COMP\_SCI 368 2/20/2023

**Lab 3 Report**

NOTE: All times discussed in this report refer to the time taken to compute a histogram 1000 times for a random input. The seed was set such that the random input was the same irrespective of the (optimized) kernel executing it.

ANOTHER NOTE: Right now, we have a call to cudaMemset() inside of opt\_2dhisto(). Supposedly, all memory allocations and transfers are to be done outside of that function, so how do we reset the bins in global memory to ensure we pass the test case?

Baseline/Reference Information

* CPU Reference Time: 12.3809 seconds
* Baseline Kernel Time: 0.56098 seconds (22.070x speedup over CPU reference)
  + Implements Parallel Strategy #2 from Lecture 10, where each block computes its own shared histogram.
  + Fastest time achieved when input was transformed to 1D and using blocks were sized 1024x1
  + Pros
    - Coalesced memory access
    - No bank conflicts when merging partial histogram back into global mem
  + Cons
    - Bank conflicts when creating own subhistogram
  + Warp votes only made things slower (6.309x slower than the time shown above)
  + Overall, we had a very hard time allocating and copying data over to the GPU; some bugs deceived us for the majority of the weekend. We spent at least 24 hours combined getting this implementation to work properly.

As follows are all alternative optimizations we tried:

* No Shared Memory Kernel: 0.28103 seconds (44.0554x speedup from CPU reference)
  + None of the strategies shown in lecture (easily) solved the problem of bank conflicts in the shared memory array. We had an idea — instead of using the coloring method, why not remove shared memory entirely?
  + This kernel is extremely simple, but surprisingly effective. It spawns one thread per element of the input, and updates the corresponding bin with an atomicAdd()
  + This strategy coalesces memory reads, and experiences no bank conflicts because shared memory is never used. We again found the best results with blocks of size 1024x1. This optimization was fairly quick to write, but it didn’t come to us at first.
* Always atomicAdd() Kernel: 2.36415 seconds (5.2369x speedup from CPU reference)
  + Since the No Shared Memory Kernel was faster, we built this off of that one
  + Idea: remove divergent code by always calling atomicAdd(), and have the host post-process overflow/large numbers
  + Instead, this significantly slowed down the kernel (8.412x slower than No Shared Mem Kernel)
    - Delay from high quantities of adds for certain bins must outweigh thread divergence
    - Plus, with the way pixels are generated here, there might not even be much thread divergence because similar bins tend to be maxed out.
  + This optimization was very easy to write, we just had to write two if statements into our already-existing code.
* After this optimization, we decided against trying other implementations shown in class given how much time it took us to implement them. Plus, the No Shared Memory Kernel was just really fast!
  + We couldn’t see how to beat the simplicity of the No Shared Memory Kernel
    - Parallel implementation #1 requires every thread to process multiple elements, which could be coalesced, but why add them to your local bins once and then update the global bins after? You still have to call atomicAdd(). Going directly there and using full occupancy seems like a better strategy.
  + Or how other methods would fit in to it
    - Can’t take advantage of ILP since we aren’t doing any arithmetic instructions inside for loops
    - We did try TLP, but it only seemed to make the kernel take longer
    - Getting more registers per thread makes no sense if we aren’t doing anything expensive with them, so using lots of threads is okay
* We did try one other optimization on top of our existing baseline implementation by shuffling the inputs first. This is accomplished by swapping the y and x axis, so elements next to each other in memory are no longer highly correlated. Hopefully, this decreases access conflicts when binning within a block and speeds up the atomic operation.
  + We initially performed a naive implementation where we simply ran an additional kernel to read from global memory and write the shuffled input to global memory as well, after which we used our baseline kernel with slightly modified dimensions. Unfortunately, this implementation actually slowed down opt2dhisto by ~10% because the overhead from the reads and writes to global memory were not worth the advantage in binning.
  + Next, we tried implementing the shuffle using shared memory instead. Here, we may have gained a very slight performance advantage, but it is narrow enough to be inconsistent from test to test. We think that this approach doesn’t introduce as much overhead as the prior implementation, which is why we don’t lose performance. However, we are shuffling within 32x32 blocks and then writing out contiguously to ensure coalescence, so the shuffling is probably not that significant and therefore not significantly impacting the bin increment operations. Also, we couldn’t get the test to pass. For some reason, the kernel is slightly overcounting a few bins. Since we made no changes to the actual histogram kernel, we think maybe the dimensions are a bit off in our shuffle kernel so not all the data is being transferred to the new shuffled input, but we did not have enough time to confirm.
  + I would like to be able to perform a more thorough shuffle (flip x and y for the whole input not just within subsets), but I’m uncertain how to achieve this given the constraints:
    - Shared memory is limited in size, or we could do it all in one block with coalesced writes and reads
    - If a block is reading a subset of the input and then placing the inputs where they would go if you flipped the axes, then the writes are not coalesced. If you deliberately pick inputs to read that you know will eventually be written together, then your read will not be coalesced.
  + The global memory shuffle was easy to implement, but the shared memory shuffle took a few hours to think through and avoid memory access errors, and then many more hours attempting to get functional. A bit frustrating since I think the solution is very close, only a couple of bins are off by a small amount.