EECS468 Lab 3 Journal

Adam Pollack and Gregory Leung

CPU Time: 11.08, 11.07

No Optimizations: Initial Solution

GPU Time: 12.269, 12.26 (No speedup compared to CPU)

Number of Blocks: 15936
Number of Threads: 1024
Goal: To solve the histogram problem using the GPU (naive implementation)
Kernel: Computes global id and uses the function atomicAddCustom which is based on the post here (From the NVIDIA developers forum) to increment bins based on the values in the input array.
Notes:

- Launched the number of threads required to compute over the entire input. This was equal to INPUT_WIDTH * INPUT_HEIGHT which is 3984 x 4096 = 16,318,464. Each block can contain up to 1024 threads so we needed 16,318,464/1024 = 15,936 blocks to compute the entire result.

Hours to Complete: ~5 hours (including setup code)

Optimization 1

GPU Time: 11.327, 11.31 (No speedup compared to CPU)

Number of Blocks: 32, 512, **1024**, 2048
Number of Threads: 1024
Goal: To improve use of each thread.
Kernel: Accesses input values over a stride length of the total number of threads. This means we launch fewer threads, but use them more often. Still using atomicAddCustom as the atomic operation.
Notes:

- Tried with multiple different grid dimensions, ran closer to 14 seconds for 32 and 2048 blocks, but ran around 11.5 seconds for 512 and 1024 blocks.
- Times recorded are for 1024 blocks per grid
- For simplicity, the number of threads was equal to the number of bins in our histogram

Hours to Complete: ~0.5 hours

Optimization 2

GPU Time: 5.395, 5.39 (2.054x speedup from CPU time)

Number of Blocks: 1024
Number of Threads: 1024
Goal: To improve the custom atomic operations used.
Kernel: Improved the speed of the custom atomic operation used to increment the values in bins by introducing a new function: atomicIncCustom() which adds 1 to the value at the address passed into it. Still has a significant amount of thread divergence, but relies on fewer calls to the native CUDA atomicCAS() function. Everything else is the same as Optimization 1.
Notes:

- Significant speedup

Hours to Complete: ~7 hours

Optimization 3

GPU Time: 4.485, 4.48 (2.470x speedup from CPU time)

Number of Blocks: 16
Number of Threads: 256
Goal: To improve the computation time by varying grid and block size.
Kernel: Same as in Optimization 2.
Notes:

- Varied grid and block size until computation time improved.
- Using 257 threads per block instead of 256 sped up computation time by ~0.1s

Hours to Complete: ~1 hours

Optimization 4 (Kernel: HistogramFastest)

GPU Time: 3.569, 3.57 (3.105x speedup from CPU time)

Number of Blocks: 15
Number of Threads: 257
Goal: To improve the computation time by varying grid and block size to eliminate bank conflicts.
Kernel: The kernel for this optimization is HistogramFastest. Same as in Optimization 2.
Notes:

- Using 257 threads per block instead of 256 and 15 blocks per grid instead of 16 sped up computation by ~0.8s.

Hours to Complete: ~0.1 hours

Optimization 5 (Kernel: HistogramShared)

GPU Time: 9.022, 9.01 (1.228x speedup from CPU time)

Number of Blocks: 32
Number of Threads: 257
Goal: To implement a shared memory solution to decrease read times from global memory.
Kernel: The Kernel for this solution is HistogramShared. This kernel allocates shared memory for a partial histogram in each block of threads, computes each partial histogram, then combines them into global memory. Using atomicIncCustom and atomicAddCustom as the atomic operations. Both custom atomic functions take advantage of the fact that the number will never overflow past 255.
Notes:

- We varied block and grid size until we found this solution. The best grid size seemed to be 16 or 32 blocks and the best block size seemed to be around 256 or 512 threads.
- Using 257 threads per block instead of 256 sped up computation time by ~0.25s
- This optimization is faster than the original naive solution, but is slower than our non-shared memory optimization.

Hours to Complete: ~10 hours

Optimization 6

GPU Time: 8.614, 8.61 (1.286x speedup from CPU time)

Number of Blocks: 15
Number of Threads: 297
Goal: To improve the shared memory solution
Kernel: Modified the number of elements in the shared histogram to be equal to the number of warps (WARP_COUNT 6) times BIN_COUNT (1024). This idea was taken from the histogram example in the NVIDIA CUDA sample code.
Notes:

- Varied block size and grid size until at a faster solution
- The value for WARP_COUNT was found using the CUDA occupancy calculator

Hours to Complete: ~2 hours

Optimization 7 (Kernel: HistogramSharedCoalesced)

GPU Time: 7.24, 7.245 (1.53x speedup from CPU time)

Number of Blocks: 16
Number of Threads: 256

Goal: To implement coalesced memory access to input data (and by side effect maybe avoid bank conflicts depending on the dimension configuration).
Kernel: The kernel for this solution is HistogramSharedCoalesced. Threads now access data from consecutive points in the array instead of strided access.
Hours to Complete: ~6 hours

Optimization 8

GPU Time: 4.236, 4.23 (2.616x speedup from CPU time)

Number of Blocks: 13
Number of Threads: 257
Goal: To apply the same techniques as above to tweak the number of blocks and threads.
Kernel: Modified the number of elements in the shared histogram to be equal to the number of warps (WARP_COUNT 6) times BIN_COUNT (1024). This idea was taken from the histogram example in the NVIDIA CUDA sample code.
Hours to Complete: ~1 hour

Optimization 9

GPU Time: 4.224, 4.22 (2.623x speedup from CPU time)

Number of Blocks: 13
Number of Threads: 257
Goal: Testing loop unrolling with #pragma unroll
Kernel: Added #pragma unroll before every loop.
Notes:

- No significant speed up from previous iterations (~0.1 seconds)

Hours to Complete: ~0.1 hours

Final Notes

Our best optimization (Optimization 4) resulted in a speedup of 3.094x as compared to the CPU computation time. Our best shared memory implementation was Optimization 9 which resulted in a 2.623x speedup compared to the CPU time.