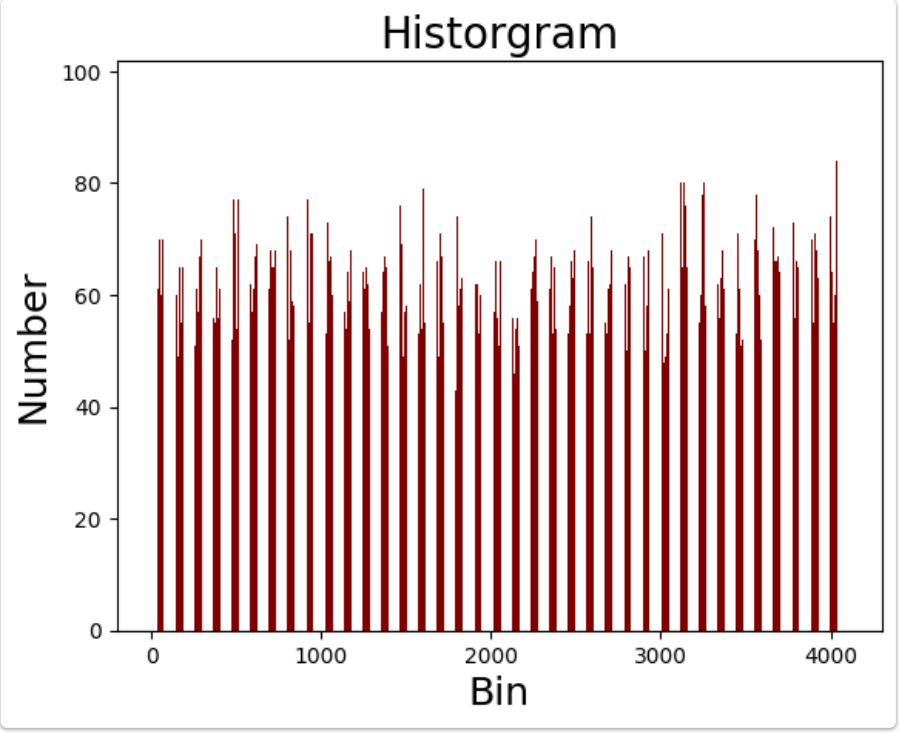
Assignment 3

Exercise 1 - Histogram and Atomics

1. Describe all optimizations you tried regardless of whether you committed to them or abandoned them and whether they improved or hurt performance.
   We try `__shared__`, `__syncthreads()` and `atomicAdd()`.
2. Which optimizations you chose in the end and why?
   We use `__shared__ unsigned int localHistogram[NUM_BINS]` to define a share memory between each thread so as to improve the performance. And `__syncthreads()` is used after initializing the 0 value to the localHistogram also after incrementing 1 to the localHistogram. If the code is without this, we can not ensure that all threads have finished updating their portion of share memory. Lastly, `atomicAdd` is applied while we increment the value to the localHistogram and update the value in global histogram, this can prevent the race condition happening so that we are able to gain the correct results. The code snippet of the important part that we've mentioned is shown below.

```
__shared__ unsigned int localHistogram[NUM_BINS];
...
// Initialize local histogram to zero
...
__syncthreads();
...
// Increment local histogram using shared memory
if (idx<num_elements){
  atomicAdd(&localHistogram[input[idx]], 1);
}
__syncthreads();
// Update global histogram using atomic operations
if (threadIdx.x == 0){
  for (int i = 0; i < num_bins; i ++){
    atomicAdd(&bins[i], localHistogram[i]);
  }
}
```

3. How many global memory reads are being performed by your kernel? Explain
   There is one global memory read in the line `atomicAdd(&localHistogram[input[idx]], 1)`, and the total time will be the number of threads that execute this function.
4. How many atomic operations are being performed by your kernel? Explain
   There are two atomic operations in our kernel, which are `atomicAdd(&localHistogram[input[idx]], 1)` and `atomicAdd(&bins[i], localHistogram[i])` respectively. The first one is used to increment local histogram using shared memory while the latter one is used for updating the global histogram.
5. How much shared memory is used in your code? Explain
   The share memory is declared as the size of `localHistogram[NUM_BINS]` Therefore, the total shared memory usage for a block is `NUM_BINS * sizeof(unsigned int)` bytes. And the total shared memory usage for the entire kernel launch, we have to further multiply this by the number of blocks in the grid.
6. How would the value distribution of the input array affect the contention among threads? For instance, what contentions would you expect if every element in the array has the same value?
   Each thread will attempt to update the same shared memory and global memory locations and increase the time for executing the atomic operation, which results in a performance drop in using parallel computing.
7. Plot a histogram generated by your code and specify your input length, thread block and grid.
   Our input length is set as 257864 , thread per block is 1024 and grid size is.



8. For a input array of 1024 elements, profile with Nvidia Nsight and report Shared Memory Configuration Size and Achieved Occupancy. Did Nvsight report any potential performance issues?
   The shared memory configuration size is 65.54 Kbyte while the achieved occupancy is only 3.5%.
   Nvsight suggests that

> *This kernel grid is too small to fill the available resources on this device, resulting in only 0.1 full waves across all SMs.*

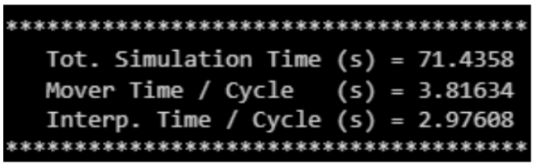
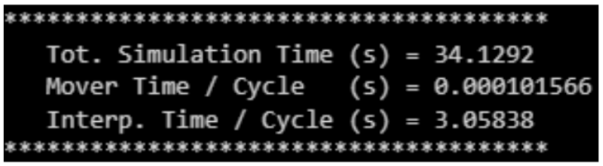
> *The grid for this launch is configured to execute only 16 blocks, which is less than the GPU's 40 multiprocessors. This can underutilize some multiprocessors. If you do not intend to execute this kernel concurrently with other workloads, consider reducing the block size to have at least one block per multiprocessor or increase the size of the grid to fully utilize the available hardware resources.*

which indicates that we should change our configuration of grid size and block size to better match the resources on the GPU, otherwise we lose the efficacy of leveraging the computational power of multiprocessors.

Exercise 2 - A Particle Simulation Application

1. Describe the environment you used, what changes you made to the Makefile, and how you ran the simulation.
   We used Google Colab to run the program. We changed the cuda arch in makefile to sm_75 to fit the gpu of google colab. We ran the simulation by following the procedure in the sputniPIC slides.
2. Describe your design of the GPU implementation of mover_PC() briefly.
   We designed a kernel function to parallelize the calculation of particles' kinematic, and implemented the mover_PC function to allocate memory, copy data and execute the kernel.
3. Compare the output of both CPU and GPU implementation to guarantee that your GPU implementations produce correct answers.
   We compare three output files "rhoe_10.vtk", "rho_net_10.vtk", "rhoi_10.vtk" that were generated by the original program and our new program. Due to the round-off error in the calculation, there are minor numerical differences between them. By taking the absolute value of the differences, we observe that the error is restricted to below 0.015
4. Compare the execution time of your GPU implementation with its CPU version.
   We may see that our modification of mover_PC has greatly sped up the move time / cycle. The program can be

further sped up by optimizing the execution time of interpolation by gpu.

CPU	GPU
	

links: https://github.com/LaiYuHong/DD2360HT23/

contribution: we collaborated to finish both the implementation of code and answering questions.