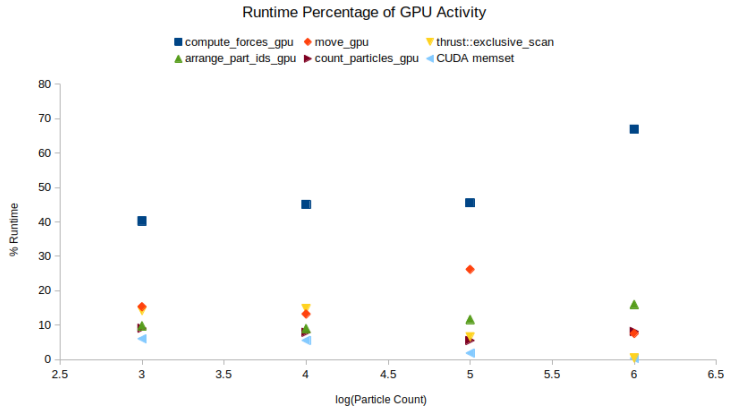

Figure 4: Percentage of runtime of different GPU activities

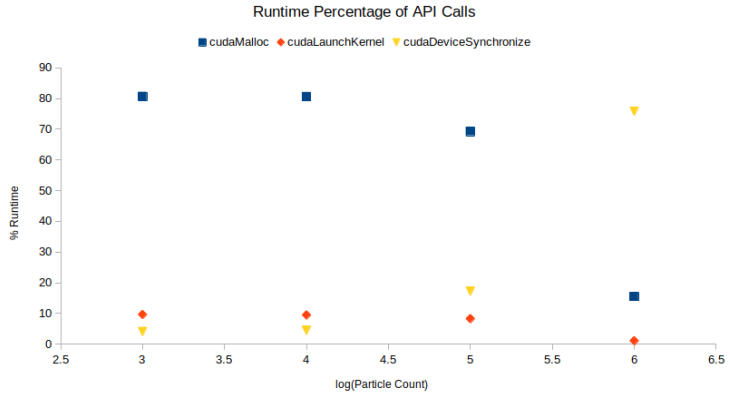

Figure 5: Percentage of runtime of

While we originally believed the cause of this synchronization time increase to be our `atomicAdd` calls, further investigation provied that this was not the case.

To better investigate the source of the synchronization time increases, we substituted all of the `atomicAdd` operations in the `compute_forces_gpu` function with a variety of similar "dummy" operations — a non-atomic read-add-save and a trivial read-add with no write to global memory. The results, shown in Figures 6 and 7, show that switching to non-atomic operations has very little impact on the the percentages of the total runtime consumed by `compute_forces_gpu` and `cudaDeviceSynchronize`. However, this trivial substitution significantly reduces the percentage of the overall runtime consumed by both of both `compute_forces_gpu` and `cudaDeviceSynchronize` (number of calls remains the same). Given that the percentage of the total runtime consumed by `cudaDeviceSynchronize` includes time spent waiting for threads to all finish a given task, this sharp increase indicates that, for larger particle counts, the time taken for the GPU to finish certain functions is longer than would be expected if assuming linear scaling behavior.

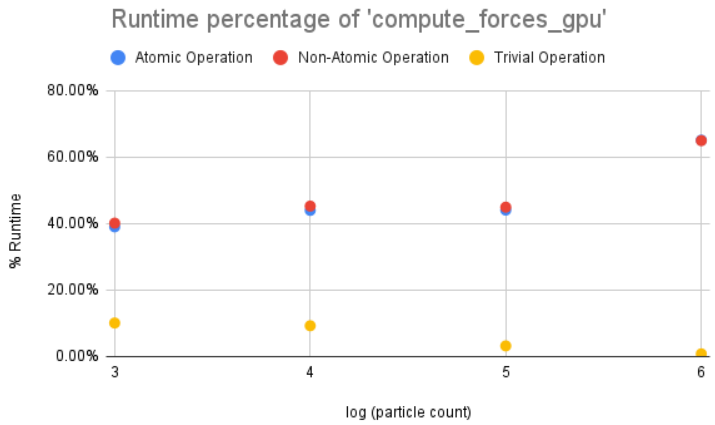

Figure 6: Percentage of overall runtime consumed by `compute_forces_gpu` GPU activities with `atomicAdd` calls substituted out

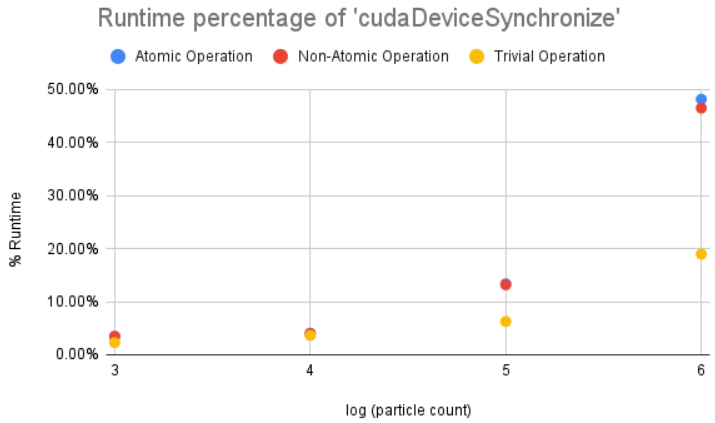

Figure 7: Percentage of overall runtime consumed by `cudaDeviceSynchronize` API calls with `atomicAdd` calls substituted out

4 Team Member Contributions

The members of our team each contributed to this report in many ways. The individual contributions of each time member are as follows:

- **Maya Lemmon-Kishi** implemented `compute_forces_gpu` and the custom exclusive prefix sum, and ran performance and profiling analyses.
- **Zach Van Hyfte** implemented the GPU-optimized bin data structure and re-binning process, and implemented most of the performance optimizations.
- **Minan Wu** debugged, tested, and profiled the code.