Jeremy Midvidy and Julie Weingart
EECS 368, Winter 2019
Lab 3 Report

About our Submission:

Most of our code can be found in the “opt_2dhisto.cu” (and associated header files). Additionally, we wrote some helper functions in “testharness.cpp” to make parts of input processing a little easier.

About out Approach:

Our implementation implements Strategy II (thread interleaving and memory coalescing), as well as a few more optimizations that I will discuss later in the report. After noticing that the histogram was a fixed value of 3984x4096 (16,218,464) elements, we decided to set up a grid of 16x1 blocks with 1024x1 threads per block. This allowed use to more easily implement the histogram function, because the histogram itself only had 1024 buckets. With the same number of threads per block as number of buckets in the histogram, one thread per block could be assigned a bucket to keep track of and then deposit from shared memory into global memory. This simplification made many parts of the lab both easier and faster.

Likewise, to make the work faster and simpler on our and as well as faster in the kernel, we flattened the 2D array created by “testharness.cpp” to a 1D array, and then proceed from there. Most of our logic follows the Strategy II that was discussed in class. This logic can be easily followed along with by looking at our code.

Part One - Correctness:

With running the ./lab3 file in the /release/ directory, our code produces “TEST PASSED” for all possible input arguments.

Our initial naive implementation took around as long as the CPU version, or about 11 seconds. With our optimizations, our GPU kernel took only 0.01 to do the entire computation, while the CPU version took 11.1 seconds. We are pleased with this result.

```
[jam658@batman labs]$ ./bin/linux/release/lab3
  Timing 'ref_2dhisto' started
    GetTimeOfDay Time (for 1000 iterations) = 11.116
    Clock Time       (for 1000 iterations) = 11.1
  Timing 'ref_2dhisto' ended
  Timing 'opt_2dhisto' started
    GetTimeOfDay Time (for 1000 iterations) = 0.003
    Clock Time       (for 1000 iterations) = 0.01
  Timing 'opt_2dhisto' ended

  Test PASSED
```

Part Two - Optimization:

Since our code followed Strategy II, the most significant optimization that our code makes is ***interleaving***. This method executes at a much higher throughput than *partitioning* (e.g. Strategy I) because memory accesses are coalesced across threads rather than split up. The interleaving is achieved through a sliding window across each block: in each thread, a global thread ID is calculated, and then each thread (nearly all but not every to be precise) iterates 996 times through the input data in global memory such that, with the combined effort in each 1024 threads across all 16 blocks, the entire 3984x4096 input matrix is scanned and frequencies are accumulated.

More specifically, each iteration jumps from current index(starting at the global thread ID of the current thread) to the 16,384 + the previous index. This value of 16,384 represents the total number of threads in the GPU (e.g. 16x1024). Moreover, this value represents the distance from the current index in global memory to the next index in global memory that a given thread will have to jump to read the next character from the input matrix that has not already been read by another thread. In other words, since there are 16,384 threads reading characters at once, after a thread reads a character, the next non-read character for it to read will be 16,384 indices away. Since the index strides along this increment, this value is called the *stride* value.

So, this process of interleaving and memory coalescing among neighboring threads (threads that are next door neighbors remain next door neighbors after each stride as they access global memory), leads to a higher efficiency in our CUDA program.

Our program also makes another signification optimization, such that it implements ***privatization*** among histograms. In this, each block has its own private histogram. Since each block has 1024 threads, and the histogram has 1024 bins, this means that each block can effectively write to its global histogram with just one write per thread! No iterations necessary. Here, each thread (within a given block) has a thread index (e.g. threadIdx.x) that maps to an index value between 0-1023 in the histogram array for that block. Since each block has its private histogram (stored in __shared__) memory within its block, after all threads have completed their scan of the input array, then threads sync up, and then all execute 1 atomicAdd from the shared histogram to the global histogram. Here, each thread takes the value of its threadIdx.x at the shared histogram within its block, and writes that value to global memory in the histogram of the entire grid. This allows each thread to only have to access 1 element in the global histogram in order to convey the findings of its entire block! It's clear why this procedure greatly reduces the number of bank conflicts that would have occurred without the privatization of histograms within each block.

Finally, our implementation utilizes CUDA's atomicAdd() to ensure that all increments or bin updates are ***serialized***. Since race conditions occur in many scenarios within the execution of our program, the use of atomic functions ensures that all functions will be realized in the final result of each computation, rather than risk the one non-atomic function canceling out the read-write-modify of another atomic function.

Part Three - Future Optimizations:

One more optimization to include in our program would be an implementation of *aggregation*. Since some data sets can have localized concentration of similar inputs, using this knowledge about the feature set could make histogram scanning even more optimal than our current implementation. This knowledge can reduce the total number of atomic operations needed and make the throughput of the program even better. Since our data was a randomly generated 2D matrix of ints, it did not make a lot of sense to pursue this option since the data was not notably localized in any way.