Carlos Moustafa (aem278)
aem278@perlmutter.nersc.gov

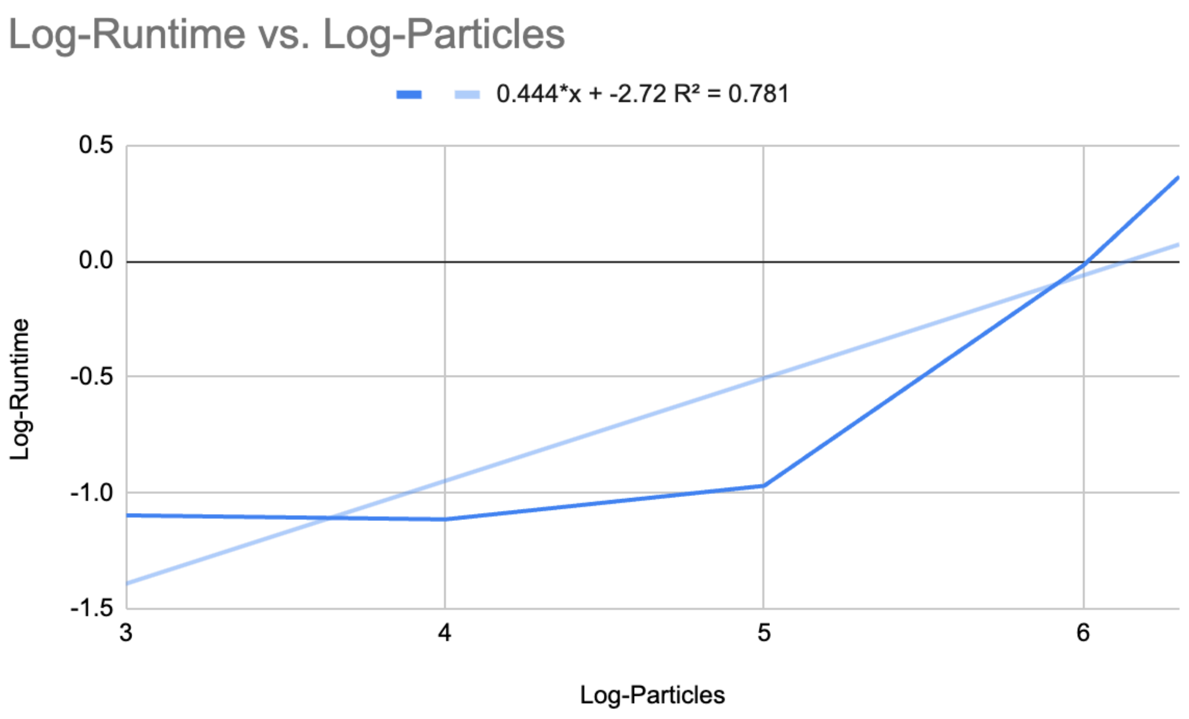
Results

Below are results measured on a single run on Perlmutter's A100 GPU.

Data

Particles	**Runtime**	**Log-Particles**	**Log-Runtime**
1000	0.08	3	-1.096910013
10000	0.0768714	4	-1.114235209
100000	0.107306	5	-0.9693759938
1000000	0.958382	6	-0.01846135165
2000000	2.32035	6.301029996	0.3655534985

Table



I observe that this plot does not appear linear, with a sub-optimal R^2 value of about 0.781. I predict this is due to noise, and the unpredictable waiting times threads experience in the **Sorting Particles** task, detailed more below.

Data Structures & Synchronization Methods & Designs

The data structures I used were two C arrays: one to store a prefix-sum over the population of each bin, and another to store an array of the particles sorted by the ID of the bin they're in.

Synchronization

This system is not embarrassingly parallel and requires some synchronization between GPU threads. One core requirement for synchronization across GPU threads is computing each bin's population in each timestep. By design, each thread processes a single particle at a time to compute these populations. Since I chose a single C array to store these populations (with each entry corresponding to a single bin), this array is considered shared memory, so I used synchronization when writing to it.

Binning

Specifically, for each particle, I compute its bin ID and atomically increment the population stored for that bin. Being an atomic operation, there may be contention and waiting for other threads to complete their increment for that bin, thus introducing possible latency.

Sorting Particles by Bin

After binning, I perform a parallel prefix-sum over these counts to obtain a list of indices that essentially partition the particle array. Then, to insert the particles into this shared array with parallelism, it is necessary to utilize synchronization to prevent race conditions. To do this, I utilized atomicCAS from the cuda library. It runs a loop starting from the beginning of the subarray (i.e. bin) the particle is in, tries to write the particle ID to that location if there's not already one written by another thread, and the loop continues iterating until it reserves an index for the particle before any other thread does.

A downside I note about this method is that it has a worst-case complexity that is linear in the number of particles. In reality, the number of iterations (and stalls) each thread must perform with this method is highly variable: it is at-most the number of particles in the bin, and exactly equal to the number of particles in the bin that have already been processed before the current thread obtaining the lock associated with that memory address. This is the “unpredictable waiting times” referenced in the **Results** section.

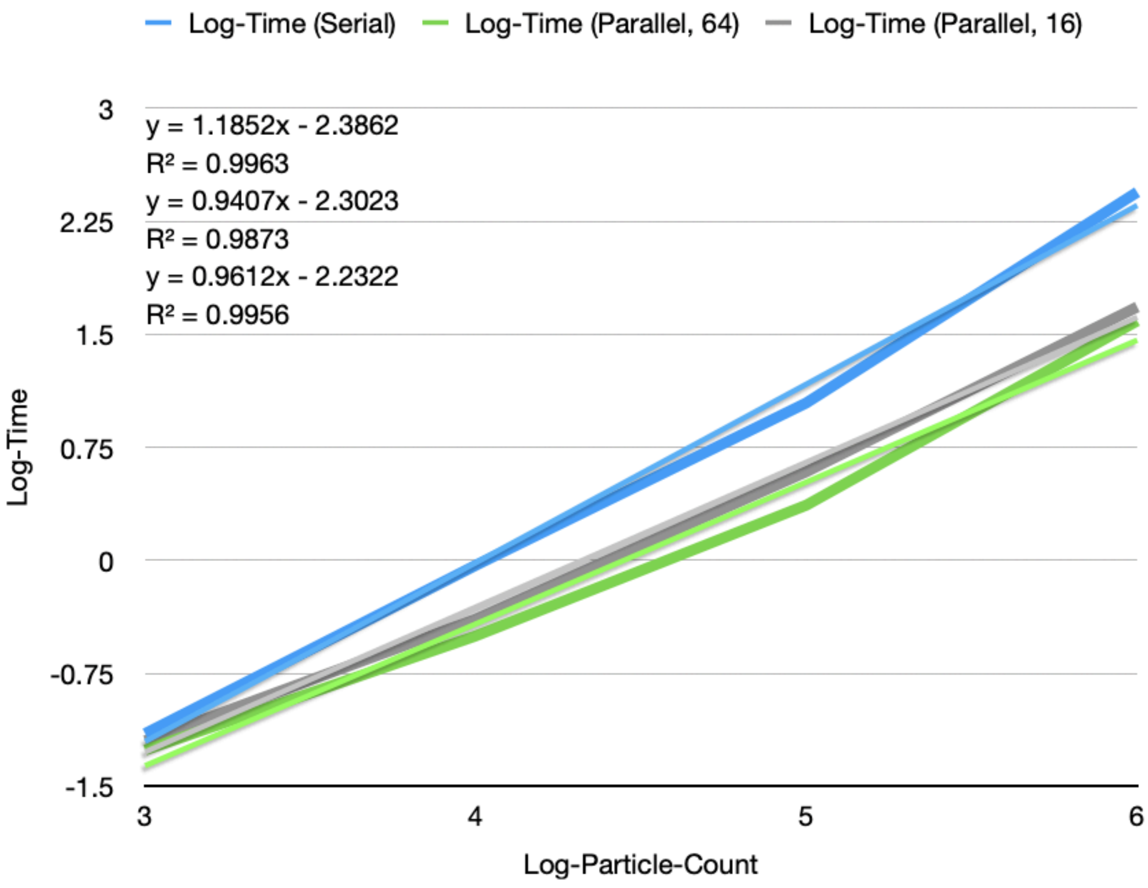
An alternative method** for this I implemented operates linearly* in the number of *bins* at the cost of an additional array (which is worth the tradeoff since there are typically many more particles than bins). This array stores the highest index of the bin’s subarray that any thread has written to, and is initialized to -1 each timestep. Then, in one atomic operation, each thread reads from this array and increments its value, and receives the value previously stored at the location. With this method, no while loop is needed, so each thread performs constant-time work for this task. I hypothesize this method would yield a higher R^2 value in the log-log plot for this reason.

*This method is linear in the number of bins because of the need to re-initialize all the values in the array to -1 each timestep. However, we note that this initialization is embarrassingly parallel, and so the performance is even better theoretically.

**I had trouble debugging the full system when I had this method implemented, for reasons that I believe were not caused by this step. Due to time constraints, after fixing the bugs I was left with the less-desirable solution first-described.

Runtime Analysis

Below is my plot from the previous assignment for comparison and interpretation:



As we can see, the best-performing regime tested took around log(1.5s) to simulate 1,000,000 particles. We note that this GPU-accelerated version performs significantly better, simulating 2,000,000 particles in an order-of-magnitude less time.

This is not too surprising, since this GPU regime utilizes 256 GPU threads, 4x more logical parallelism. That being said, it is still worth noting that the synchronization methods used in this GPU regime are not magnitudes-worse than the synchronization methods I used in OpenMP.

Additionally, I note that my observations show that a majority of runtime is spent in synchronization: specifically, the two atomic operations, especially the atomicCAS iterations in the sorting step.

Discussion

If time permitted, I would have pursued more detailed metrics on thread contention showed in class, including longest-idle-time, average-idle-time, and min-idle-time. This would give a fuller view of the contention caused by the atomic operations in the two synchronization-dependent tasks discussed earlier.

Additionally, I would have pursued more meticulous use of the blocks-and-warps architecture of Perlmutter’s GPU. For example, I considered the effectiveness of parallelizing over bins instead, with each block/warp holding a copy of all particle-data needed for all computation in every bin processed in the warp. However, the starter code seemed to incentivize parallelizing across particles, so I stuck with that.