Applied GPU Programming - Assignment 3 - Group 19

Group members and contributions

We did the assignment together and in-person, therefore both of us contributed the same amount of work.

- Diogo Gaspar - dgaspar@kth.se (mailto:dgaspar@kth.se) - 50%
- Diogo Melita - diogogm@kth.se (mailto:diogogm@kth.se) - 50%

Exercise 1

1. Optimizations Tried

After revising the material for module III, we decided to try to use atomic operations and shared memory to enhance the efficiency of the histogram calculation. Atomic operations allow us to have thread-safe increments of the histogram bins, while shared memory minimizes the need for global memory accesses. Together, they significantly boost the performance of the histogram computation.

2. Final Optimizations Chosen

We ended up adopting the aforementioned techniques to optimize the calculation of the histogram's bins.

3. Global Memory Reads

Each thread reads elements from the `input` array in a strided pattern; assuming N elements and T threads, there should be $N + num\_bins$ reads, since each input element is read once, for the histogram calculation, and then we read all bins for the saturation procedure.

4. Atomic Operations

Each input element corresponds to one atomic addition in shared memory during local histogram computation. After the local bins are accumulated, threads write to global bins:

- Atomic operations in **shared** memory are N, one per input element.
- Atomic operations in **global** memory are $NUM\_BINS \times B$, where B is the number of thread blocks, as each block merges its local histogram to global bins.

5. Shared Memory Usage

Each block allocates an array, `shared_bins`, of size NUM_BINS. Thus, the size per block will be of $NUM\_BINS \times 4 = 16\,384$ bytes, assuming 4 bytes per unsigned int, and the total shared memory will be of $Blocks\_per\_SM \times 16384$, limited by the GPU's shared memory per SM.

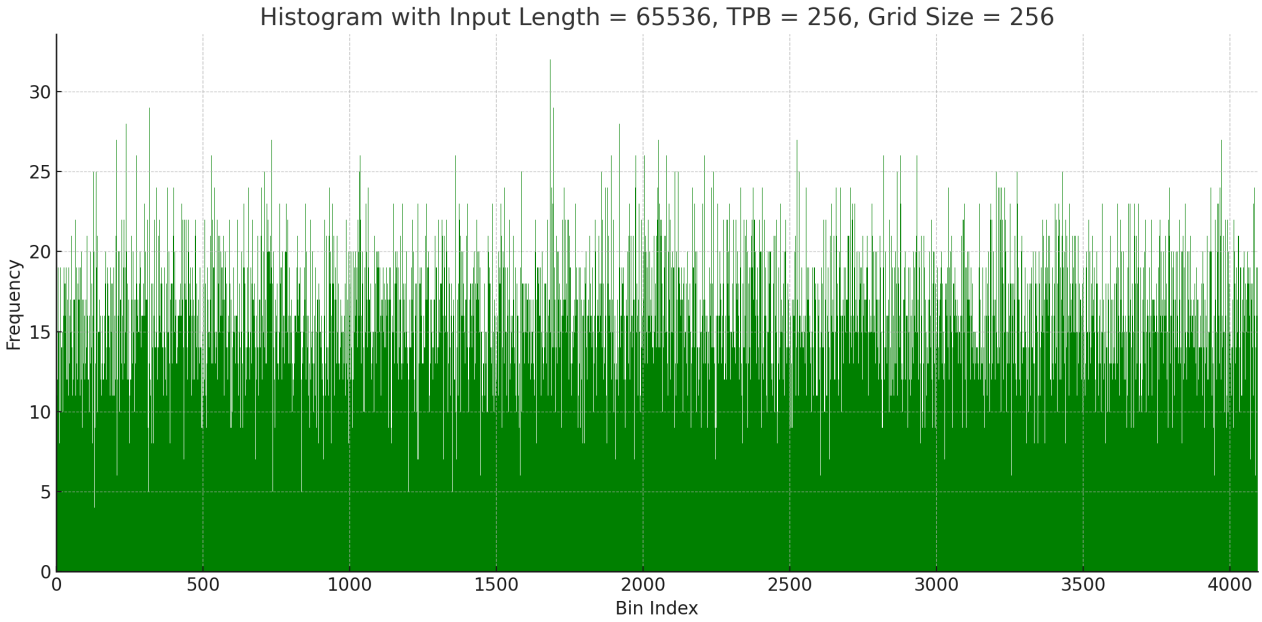
6. Effect of Input Distribution

There are two main scenarios here, uniform and skewed distributions. With uniform ones, there should be minimal contention, of course, having as input values would spread across bins evenly. With skewed ones (e.g., many elements with the same value), however, some problems would arise – in particular, the possibility of high contention for specific bins, increasing the overhead of atomic operations in shared memory and global memory, as well as the performance degradation due to serialization.

7. Histogram Plot

Below is the histogram plot for an input length of 2^{16} $(65,536)$ elements:

- **Input Length**: $2^{16} = 65,536$
- **Thread Block Size (TPB)**: 256
- **Grid Size**: $\lceil 65,536/256 \rceil = 256$



The histogram reflects, as expected, a uniform random distribution across the 4096 bins.

8. Profiling

For a block size of 256, we got the output as shown below, for the histrogram calculation kernel, which is the important one. It is possible to see that NSIGHT reports some problems regarding the block size and how its value might underutilize some multiprocessors.

After some tests, we found that for bigger values of block size, such as 512 or 1024, the Achieved Occupancy increases, proportionally, doubling from 256 to 512 and from 512 to 1024, respectively, which is expected. On the other hand, the Shared Memory Configuration Size reduces in half when changing the TPB for one of this two values aformentioned, as expected. It is also trivial to understand that reducing the block size will lead to a lower Achieved Occupancy and to an increased Shared Memory Configuration Size.

```
Section: Launch Statistics
-------------------------------- --------------- ---------------
Metric Name                          Metric Unit    Metric Value
-------------------------------- --------------- ---------------
Block Size                                                   256
Function Cache Configuration                      CachePreferNone
Grid Size                                                      4
Registers Per Thread             register/thread              16
Shared Memory Configuration Size           Kbyte           65.54
Driver Shared Memory Per Block        byte/block               0
Dynamic Shared Memory Per Block       byte/block               0
Static Shared Memory Per Block       Kbyte/block           16.38
Threads                                    thread           1,024
Waves Per SM                                                0.03
-------------------------------- --------------- ---------------

OPT   Estimated Speedup: 90%
      The grid for this launch is configured to execute only 4 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. See the
      Hardware Model (https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#metrics-hw-model)
      description for more details on launch configurations.

Section: Occupancy
------------------------------ ----------- ------------
Metric Name                    Metric Unit Metric Value
------------------------------ ----------- ------------
Block Limit SM                       block           16
Block Limit Registers                block           16
Block Limit Shared Mem               block            4
Block Limit Warps                    block            4
Theoretical Active Warps per SM       warp           32
Theoretical Occupancy                    %          100
Achieved Occupancy                       %        24.83
Achieved Active Warps Per SM          warp         7.95
------------------------------ ----------- ------------

OPT   Estimated Speedup: 75.17%
      This kernel's theoretical occupancy is not impacted by any block limit. The difference between calculated
      theoretical (100.0%) and measured achieved occupancy (24.8%) can be the result of warp scheduling overheads
      or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block
      as well as across blocks of the same kernel. See the CUDA Best Practices Guide
      (https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#occupancy) for more details on
      optimizing occupancy.
```

Exercise 2

Changes Description

We ran our experiments on Google Colab, with a T4 GPU. Regarding the Makefile, the only change we made was to change `ARCH` from `sm_30` to `sm_75`, to correctly reflect the GPU architecture in use; we consulted the required information here (https://arnon.dk/matching-sm-architectures-arch-and-gencode-for-various-nvidia-cards/).

As for running the simulations, we simply ran `make` and then `./bin/sputniPIC.out inputfiles/GEM_2D.inp` for both CPU and GPU configurations.

GPU Implementation Design

In our GPU implementation of the `mover_PC` function, the core computation is

moved into a CUDA kernel called `particle_kernel`, where each thread processes a single particle simultaneously, rather than the sequential processing of the CPU version.
The implementation also introduces a couple of helper functions for GPU memory management: `h_part`, which handles allocation and copying of particle data to the GPU, and `retrieve_particles`, which manages the return transfer and cleanup. To optimize GPU memory access patterns, the code we wrote also converts 3D array structures to flattened 1D arrays with calculated indices.
We also reorganized the subcycling structure, moving the particle loop into the GPU kernel, while maintaining the subcycle loop on the host. Each subcycle launches the kernel again with updated particle positions.

CPU vs GPU correctness comparison

We have confirmed that the outputs match, thus the GPU implementation should produce correct answers.

CPU vs GPU execution time comparison

As per the image above, we can see that the GPU implementation produces a 2x speedup, halving the total execution time. Furthermore, we also see a huge decrease in mover time per cycle, from 3.42s in the CPU implementation to 0.25s in the GPU one.